

Proceedings of the
International
Machine Learning Workshop

June 22-24, 1983

Allerton House
Monticello, Illinois



Sponsored
by

The Office of Naval Research
under Grant No. N00014 83G0030

and

The Department of Computer Science
University of Illinois at Urbana-Champaign

ORGANIZATION OF THE WORKSHOP

GENERAL CHAIRMAN:

*Ryszard S. Michalski
Department of Computer Science
University of Illinois
1304 W. Springfield Avenue
Urbana, Illinois 61801*

CO-CHAIRMEN:

*Jaime G. Carbonell
Carnegie-Mellon University*

*Tom M. Mitchell
Rutgers University*

MEMBERS OF THE LOCAL ORGANIZING COMMITTEE:

*A. B. Baskin
Gerald DeJong
University of Illinois
at Urbana-Champaign*

ADMINISTRATIVE MANAGER:

Kaye J. Liddle

THE PROCEEDINGS WERE PREPARED BY THE DEPARTMENT OF COMPUTER SCIENCE,
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN.

EDITOR: *Ryszard S. Michalski*

TO OBTAIN ADDITIONAL COPIES OF THE PROCEEDINGS, WRITE TO:

*Jane Wingler
University of Illinois
Department of Computer Science
1304 W. Springfield Avenue
Urbana, Illinois 61801*

FOREWORD

This collection contains papers presented at the International Machine Learning Workshop held June 22-24, 1983 at Allerton House, the residential conference center of the University of Illinois. This was the second such Workshop. The first was held at Carnegie-Mellon University in Pittsburgh, July 16-18, 1980. Machine Learning Workshops serve as a forum for exchange of ideas and for presenting current research in machine learning from an artificial intelligence perspective.

We are grateful to the Office of Naval Research for sponsoring both Workshops, and to the Department of Computer Science of the University of Illinois for providing organizational help in this Workshop. Special thanks go to James N. Snyder, Richard Canaday, June Wingler and Donna Hart for their invaluable help and support. We are also indebted to Research Assistants of the Intelligent Systems Group who assisted in innumerable ways.

Ryszard S. Michalski
Jaime G. Carbonell
Tom M. Mitchell

TABLE OF CONTENTS

AUTHOR INDEX.....	1
LEARNING BY ANALOGY	
Learning by Augmenting Rules and Accumulating Censors <i>Patrick H. Winston, Massachusetts Institute of Technology.....</i>	2
Derivational Analogy in Problem Solving and Knowledge Acquisition <i>Jaime G. Carbonell, Carnegie-Mellon University.....</i>	12
Concept Formation by Incremental Analogical Reasoning and Debugging <i>Mark H. Burstein, Yale University.....</i>	19
Programming by Analogy <i>Nachum Dershowitz, University of Illinois.....</i>	26
Reasoning by Analogy in Scientific Theory Construction <i>Lindley Darden, University of Maryland.....</i>	32
LEARNING CONCEPTS AND RULES FROM EXAMPLES	
Discovering Patterns in Sequences of Objects <i>Thomas G. Dietterich, Stanford University; and Ryszard S. Michalski, University of Illinois.....</i>	41
Learning from Noisy Data <i>J. R. Quinlan, New South Wales Institute of Technology.....</i>	58
Inductive Rule Generation in the Context of the Fifth Generation <i>Donald Michie, University of Edinburgh.....</i>	65
Knowledge Acquisition and Learning in EXPERT <i>Casimir A. Kulikowski, Rutgers University.....</i>	71
Hierarchical Memories: An Aid to Concept Learning <i>Claude Sammut and Ranan Banerji, St. Joseph's University.....</i>	74
Learning as a Non-Deterministic But Exact Logical Process <i>Yves Kodratoff and J.-G. Ganascia, Universite de Paris-Sud.....</i>	81
Escaping Brittleness <i>John H. Holland, The University of Michigan.....</i>	92
Two Programs for Testing Hypotheses of Any Logical Form <i>Clark Glymour, Kevin Kelly and Richard Scheines, University of Pittsburgh.....</i>	96

LEARNING STRATEGIES FOR PROBLEM SOLVING

Learning Equation Solving Methods from Worked Examples <i>Bernard Silver, University of Edinburgh</i>	99
Adjusting Bias in Concept Learning <i>Paul E. Utgoff, Rutgers University</i>	105
Operationalizing Advice: A Problem-Solving Model <i>Jack Mostow, USC Information Sciences Institute</i>	110
Goal Directed Learning <i>Tom M. Mitchell and Richard M. Keller, Rutgers University</i>	117
Program Synthesis as a Theory Formation Task: Problem Representations and Problem Methods <i>Saul Amarel, SRI International</i>	119

LEARNING FROM OBSERVATION AND DISCOVERY

Mechanisms for Qualitative and Quantitative Discovery <i>Pat Langley, Jan Zytkow, Herbert A. Simon and Gary L. Bradshaw, Carnegie-Mellon University</i>	121
Cognitive Economy in a Fluid Task Environment <i>Douglas B. Lenat, Stanford University; Frederick Hayes-Roth, Teknowledge, Inc.; and Philip Klahr, The Rand Corp</i>	133
The Role of Experimentation in Theory Formation <i>Thomas G. Dietterich and Bruce G. Buchanan, Stanford University</i>	147
How to Structure Structured Objects <i>Ryszard S. Michalski and Robert E. Stepp, University of Illinois</i>	156
The Architecture of Jumbo <i>Douglas R. Hofstadter, Indiana University</i>	161
An Approach to Learning From Observation <i>Gerald DeJong, University of Illinois</i>	171
Concept Learning in a Rich Input Domain <i>Michael Lebowitz, Columbia University</i>	177

COGNITIVE MODELLING AND LANGUAGE LEARNING

The Chunking of Goal Hierarchies: A Generalized Model of Practice <i>Paul S. Rosenbloom and Allen Newell, Carnegie-Mellon University</i>	183
Learning Physical Domains: Towards a Theoretical Framework <i>Kenneth D. Forbus and Dedre Gentner, Bolt, Beranek and Newman, Inc.</i>	198

Knowledge Compilation: The General Learning Mechanism <i>John R. Anderson, Carnegie-Mellon University</i>	203
Linear Separability and Concept Naturalness <i>Douglas L. Medin, University of Illinois</i>	213
How Can CHILD Learn About Agreement? Explorations of CHILD's Syntactic Inadequacies <i>Mallory Selfridge, University of Connecticut</i>	218
Inferring (MAL) Rules from Pupil's Protocols <i>Derek Sleeman, Stanford University</i>	221
Domain-specific Learning and the Subset Principle <i>Robert C. Berwick, Massachusetts Institute of Technology</i>	228
Validating a Theory of Human Skill Acquisition <i>Kurt VanLehn, Xerox Parc</i>	234

AUTHOR INDEX

	Page
Saul Amarel.....	119
John R. Anderson.....	203
Ranan Banerji.....	74
Robert C. Berwick.....	228
Gary L. Bradshaw.....	121
Bruce G. Buchanan.....	147
Mark H. Burstein.....	19
Jaime G. Carbonell.....	12
Lindley Darden.....	32
Gerald DeJong.....	171
Nachum Dershowitz.....	26
J.-G. Ganascia.....	81
Thomas G. Dietterich.....	41, 147
Kenneth D. Forbus.....	198
Dedre Gentner.....	198
Clark Glymour.....	96
Frederick Hayes-Roth.....	133
Douglas R. Hofstadter.....	161
John H. Holland.....	92
Richard M. Keller.....	117
Kevin Kelly.....	96
Philip Klahr.....	133
Yves Kodratoff.....	81
Casimir A. Kulikowski.....	71
Pat Langley.....	121
Michael Lebowitz.....	177
Douglas B. Lenat.....	133
Douglas L. Medin.....	213
Ryszard S. Michalski.....	41, 156
Donald Michie.....	65
Tom M. Mitchell.....	117
Jack Mostow.....	110
Allen Newell.....	183
J. R. Quinlan.....	58
Paul S. Rosenbloom.....	183
Claude Sammut.....	74
Richard Scheines.....	96
Mallory Selfridge.....	218
Bernard Silver.....	99
Herbert A. Simon.....	121
Derek Sleeman.....	221
Robert E. Stepp.....	156
Paul E. Utgoff.....	105
Kurt VanLehn.....	234
Patrick H. Winston.....	2
Jan Zytkow.....	121

LEARNING BY AUGMENTING RULES AND ACCUMULATING CENSORS

Patrick H. Winston
 Artificial Intelligence Laboratory
 Massachusetts Institute of Technology
 545 Technology Square
 Cambridge, MA 02139

ABSTRACT

This paper is a synthesis of several sets of ideas: ideas about learning from precedents and exercises, ideas about learning using near misses, ideas about generalizing if-then rules, and ideas about using sensors to prevent procedure misapplication.

The extensions are as follows: (1) If-then rules are augmented by *unless* conditions, creating *augmented if-then rules*. An augmented if-then rule is blocked whenever facts in hand directly demonstrate the truth of an unless condition. When an augmented if-then rule is used to demonstrate the truth of an unless condition, the rule is called a *censor*. Like ordinary augmented if-then rules, sensors can be learned. (2) Definition rules are introduced that facilitate graceful refinement. The definition rules are also augmented if-then rules. They work by virtue of *unless* entries that capture certain nuances of meaning different from those expressible by necessary conditions. Like ordinary augmented if-then rules, definition rules can be learned.

Key terms: Learning, analogy, if-then rules, sensors.

KEY IDEAS

This paper builds primarily on a previous paper that introduced a theory of learning from precedents and exercises using *constraint transfer* [Winston 1981]. The theory addresses the analogy process at work when we exploit past experience in fields like Management, Political Science, Economics, Medicine, and Law, as well as from everyday life.

Two extensions to the theory are described. Work on the first extension was stimulated by some of the apparent blunders of the extant system. Work on the second extension was stimulated by some problems encountered in making definitions.

After a brief review of the overall theory, I present an example showing that the rules generated by the

unextended learning system can be misapplied. Next, I discuss various solutions to the misapplication problem, including the introduction of sensors. At this point *augmented if-then rules* are discussed. Each augmented if-then rule contains not only *if* and *then* parts, but also an *unless* part. Before a rule acts, sensors determine if any existing facts directly demonstrate that an *unless* relation is true. If so, the rule is *blocked*.

This leads to the development of definition rules based on augmented if-then rules and a discussion of their relevance to the problem of concise definition versus unlimited nuance.

Next, it is shown that sensors can block sensors and that sensors can be learned, both by precedent and exercise and by near miss.

Finally, I describe precedents for this work itself, including ideas that stimulated what I have done, such as Minsky's ideas on the role of sensors in problem solving [Minsky 1980], as well as other ideas that I reinvented or borrowed from as my work progressed, such as Goldstein and Grimson's ideas on generalizing if-then rules [1977].

There are references throughout to an implemented system that actually does acquire and use sensors. This implemented system inherits some key ingredients from previous work:

- o **Analogy-based reasoning using *constraint transfer*.** Analogy requires the ability to determine how two situations that are similar in some respects may be similar in other respects as well. Here the determination is done by transferring constraints from the precedent situation to the exercise situation.
- o **Learned if-then rules.** In contrast to current practice in Knowledge Engineering, if-then rules emerge automatically as problems are solved. Teachers supply precedents and exercises, leaving the work of formulating the if-then rules to the system.

- o **Actor-object representation.** Situations are represented using relations between pairs of situation parts. Supplementary descriptions can be attached to the relations when elaboration is needed.
- o **Importance-dominated matching.** The similarity between two situations is measured by finding the best possible match according to what is important in the situations. Importance is determined by causal connections in the situations themselves. Causal connection is viewed as a common importance-determining constraint.

LEARNING FROM PRECEDENTS

Let us begin by reviewing the sort of task performed by the theory as previously reported. Consider the following precis of *Macbeth*, given by a teacher as a precedent:

MA is a story about Macbeth, Lady-macbeth, Duncan, and Macduff. Macbeth is an evil noble. Lady-macbeth is a greedy, ambitious woman. Duncan is a king. Macduff is a noble.

Lady-macbeth persuades Macbeth to want to be king because she is greedy. She is able to influence him because he is married to her and because he is weak. Macbeth murders Duncan with a knife. Macbeth murders Duncan because Macbeth wants to be king and because Macbeth is evil. Lady-macbeth kills herself. Macduff is angry. Macduff kills Macbeth because Macbeth murdered Duncan and because Macduff is loyal to Duncan.

Next, consider the following exercise:

Let E be an exercise. E is a story about a weak noble and a greedy lady. The lady is married to the noble. In E show that the noble may want to be king.

Told by a teacher that *Macbeth* is to be considered a precedent, it is announced that the precedent suggests that the noble may want to be king. Then a principle-capturing if-then rule is created suggesting that the weakness of a noble and the greed of his wife can cause the noble to want to be king. The rule looks

like this:

```
Rule
  RULE-1
  if
    [LADY-4 HQ GREEDY]
    [NOBLE-4 HQ WEAK]
    [[NOBLE-4 HQ MARRIED] TO LADY-4]
  then
    [NOBLE-4 WANT [NOBLE-4 AKO KING]]
  case
    MA
```

The exercise problem could have been handled by this rule directly, without recourse to the *Macbeth* precedent, were it available when the problem was posed. Thus the rule adds power. Unfortunately, it also adds blunder, as when the following exercise is given:

Let E be an exercise. E is a story about a weak noble and a greedy lady. The lady is married to the noble. He does not like her. In E show that the noble may want to be king.

This situation is different because we know that it is difficult for a person to influence someone who does not like him. Evidently, the rule is overly general, ready to reach conclusions when it should not.

This paper introduces extensions to the existing theory such that the implemented system behaves correctly on the given example and many others. The improved system works because of the following:

- o The *blocking principle*: Suppose a rule, derived from a precedent, seems to apply to a problem. Consider the relations in that part of the precedents's causal structure involved in forming the rule. If any such relation corresponds to a relation that is either false or manifestly implausible in the problem situation, then the rule based on the precedent does not apply.
- o The *prima facie conjecture*: A relation is manifestly implausible if its negation can be shown by a direct, one-step inference from relations already in place.

Thus the improved system works, satisfying one important criterion for success, and it works because it exploits identifiable ideas, satisfying another.

CREATING RULES USING ANALOGY

Let us review how rules are generated. Consider the *Macbeth* precedent, given earlier, together with the

exercise, both expressed in semantic-network form, as shown in figure 1. When asked to demonstrate that the man may want to be king, given the *Macbeth* precedent, the system proceeds as follows:

- o The people in the precedent are matched with the people in the exercise. More generally, precedent parts are matched with exercise parts.
- o The causal structure of the precedent is mapped onto the exercise.
- o It is determined that the mapped causal structure ties the relation to be shown to relations known to be true.
- o A rule is constructed, with generalizations of the exercise relations used becoming *if* parts and a generalization of the relation to be shown becoming the *then* part.

When a single precedent cannot supply the total causal structure needed, the system attempts to chain several together. In the example, if it were not known already that the woman is greedy, as required for application of the *Macbeth* precedent, greed might be established through another precedent or already-learned rule. A previous paper explains this in detail [Winston 1981].

ENABLING CENSORS

So far we have established that rules can be generated and that they need to be blocked in certain circumstances. There are three obvious ways to arrange for blocking:

First, expand the *if* part of an offending rule, restricting its use. One problem with this idea is that rules can become bloated with endless tests for increasingly unlikely minutiae. Such bloat makes rules obscure and hard to criticize, debug, and improve, for both us people and for reasoning programs.

Second, attach censors to each rule. Have the censors check the problem to be solved for contraindications to the rules the censors are attached to. One problem is that the rules can become bloated with censor names; these censor names would give no explicit insight into when the rules do not apply.

Third, have censors watch for particular relations. Forbid any rule or precedent to work toward establishing a relation that a censor objects to. One problem is that the rules continue to look silly, containing no hint about when they do not apply.

Censors can Block Augmented If-then Rules

A better, less obvious idea, is this:

- o Augment each rule at the time it is generated with entries that correspond to all relations in the causal structure lying between relations that enter the *if* part of the rule and the relation that enters the *then* part of the rule. Negations of these intermediate entries constitute the *unless* part of the rule. According to the blocking principle, if any entry in the *unless* part of the rule corresponds to something that is manifestly true, then the rule does not apply.

Clearly a relation is manifestly true if the existing facts indicate that the relation is true. But introspectively, it seems unreasonable to go deeply into reasoning about *unless* entries. Hence the implemented system adheres to the following specialization of the *prima facie* conjecture:

- o If any entry in the *unless* part of a rule corresponds to a relation that can be shown to be true by another rule working directly from relations already in place, then block the rule.

Suppose, for example, that a rule's *unless* part is triggered when someone is unable to influence another. Such a rule will be blocked if the person to be influenced does not like the other. The augmented form of RULE-1 is:

```

Rule
  RULE-1
  if
    [LADY-4 HQ GREEDY]
    [NOBLE-4 HQ WEAK]
    [[NOBLE-4 HQ MARRIED] TO LADY-4]
  then
    [NOBLE-4 WANT [NOBLE-4 AKO KING]]
  unless
    [[LADY-4
      PERSUADE
      [NOBLE-4 WANT [NOBLE-4 AKO KING]]]
      HQ FALSE]
    [[LADY-4 HQ ABLE
      TO
      [LADY-4 INFLUENCE NOBLE-4]]
      HQ FALSE]
```

The blocking rule is:

```

Rule
  RULE-2
if
  [[PERSON-8 LIKE PERSON-7] HQ FALSE]
then
  [[[PERSON-7 HQ ABLE]
   TO
   [PERSON-7 INFLUENCE PERSON-8]]
   HQ FALSE]

```

A rule becomes a *censor* when it blocks the application of another rule. Since censors look just like any other rules, censors can be learned, stored, and retrieved in the same ways.

Note that when the illustrated rule is used to block another, it only works if it is known at the time of use that there is dislike. There is no attempt to demonstrate dislike when not already known.

Note that the viability of the *prima facie* conjecture depends on having a rich vocabulary of relations. It would be difficult to demonstrate anything in one step if all relations were reduced to canonical constellations of small-vocabulary primitives. This opens the question of just how rich the vocabulary should be, a question answered operationally by using freely those relations for which there are common natural-language words.

The viability of the *prima facie* conjecture also depends on having all solid facts available before backward-chaining problem solving begins. This means that all solid facts are either given facts or deduced already by forward chaining from given facts using reliable, potentially relevant rules. Reliability is insured by forward chaining only with rules that reach unassailable conclusions. Relevance cannot be insured, but can be rendered more likely. One way is to use the context mechanism described in an earlier paper [Winston 1981].

Censors can Block Censors

Actually, it is possible to be influenced by someone you dislike if for some reason you trust them in spite of the dislike. Perhaps the real able-to-influence censor should look like this:

```

Rule
  RULE-2
if
  [[PERSON-8 LIKE PERSON-7] HQ FALSE]
then
  [[[PERSON-7 HQ ABLE]
   TO
   [PERSON-7 INFLUENCE PERSON-8]]
   HQ FALSE]
unless
  [PERSON-8 TRUST PERSON-7]

```

Such a censor could be blocked by another censor which states that you believe someone if they have the ability to convince you:

```

Rule
  CENSOR-1
if
  [[PERSON-6 HQ ABLE]
   TO
   [PERSON-6 CONVINCE PERSON-5]]
then
  [PERSON-5 TRUST PERSON-6]

```

To illustrate how these can interact, consider the following situation:

Let E be an exercise. E is a story about a weak noble and a greedy lady. The lady is married to the noble. He does not like her. The lady is able to convince the noble. In E show that the noble may want to be king.

This produces the following scenario:

- o First, the problem is posed and RULE-1 is fetched. Its *if* parts are satisfied.
- o Next, the *unless* part of RULE-1 is examined. The line involving ability to influence causes RULE-2 to be fetched. Its *if* parts are satisfied. RULE-1 is about to be blocked.
- o But RULE-2's *unless* part must be examined. The line involving believing causes CENSOR-1 to be fetched. Its *if* parts are satisfied. Thus CENSOR-1 blocks RULE-2, preventing RULE-2 from blocking RULE-1.
- o Finally, RULE-1 succeeds, establishing the relation originally asked about.

Augmented If-then Rules are not Rules of Inference

It is tempting to write censors in the following way:

$$A_1 \wedge \dots \wedge A_n \wedge \neg(B_1 \vee \dots \vee B_n) \Rightarrow C,$$

or alternatively,

$$A_1 \wedge \dots \wedge A_n \wedge \neg B_1 \wedge \dots \wedge \neg B_n \Rightarrow C,$$

where the *As* are in the *if* part of the rule and the *Bs* are in the *unless* part.

Logical notation is deceptive, however, for in the use of augmented if-then rules, the *As* and *Bs* get treated differently from each other, in contrast to the conventions of traditional logic: unlimited effort is to be put into showing the *As* are true; only one-step effort is put into showing that the *Bs* are true, with the *Bs* assumed false on failure.

Note that rules used as censors are not permitted to create new objects. This insures that the amount of computation added by the application of censors to *unless* entries is bounded even though censors have their own *unless* parts that must be checked by censors. I believe it is likely that censor computations will prove in practice to be broad and shallow, as well as bounded, suggesting parallel implementation.

Augmented Rules suggest an Approach to certain Definition Problems

Winograd has discussed the difficulty of definition using the word *bachelor* [Winograd 1976]. To be sure, a bachelor is an unmarried adult man, but Winograd notes that such a definition can cause trouble if used when someone says, "Please invite some nice bachelors to my party," for it would be strange to invite certain kinds of bachelors. For example, Catholic priests and misogynists, while satisfying the dictionary definition, are clearly not what a party giver has in mind.

Since the exception possibilities seem limitless, Winograd feels it is inappropriate to rest a definition of *bachelor* on a clearly defined, small set of primitive propositions, arguing that it is better to think of using some abstract measure of closeness to an extensible set of exemplars. Woods takes issue with Winograd's view, feeling that correct understanding must involve an explicit selection of a particular word sense, rather than closeness to a generally applicable exemplar set [Woods 1981].

The augmented-rule idea may offer a slightly different approach to the problem. Consider the following definition of *bachelor*, stated as an augmented if-then rule:

```

Rule
  RULE-2
  if
    [MAN-10 AKO MAN]
    [[MAN-10 HQ MARRIED] HQ FALSE]
    [MAN-10 AKO ADULT]
  then
    [MAN-10 AKO BACHELOR]
  unless
    [[MAN-10 HQ MARRIED] HQ EXPECTED]
    HQ FALSE]
    [[MAN-10 HQ ABLE] TO [MAN-10 HQ MARRIED]]
    HQ FALSE]

```

With this definition, the conclusion can be avoided, even though the *if* part of the rule is fully satisfied, providing that the individual involved is not able to be married or is not expected to be married. This takes care of the priest and the misogynist problems, given the following censors:

```

Rule
  CENSOR-1
  if
    [PERSON-1 AKO MISOGAMIST]
  then
    [[[PERSON-1 HQ MARRIED] HQ EXPECTED]
    HQ FALSE]

Rule
  CENSOR-2
  if
    [MAN-4 AKO PRIEST]
  then
    [[[MAN-4 HQ ABLE] TO [MAN-4 HQ MARRIED]]
    HQ FALSE]

```

Evidently, it is possible to have a simple, stable definition of bachelor, while at the same time allowing for knowledge relevant to bachelors to interact with the definition, when appropriate, as that knowledge is accumulated. As more is learned, the definition is used more intelligently, and, in a sense, the definition is never closed.

How does capturing the meaning of *bachelor* with an augmented if-then rule compare with other approaches? One point of view is that Winograd's exemplars correspond to rule-generating precedents, and learned augmented if-then rules correspond to Woods's selectable word senses. We will turn to learning about bachelors from precedents in a moment.

Censors can Improve Precedent Reasoning

While censors were originally investigated in this work in order to cure the apparent silliness of some learned rules, they help in another context too. When ordinary precedent-exercise problem solving is in progress, the analogy part of the system works back through the causal structure in the precedent, looking for relations that correspond to relations in the exercise. Each time there is no corresponding relation, before the system moves further through the causal structure, it does a censor check.

- o If a relation is encountered in the causal structure of the precedent that corresponds to a relation that is manifestly improbable in the exercise, then the precedent cannot support a conclusion.

LEARNING AUGMENTED RULES

Since censor rules and definition rules are just rules used in a special way, they can be learned just like any other rules. This may be by direct telling, or it may be by precedent and exercise, or it may be by near-miss.

Augmented Rules can be Learned by Precedent and Exercise

Here is a precedent and an exercise for learning the bachelor definition rule:

Let S be a story. S is a story about Casanova. Casanova is a bachelor because he is a man and because he is expected to be married. He is expected to be married because he is able to be married. He is able to be married because he is an adult and because he is not married.

Let E be an exercise. E is a story about Henry. He is a man and an adult. He is not married. In E show that Henry is a bachelor.

Of course, one might argue that providing the precedent involving Casanova is unrealistic spoon feeding. Indeed, it may well be, so it is important to understand that the same bachelor rule can be learned using several independent precedents:

Let S be a story. S is a story about a man. He is a bachelor because he is expected to be married. He is a bachelor because he is a man.

Let S be a story. S is a story about a man. He is expected to be married because he is able to be married.

Let S be a story. S is a story about a man. He is able to be married because he is an adult and because he is not married.

Alternatively, the bachelor rule can be learned using several previously-learned rules:

```
Rule
  STORY-1
  if
    [MAN-1 AKO MAN]
    [[MAN-1 HQ MARRIED] HQ EXPECTED]
  then
    [MAN-1 AKO BACHELOR]
```

```
Rule
  STORY-2
  if
    [[MAN-2 HQ ABLE] TO [MAN-2 HQ MARRIED]]
  then
    [[MAN-2 HQ MARRIED] HQ EXPECTED]
```

```
Rule
  STORY-3
  if
    [MAN-3 AKO ADULT]
    [[MAN-3 HQ MARRIED] HQ FALSE]
  then
    [[MAN-3 HQ ABLE] TO [MAN-3 HQ MARRIED]]
```

Also, it is possible to learn a rule that allows a married Moslem, seeking an additional wife, to be considered a bachelor.

Augmented Rules can be Learned by Near-miss

Of course, there should be some way of recovering if an impoverished definition is acquired early on. The near-miss idea seems useful in such situations. Consider this scenario:

- o A teacher tells the system that a bachelor is an unmarried, adult man. This produces an impoverished definition of bachelor, one without anything in the *unless* part.
- o The teacher complains when the system identifies a Catholic priest as a bachelor.
- o The system notices that the only robust difference between the priest and other people who are correctly identified as bachelors is that the priest is not able to be married.
- o The system guesses that bachelors must be able to be married and puts an appropriate entry in the *unless* part of the bachelor definition.

Of course, this is a particularly simple situation since there is but one object involved and the descriptions are such that the near-miss-causing relation is the only relation that is caused by something and not deemed plausible in a situation where the rule does apply. It is not known how difficult it would be to identify the right difference in general. Recent work by Berwick on syntax acquisition [1982] and by Minsky in concept

learning [unpublished draft] suggest that if it is difficult to identify the right difference, a learning system should simply give up, waiting for more transparent examples to come along.

THE IMPLEMENTED SYSTEM

The example precedents, exercises, rules, and censors in this paper are shown in the exact English form used by the implemented system. Translation from English into the semantic net representation used by the system is done by a parser developed and implemented by Boris Katz [Katz 1980, Katz and Winston 1982]. The grammar used by the parser is also used by a generator, which produces English versions of the rules. For example, the generator converts

```

Rule
  RULE-2
if
  [MAN-10 AKO MAN]
  [[MAN-10 HQ MARRIED] HQ FALSE]
  [MAN-10 AKO ADULT]
then
  [MAN-10 AKO BACHELOR]
unless
  [[MAN-10 HQ MARRIED] HQ EXPECTED]
  HQ FALSE]
  [[MAN-10 HQ ABLE] TO [MAN-10 HQ MARRIED]]
  HQ FALSE]
into

```

Rule-2 concerns a man. If the man is not married and he is an adult, then he is a bachelor, unless he is not expected to be married or he is not able to be married.

So far, the system knows a few dozen censors, most of which it is told, all of which it can learn from precedents or rules and exercises. Clearly the number is enough to do surface-scratching experiments and to illustrate the ideas, but an order of magnitude or two more will be required to demonstrate the ideas.

OPEN QUESTIONS

It is plain that this work is only a beginning. Work is in progress on several related fronts:

- o In collaboration with Boris Katz: the problem of retrieving precedents from a data base so that they need not be given by a teacher.
- o In collaboration with Tomas O. Binford (Stanford University), Michael Lowry (Stanford University),

and Boris Katz: the problem of creating appearance descriptions from functional descriptions, precedents, and examples.

- o In response to a suggestion by J. Michael Brady: an augmentation of the rules with an *if-relevant* part in addition to the *unless* part described in this paper. The idea is that the *if-relevant* part will somehow keep track of the ultimate goals a rule may be relevant to, so that the rule is used in forward chaining only if one of the potential ultimate goals is involved in the problem to be solved. This would make the rules look like this in logical notation:

$$\begin{aligned}
 &A_1 \wedge \dots \wedge A_n \\
 &\wedge \neg(B_1 \vee \dots \vee B_n) \\
 &\wedge (G_1 \vee \dots \vee G_n) \\
 &\Rightarrow C,
 \end{aligned}$$

where the *As* are in the *if* part of the rule, the *Bs* are in the *unless* part, and the *Gs* are in the *if-relevant* part; and where it is understood that only one-step effort is to be put into the *Bs* and *Gs*.

This would complement the existing context mechanism explained previously [Winston 1981].

In addition, the following open questions, enumerated in a previous paper, remain open [Winston 1981]:

- o There is no way to handle degree of certainty of cause. Moreover, there is no way to handle subcategories of cause such as those sketched by Rieger [1978].
- o There is no way to handle constraints about quantities such as those constraints that appear in the work of Forbus [1982].
- o There is no way to summarize an episode in a story so as to make a general precis leading to more abstract rules. Lehnert's summarization work should be tried [Lehnert 1981].
- o There are no satisfying ideas about the role of abstraction in doing matching and indexing and retrieving.

- o The representation for time is impoverished. Similarly quantification, negation, disjunction, and perspective are missing.

CONCLUSION: SIMPLE IDEAS HAVE PROMISE

This paper is about a set of ideas that enable improvement in the reliability of learned rules. The extended theory enables improved performance in those domains subject to problem solving by analogy. Such domains satisfy certain restrictions:

- o The situations in the domain can be represented by the relations between the parts together with the classes and properties of those parts.
- o The importance of a part of a description is determined by the constraints it participates in.
- o Constraints that once determine something will tend to do so again.

Things that involve spatial, visual, and aural reasoning do not seem to satisfy all the restrictions. Things that involve Management, Political Science, Economics, Law, Medicine, and ordinary common sense do seem to satisfy the restrictions, however, and are targets for the learning and reasoning ideas of the theory:

- o Actor-object representation.
- o Importance-dominated matching.
- o Analogy-based reasoning using constraint transfer.
- o If-then rules learned by solving problems.
- o If-then rules improved by modifications based on near misses.
- o If-then rules augmented by unless parts.
- o Blocking censors that create fences around rules using *prima facie* evidence.

RELATED WORK

This work builds on the MACBETH system [Winston 1980, 1982], which concentrated on analogy and rule acquisition. Also, Minsky's views on censors had a major influence [Minsky 1980]. To a lesser extent, the idea of learning by near miss is involved [Winston 1970].

The augmented if-then rule is a special case of the annotated if-then rule introduced by Goldstein and Grimson in a paper on flight simulation [1977]. They had the idea that if-then rules should exhibit certain unless-like conditions (which they called caveats), as well as rationales, plans, and control information. The work of Brown and VanLehn on explaining subtraction bugs is a more recent precedent for using censors to block rules, although their censors (which they call critics) are triggered by what a rule does, rather than by unless conditions [Brown and VanLehn 1980].

The idea that censors should work only with the facts in hand is a variant on the theme of reasoning using limited resources, an idea that is discussed widely, particularly in the expert-systems literature.

John Mallery observed in conversation that the definition of *bachelor* really should say something about being expected to be married, stimulating me to try handling the bachelor problem within the *unless* framework. Boris Katz pointed out that the *prima facie* conjecture does not make sense unless all reliable, potentially relevant forward chaining is done first. Jonathan H. Connell suggested using the designator *unless*, rather than *if-plausible*, which was used in a previous version of this paper.

ACKNOWLEDGMENTS

This paper was improved by comments from Robert Berwick, J. Michael Brady, Boris Katz, Michael Lowry, and Karen A. Prendergast.

This research was done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial-intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505.

REFERENCES

- Berwick, Robert "Locality Principles and the Acquisition of Syntactic Knowledge," PhD Thesis, Department of Electrical Engineering & Computer Science, MIT, 1982.
- Brown, John Seely and Kurt VanLehn, "Repair Theory: A Generative Theory of Bugs in Procedural Skills," *Cognitive Science*, vol. 4, no. 4, October-December, 1980.
- Davis, Randall "Applications of Meta Level Knowledge to the Construction, Maintenance, and Use of Large Knowledge Bases," Ph. D. Thesis, Stanford University, Stanford, California, 1979. Also in *Knowledge-Based Systems in Artificial Intelligence*, Randall Davis and Douglas Lenat, 1980.
- Forbus, Ken, "Qualitative Reasoning about Physical Processes," submitted for publication, 1982.
- Goldstein, Ira P. and Eric Grimson, "Annotated Production Systems A Model for Skill Acquisition," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977. Available through Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 15213.
- Katz, Boris and Patrick H. Winston, "Parsing and Generating English using Commutative Transformations," Artificial Intelligence Laboratory Memo No. 677, May, 1982. Also see "A Two-way Natural Language Interface," in *Integrated Interactive Computing Systems*, edited by P. Degano and E. Sandewall, North-Holland, Amsterdam, 1982.
- Lehnert, Wendy, "Plot Units and Narrative Summarization," *Cognitive Science*, vol 4, 1981.
- Minsky, Marvin, "Jokes and the Logic of the Cognitive Unconscious," M.I.T. Artificial Intelligence Laboratory Memo No. 603, November 1980.
- Minsky, Marvin, unpublished draft on concept learning.
- Rieger, Chuck, "On Organization of Knowledge for Problem Solving and Language Comprehension," *Artificial Intelligence*, vol. 7, no. 2, 89-128, 1978.
- Winograd, Terry, "Towards a Procedural Understanding of Semantics," Stanford Artificial Intelligence Laboratory Memo AIM-292, Stanford Computer Science Department Report No. STAN-CS-76-580, November, 1976.
- Winston, Patrick Henry, "Learning Structural Descriptions from Examples," Ph.D. thesis, MIT, 1970. A shortened version is in *The Psychology of Computer Vision*, edited by Patrick Henry Winston, McGraw-Hill Book Company, New York, 1975.
- Winston, Patrick Henry, "Learning by Creating and Justifying Transfer Frames," *Artificial Intelligence*, vol. 10, no. 2, 147-172, 1978.
- Winston, Patrick Henry, "Learning and Reasoning by Analogy," *CACM*, vol. 23, no. 12, December, 1980. A version with details is available as "Learning and Reasoning by Analogy: the Details," M.I.T. Artificial Intelligence Laboratory Memo No. 520, April 1979.
- Winston, Patrick Henry, "Learning New Principles from Precedents and Exercises," to appear in *Artificial Intelligence*. A version with details is available as "Learning New Principles from Precedents and Exercises: the Details," M.I.T. Artificial Intelligence Laboratory Memo No. 632, May 1981.
- Woods, William A., "Procedural Semantics as a Theory of Meaning," Bolt Beranek and Newman Report No. 4627, March 1981. Also available in *Computational Aspects of Linguistic Structures*, A. Joshi, I. Sag, and B. Webber (eds.), Cambridge University Press.

Derivational Analogy in Problem Solving and Knowledge Acquisition

Jaime G. Carbonell
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Derivational analogy, a method of solving problems based upon the transfer of past experience to new problem situations, is discussed in the context of other general approaches to problem solving. The experience transfer process consists of recreating lines of reasoning, including decision sequences and accompanying justifications, that proved effective in solving particular problems requiring similar initial analysis. The derivational analogy approach is advocated as a method for instructing expert systems, as an alternative approach to current painstaking knowledge acquisition methods of handcrafting, testing and tuning individual, domain specific heuristic rules.

1. Introduction: The Role of Analogy in Problem Solving

The term "problem solving" in artificial intelligence has been used to denote disparate forms of intelligent action to achieve well-defined goals. Perhaps the most common usage stems from Newell and Simon's work [22] in which problem solving consists of selecting a sequence of operators (from a pre-analyzed finite set) that transforms an initial problem state into a desired goal state. Intelligent behavior consists of a focused search for a suitable operator sequence by analyzing the states resulting from the application of different operators to earlier states.¹ Many researchers have adopted this viewpoint [12, 26, 23].

However, a totally different approach has been advocated by McDermott [19] and by Wilensky [32, 33] that views problem solving as plan instantiation. For each problem posed there are one or more plans that outline a solution, and problem solving consists of identifying and instantiating these plans. In order to select, instantiate, or refine plans, additional plans that tell how to instantiate other plans or how to solve subproblems are brought to bear in a recursive manner. Traditional notions of search are totally absent from this formulation. Some systems, such as the counterplanning mechanism in POLITICS [6, 3], provide a hybrid approach, instantiating plans whenever possible, and searching to construct potential solutions in the absence of applicable plans.

A third approach is to solve a new problem by analogy to a previously solved similar problem. This process entails

searching for related past problems and transforming their solutions into ones potentially applicable to the new problem [24]. I developed and advocated such a method [7, 8] primarily as a means of bringing to bear problem solving expertise acquired from past experience. The analogical transformation process itself may require search, as it is seldom immediately clear how a solution to a similar problem can be adapted to a new situation.

A useful means of classifying different problem solving methods is to compare them in terms of the amount and specificity of domain knowledge they require.

- If no structuring domain knowledge is available and there is no useful past experience to draw upon; weak methods such as heuristic search and means-ends analysis are the only tools that can be brought to bear. Even in these knowledge-poor situations, information about goal states, possible actions, their known preconditions and their expected outcomes is required.
- If specific domain knowledge in the form of plans or procedures exists, such plans may be instantiated directly, recursively solving any subproblems that arise in the process.
- If general plans apply, but no specific ones do so, the general plans can be used to reduce the problem (by partitioning the problem or providing islands in the search space). For instance, in computing the pressure at a particular point in a fluid statics problem, one may use the general plan of applying the principle of equilibrium of forces on the point of interest (the vector sum of the forces = 0). But, the application of this plan only reduces the original problem to one of finding and combining the appropriate forces, without hinting how that may be accomplished in a specific problem [5].
- If no specific plans apply, but the problem resembles one solved previously, apply analogical transformation to adapt the solution of that similar past problem to the new situation. For instance, in some studies it has proven easier for students to solve mechanics problems by analogy to simpler solved problems than by appealing to first principles or by applying general procedures presented in a physics text [9]. As an example of analogy involving composite skills rather than pure cognition, consider a person who knows how to drive a car

¹In means-ends analysis, the current state is compared to the goal state and one or more operators that reduce the difference are selected, whereas in heuristic search, the present state is evaluated in isolation and compared to alternate states resulting from the application of different operators to states generated earlier in the search, and the search for a solution continues from the highest-rated state.

and is asked to drive a truck. Such a person may have no general plan or procedure for driving trucks, but is likely to perform most of the steps correctly by transferring much of his or her automobile driving knowledge. Would that we had robots that were so self-adaptable to new, if recognizably related tasks!

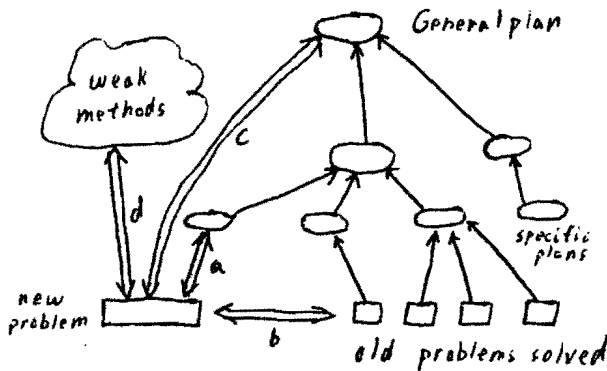


Figure 1-1: Problem solving may occur by a) instantiating specific plans, b) analogical transformation to a known solution of a similar problem, c) applying general plans to reduce the problem, d) applying weak methods to search heuristically for a possible solution, or e) a combination of these approaches.

Clearly, these problem solving approaches are not mutually exclusive; for instance, one approach can be used to reduce a problem to simpler subproblems, which can in turn be solved by the other methods. In fact, Larkin and I [5] are developing a general inference engine for problem solving in the natural sciences that combines all four approaches.

As discussed earlier, only direct plan instantiation and weak methods have received substantial attention by AI practitioners. For instance, Newell and Laird's recent formulation of a universal weak method [17] as a general problem solving engine is developed completely within the search paradigm. Expert systems, for the most part, combine aspects of plan instantiation (often broken into small rule-size chunks of knowledge) and heuristic search in whatever manner best exploits the explicit and implicit constraints of the specific domain [30, 11, 20]. I am more concerned with the other two approaches, as the they could conceivably provide powerful reasoning mechanisms not heretofore analyzed in the context of automating problem-solving processes. The rest of this paper focuses on a new formulation of the analogical problem solving approach.

2. Analogy and Experiential Reasoning

The term *analogy* often conjures up recollections of artificially contrived problems asking: "X is to Y as Z is to ?" in various psychometric exams. This aspect of analogy is far too narrow and independent of context to be useful in general problem solving domains. Rather, I propose the following operational definition of analogical problem solving consistent with past AI research efforts [16, 34, 35, 13, 4, 8].

Definition: *Analogical problem solving consists of transferring knowledge from past problem solving*

episodes to new problems that share significant aspects in common with corresponding past experience.

In order to make this definition operational, the problem solving method must specify:

- what it means for problems to "share significant aspects in common",
- what knowledge is transferred from past experience to the new situation,
- precisely how the knowledge transfer process occurs,
- and how analogically related experiences are selected from a potentially vast long term memory of past problem solving episodes.

The remainder of this paper discusses two major approaches to analogical problem solving I have analyzed in terms of these four criteria. The first approach has been successfully implemented in ARIES (Analogical Reasoning and Inductive Experimentation System), and we are actively experimenting with the other approach. This short paper focuses on a comparative analysis of the two methods, rather than discussing implementation techniques or examining our preliminary empirical results.

2.1. Analogical Transformation of Past Solutions

If a particular solution has been found to work on a problem similar to the one at hand, perhaps it can be used, with minor modification, for the present problem. By "solution" I mean only a sequence of actions that if applied to the initial state of a problem brings about its goal state. Simple though this process may appear, an effective computer implementation requires that many difficult issues be faced, to wit:

1. Past problems descriptions and their solutions must be remembered and indexed for later retrieval.
2. The new problem must be matched against large numbers of potentially relevant past problems to find closely related ones, if any. An operational similarity metric is required as a basis for selecting the most suitable past experiences.
3. The solution to a selected old problem must be transformed to satisfy the requirements of the new problem statement.

In order to achieve these objectives, the initial analogical problem solver [8] required a partial matcher with a built-in similarity criterion, a set of possible transformations to map the solution of one problem into the solution to a closely related problem, and a memory indexing mechanism based on a MOPS-like memory encoding of events and actions [28]. The solution transformation process was implemented as a set of primitive transform operators and a means-ends problem solver that searched for sequences of primitive transformations yielding a solution to the desired problem. The resultant system, called ARIES-I, turned out to be far more complex than originally envisioned. Partial pattern matching of problem descriptions and searching in the space of solution transformations are difficult tasks in themselves.

In terms of the four criteria, the solution transformation

process may be classified as follows:

1. Two problems share significant aspects if they match within a certain preset threshold in the initial partial matching process.
2. The knowledge transferred to the new situation is the sequence of actions from the retrieved solution, whether or not that sequence is later modified in the analogical mapping process.
3. The knowledge transfer process is accomplished by copying the retrieved solution and perturbing it incrementally according to the primitive transformation steps in a heuristically guided manner until it satisfies the requirements of the new problem. (See [8] for details.)
4. The selection of relevant past problems is constrained by the memory indexing scheme and the partial pattern matcher.

Since a significant fraction of problems encountered in mundane situations and in areas requiring significant domain expertise (but not in abstract mathematical puzzles) bear close resemblance to past solved problems, the ARIES-I method proved effective when tested in various domains, including algebra problems and route planning tasks. An experiential learning component was added to ARIES that constructed simple plans (generalized sequences of actions) for recurring classes of problems, hence allowing the system to solve new problems of this class by the more direct plan instantiation approach. However, no sooner was the solution transformation method implemented and analyzed than some of its shortcomings became strikingly apparent. In response to these deficiencies, I started analyzing more sophisticated methods of drawing analogies, as discussed in the following sections.

3. The Derivational Analogy Method

In formulating plans and solving problems, a considerable amount of intermediate information is produced in addition to the resultant plan or specific solution. For instance, formulation of subgoal structures, generation and subsequent rejection of alternatives, and access to various knowledge structures all typically take place in the problem solving process. But, the solution transformation method outlined above ignores all such information, focusing only upon the resultant sequence of actions and disregarding, among other things, the reasons for those actions. Why should one take such extra information into account? It would certainly complicate the analogical problem solving process, but what benefits would accrue from such an endeavor? Perhaps the best way to answer this question is by analysis of where the simple solution transformation process falls short and how such problems may be alleviated or circumvented by preserving more information from which qualitatively different analogies may be drawn.

3.1. The Need for Preserving Derivation Histories

Consider, for instance, the domain of constructing computer programs to meet a set of pre-defined specifications. In the automatic programming literature, perhaps the most widely used technique developed thus far is one of progressive refinement [2, 1, 15]. In brief, progressive refinement is a multi-stage process that starts from abstract specifications stated in a high

level language (typically English or some variant of first order logic), and produces progressively more operational or algorithmic descriptions of the specifications committing to control decisions, data structures and eventually specific statements in the target computer language. However, humans (well, at least this writer) seldom follow such a long painstaking process, unless perhaps the specifications call for a truly novel program unlike anything in one's past experience. Instead, a common practice is to recall similar past programs and reconstruct the new programming problem along the same directions. For instance, one should be able to program a quicksort algorithm in LISP quite easily if one has recently implemented quicksort in PASCAL. Similarly, writing LISP programs that perform tasks centered around depth-first tree traversal (such as testing equality of S-expressions or finding the node with maximal value) are rather trivial for LISP programmers but surprisingly difficult for those who lack the appropriate experience.

The solution transformation process proves singularly inappropriate as a means of exploiting past experience in such problems. A PASCAL implementation of quicksort may look very different than a good LISP implementation. In fact, attempting to transfer corresponding steps from the PASCAL program into LISP is clearly not a good way to produce any LISP program, let alone an elegant or efficient one. Although the two problem statements may have been similar, and the problem solving processes may preserve much of the inherent similarity, the resultant solutions (i.e., the PASCAL and LISP programs) may bear little if any direct similarities.

The useful similarities lie in the algorithms implemented and in the set of decisions and internal reasoning steps required to produce the two programs. Therefore, the analogy must take place starting at earlier stages of the original PASCAL implementation, and it must be guided by a reconsideration of the key decisions in light of the new situation. In particular, the derivation of the LISP quicksort program starts from the same specifications, keeping the same divide and conquer strategy, but may diverge in selecting data structures (e.g. lists vs arrays), or in the method of choosing the comparison element, depending on the tools available in each language and their expected efficiency. However, future decisions (e.g. whether to recurse or iterate, what mnemonics to use as variable names, etc.) that do not depend on earlier divergent decisions can still be transferred to the new domain rather than recomputed. Thus, the derivational analogy method walks through the reasoning steps in the construction of the past solution and considers whether they are still appropriate in the new situation or whether they should be reconsidered in light of significant differences between the two situations.

The difference between the solution transformation approach and the derivational analogy approach just outlined can be stated in terms of the operational knowledge that can be brought to bear. The former corresponds to a person who has has never before programmed quicksort and is given the PASCAL code to help him construct the LISP implementation, whereas the latter is akin to a person who has programmed the PASCAL version himself and therefore has a better understanding of the issues involved before undertaking the LISP implementation. Swartout and Balzer [31] and Scherlis [25] have argued independently in favor of working with program derivations as the basic entities in tasks relating to automatic programming. The advantages of the derivational analogy approach are quite evident in programming because the of the frequent inappropriateness of direct solution transformation, but even in

domains whether the latter is useful, one can create problems that demonstrate the need for preserving or reconstructing past reasoning processes.

3.2. The Process of Drawing Analogies by Derivational Transformation

Let us examine in greater detail the process of drawing analogies from past reasoning processes. The essential insight is that useful experience is encoded in the reasoning process used to derive solutions to similar problems, rather than just in the resultant solution. And, a method of bringing that experience to bear in the problem solving process is required in order to make this form of analogy a computationally tractable approach. Here we outline such a method:

1. When solving a problem by whatever means store each step taken in the solution process, including:

- The subgoal structure of the problem
- Each decision made (whether a decision to take action, to explore new possibilities, or to abandon present plans), including:
 - Alternatives considered and rejected
 - The reasons for the decisions taken (with dependency links to the problem description or information derived therefrom)
 - The start of a false path taken (with the reason why this appeared to be a promising alternative, and the reason why it proved otherwise, again with dependency links to the problem description. Note that the body of the false path and other resultant information need not be preserved.)
 - Dependencies of later decisions on earlier ones in the derivation.
- Pointers to the knowledge that was accessed and proved useful in the eventual construction of the solution
- The resultant solution itself
 - In the event that the problem solver proved incapable of solving the problem, the closest approach to a solution should be stored, along with the reasons why no further progress could be made (e.g., a conjunctive subgoal that could not be satisfied).
 - In the event that the solution depends, perhaps indirectly, on volatile assumptions not stated in the problem description (such as the cooperation of another agent, or time-dependent states) store the appropriate dependencies.

2. When a new problem is encountered that does not

lend itself to direct plan instantiation or other direct recognition of the solution pattern, start to analyze the problem by applying general plans or weak methods, whichever is appropriate to the situation.

3. If after commencing the analysis of the problem, the reasoning process (the initial decisions made and the information taken into account) parallels that of past problem situations, retrieve the full reasoning traces and proceed with the derivational transformation process. If not, consider the possibility of solution transformation analogy or, failing that, proceed with the present line of non-analogical reasoning.

- Two problems are considered similar if their analysis results in equivalent reasoning processes, at least in its initial stages. This replaces the more arbitrary context-free similarity metric required for partial matching among problem descriptions in drawing analogies by direct solution transformation. Hence, past reasoning traces (henceforth *derivations*) are retrieved if their initial segment matches that the first stages of the analysis of the present problem.
- The retrieved reasoning processes are then used much as individual relevant cases in medicine are used to generate expectations and drive the diagnostic analysis. Reasoning from individual cases has been recognized as an important component of expertise [29], but little has been said of the necessary information that each case must contain, let alone providing a simple method of retrieving the appropriate cases in a manner that does not rely on arbitrary similarity metrics. Here, I take the stand that cases must contain the reasoning process used to yield an answer, together with dependencies to the particular circumstances of the problem, pointers to data that proved useful, list of alternative reasoning paths not taken, and failed attempts (coupled with both reasons for their failure and reasons for having originally made the attempt). Case-based reasoning is nothing more than derivational analogy applied to domains of extensive expertise.
- It is important to know that although one may view derivational analogy as an interim step in reasoning from particular past experience as more general plans are acquired, it is a mechanism that remains forever useful, since knowledge is always incomplete and exceptions to the best formulated general plans require representation and use of individual reasoning episodes.

4. A retrieved derivation is applied to the current situation as follows. For each step in the derivation, starting immediately after the matched initial segment, check whether the reasons for performing that step are still valid by tracing dependencies in the retrieved derivation to relevant parts of the old

problem description or to volatile external assumptions made in the initial problem solving.

- If parts of the problem statement or external assumptions on which the retrieved situation rests are also true in the present problem situation, proceed to check the next step in the retrieved derivation.
- If there is a violated assumption or problem statement, check whether the decision made would still be justified by a different derivation path from the new assumptions or statements. If so, store the new dependencies and proceed to the next step in the retrieved derivation. The idea of tracing causal dependences and verifying past inference paths borrow heavily from TMS [10] and some of the non-monotonic logic literature [18]. However, the role played by data dependencies in derivational analogy is somewhat different and more constrained than in maintaining global consistency in deductive data bases.
- If the old decision cannot be justified by new problem situation,
 - evaluate the alternatives not chosen at that juncture and select the an appropriate one in the usual problem solving manner, storing it along with its justifications, or
 - initiate the subgoal of establishing the right supports in order for the old decision to apply in the new problem² (clearly, any problem solving method can be brought to bear in achieving the new subgoal), or
 - abandon this derivational analogy in favor of another more appropriate problem solving experience from which to draw the analogy or in favor of other means of problem solving.
- If one or more failure paths are associated with the current decision, check the cause of failure and the reasons these alternatives appeared viable in the context of the original problem (by tracing dependency links when required). In the case that their reasons for failure no longer apply, but the initial reasons for selecting these alternatives are still present, consider reconstructing this alternate solution path in favor of continuing to apply and modify the present derivation (especially if quality of solution is more important than problem solving effort).

- In the event that a different decision is taken at some point in the rederivation, do not abandon the old derivation, since future decisions may be independent of some past decisions, or may still be valid (via different justifications) in spite of the somewhat different circumstances. This requires that dependency links be kept between decisions at different stages in the derivation.

- The derivational analogy should be abandoned in the event that a preponderance of the old decisions are invalidated in the new problem situation. Exactly what the perseverance threshold should be is a topic for empirical investigation, as it depends on whether there are other tractable means of solving this problem and on the overhead cost of reevaluating individual past decisions no longer be supported and may or may not have independent justification.

5. After an entire derivation has been found to apply to the new problem, store its divergence from the parent derivation as another potentially useful source of analogies, and as an instance from which more general plans can be formulated if a large number of problems of share a common solution procedure [8].

3.3. Efficiency Concerns

An important aspect of the derivational analogy approach is the ability to store and trace dependency links. It should be noted that some of the inherent inefficiencies in maintaining global consistency in a large deductive data base do not apply, as the dependency links are internal to each derivation with external pointers only to the problem description and to any volatile assumptions necessitated in constructing the resultant solution. Hence, the size of each dependency network is quite small, compared to a dependency network spanning all of memory. Dependencies are also stored among decisions taken at different stages in the temporal sequence of the derivation, thus providing the derivational analogy process access to causal relations computed at the time the initial problem was solved.

The analogical transformation process is not inherently space inefficient, although it may so appear at first glance. The sequence of decisions in the solution path of a problem are stored, together with necessary dependencies, the problem description, the resultant solution, and alternative reasoning paths not chosen. Failed paths are not stored, only the initial decision that was taken to embark upon that path, and the eventual reason for failure (with its causal dependencies), are remembered. Hence, the size of the memory for derivational traces is proportional to the depth of the search tree, rather than to the number of nodes visited. Problems that share large portions of their derivational structure can be so represented in memory, saving space and allowing similarity-based indexing. Moreover, when a generalized plan is formulated for recurring problems that share a common derivational structure, the individual derivations that are totally subsumed by the more general structure can be permanently masked or deleted. Those derivations that represent exceptions to the general rule, however, are precisely the instances that should be saved and indexed accordingly for future problem solving [14].

²This approach only works if the missing or violated premise relates to that part of the global state under control of the problem solver, such as acquiring a missing tool or resource, rather than under the control of an uncooperative external agent or a recalcitrant environment. The discussion of strategy-based counterplanning gives a more complete account of subgoal to rectify unfulfilled expectations [3, 6].

3.4. Implication for Knowledge Acquisition

At present the only effective means of encoding new information into an expert system is to painstakingly formulate each and every heuristic rule, test its function in new and old cases, tune and update the information until it integrates properly with previous knowledge, and verify that the new performance accords with the expert's advice. This is seldom a one pass process, since experts find it quite difficult to elicit their knowledge in a general and formal framework required to abstract the appropriate rules of behavior. Moreover, experts tire of the multiple iterations required to encode the knowledge in precisely the right form so that the expert system may make use of it in the problem solving process.

A viable alternative to explicit encoding of domain specific rules is to provide the reasoning system with full solutions to problems in the domain -- much like medical cases compiled in the literature that include descriptions of the problem, analysis performed, justifications for the diagnostic and treatment processes, alternatives considered, and outcomes of the selected treatment. In ill-structured domains, such as medicine and fault-diagnosis in electrical and mechanical systems, expert reason much more naturally from cases. In fact the educational process is heavily tilted towards this case based reasoning, leaving it up to the students to transfer the expertise of their teachers to new specific problems bearing appropriate similarity to the ones analyzed while learning. Thus, skill and knowledge are transferred implicitly, placing less burden on the expert and somewhat more burden in the student. The derivational analogy method proposed in this paper precisely addresses this mode of expertise transfer, where the student (expert system) observes and emulates the master (domain expert), gradually acquiring his/her/its own expertise.

3.5. Concluding Remarks

Derivational analogy bears closer resemblance to Schank's reconstructive memory [27, 28] and Minsky's K-lines [21] than to traditional notions of analogy. Although derivational analogy is less ambitious in scope than either of these theories, it is a more precisely defined inference process that can lead to an operational method of reasoning from particular experiential instances. The key notion is to reconstruct the relevant aspects of past problem solving situations and thereby transfer knowledge to the new scenario, where that knowledge consists of decision sequences and their justifications, rather than individual declarative assertions. To summarize, let us describe the process of derivational analogy in terms of the four criteria for analogical reasoning:

1. Two problems share significant aspects if their initial analysis yields the same reasoning steps, i.e., if the initial segments of their respective derivations start by considering the same issues and making the same decisions.
2. The either derivation may transferred to the new situation, in essence recreating the significant aspects of the reasoning process that solved the past problem.
3. Knowledge transfer is accomplished by reconsidering old decisions in light of the new problem situation, preserving those that apply, and replacing or modifying those whose supports are no longer valid in the new situation.

4. Problems and their derivations are stored in a large episodic memory along the line of Schank's MOPS [28], and retrieval occurs by replication of initial segments of decision sequences recalling the past reasoning process.

4. References

1. Balzer, R., "Imprecise Program Specification," Tech. report RR-75-36, USC/Information Sciences Institute, 1975.
2. Barstow, D.R., *Automatic Construction of Algorithms and Data Structures Using a Knowledge Base of Programming Rules*, PhD dissertation, Stanford University, Nov. 1977.
3. Carbonell, J. G., "Counterplanning: A Strategy-Based Model of Adversary Planning in Real-World Situations," *Artificial Intelligence*, Vol. 16, 1981, pp. 295-329.
4. Carbonell, J. G., "A Computational Model of Problem Solving by Analogy," *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, August 1981, pp. 147-152.
5. Carbonell, J. G., Larkin, J.H. and Reif, F., "Towards a General Scientific Reasoning Engine," Tech. report, Carnegie-Mellon University, Computer Science Department, 1983, CIP # 445.
6. Carbonell, J. G., *Subjective Understanding: Computer Models of Belief Systems*, Ann Arbor, MI: UMI research press, 1981.
7. Carbonell, J. G., "Experiential Learning in Analogical Problem Solving," *Proceedings of the Second Meeting of the American Association for Artificial Intelligence*, Pittsburgh, PA, 1982.
8. Carbonell, J. G., "Learning by Analogy: Formulating and Generalizing Plans from Past Experience," in *Machine Learning, An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell and T. M. Mitchell, eds., Tioga Press, Palo Alto, CA, 1983.
9. Clements, J., "Analogical Reasoning Patterns in Expert Problem Solving," *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, 1982.
10. Doyle, J., "A Truth Maintenance System," *Artificial Intelligence*, Vol. 12, 1979, pp. 231-272.
11. Duda, R. O., Hart, P. E., Konolige, K. and Reboh, R., "A Computer-Based Consultant for Mineral Exploration," Tech. report 6415, SRI, 1979.
12. Fikes, R. E. and Nilsson, N. J., "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," *Artificial Intelligence*, Vol. 2, 1971, pp. 189-208.
13. Gentner, D., "The Structure of Analogical Models in Science," Tech. report 4451, Bolt Beranek and Newman, 1980.
14. Hayes-Roth, F., "Using Proofs and Refutations to Learn from Experience," in *Machine Learning, An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell and T. M. Mitchell, eds., Tioga Press, Palo Alto, CA, 1983.

15. Kant, E., *Efficiency in Program Synthesis*, UMI Research press, Ann Arbor, MI, 1981.
16. Kling, R. E., "A Paradigm for Reasoning by Analogy," *Artificial Intelligence*, Vol. 2, 1971, pp. 147-178.
17. Laird, J. E. and Newell, A., "A Universal Weak Method," *Proceedings of the Eight Joint Conference on Artificial Intelligence*, 1983, (submitted).
18. McDermott, D. V. and Doyle J., "Non-Monotonic Logic I," *Artificial Intelligence*, Vol. 13, 1980, pp. 41-72.
19. McDermott, D. V., "Planning and Acting," *Cognitive Science*, Vol. 2, No. 2, 1967, pp. 71-109.
20. McDermott, J., "XSEL: A Computer Salesperson's Assistant," in *Machine Intelligence 10*, Hayes, J., Michie, D. and Pao, Y-H., eds., Chichester UK: Ellis Horwood Ltd., 1982", pp. 325-337.
21. Minsky, M., "K-Lines: A Theory of Memory," *Cognitive Science*, Vol. 4, No. 2, 1980, pp. 117-133.
22. Newell, A. and Simon, H. A., *Human Problem Solving*, New Jersey: Prentice-Hall, 1972.
23. Nilsson, N. J., *Principles of Artificial Intelligence*, Tioga Press, Palo Alto, CA, 1980.
24. Polya, G., *How to Solve It*, Princeton NJ: Princeton U. Press, 1945.
25. Reif, J. H. and Scherlis, W. L., "Deriving Efficient Graph Algorithms," Tech. report, Carnegie-Mellon University, Computer Science Department, 1982.
26. Sacerdoti, E. D., "Planning in a Hierarchy of Abstraction Spaces," *Artificial Intelligence*, Vol. 5, No. 2, 1974, pp. 115-135.
27. Schank, R. C., "Language and Memory," *Cognitive Science*, Vol. 4, No. 3, 1980, pp. 243-284.
28. Schank, R. C., *Dynamic Memory*, Cambridge University Press, 1982.
29. Schank, R. C., "The Current State of AI: One Man's Opinion," *AI Magazine*, Vol. IV, No. 1, 1983, pp. 1-8.
30. Shortliffe, E., *Computer Based Medical Consultations: MYCIN*, New York: Elsevier, 1976.
31. Swartout, W. and Balzer, R., "An Inevitable Intertwining of Specification and Implementation," *Comm. ACM*, Vol. 25, No. 7, 1982.
32. Wilensky, R., *Understanding Goal-Based Stories*, PhD dissertation, Yale University, Sept. 1978.
33. Wilensky, R., *Planning and Understanding*, Addison Wesley, Reading, MA, 1983.
34. Winston, P., "Learning by Creating and Justifying Transfer Frames," Tech. report AIM-520, AI Laboratory, M.I.T., January 1978.
35. Winston, P. H., "Learning and Reasoning by Analogy," *Comm. ACM*, Vol. 23, No. 12, 1979, pp. 689-703.

Concept Formation by Incremental Analogical Reasoning and Debugging

Mark H. Burstein
Department of Computer Science
Yale University
New Haven, Connecticut 06520

Abstract

This paper presents a model of analogical reasoning for learning. The model is based on two main ideas. First, that the analogies used in learning about an unfamiliar domain depend heavily on the use of previously formed causal abstractions in a familiar or base domain. Second, that these analogies are extended incrementally to related situations as needed. The analogical mapping component of CARL, a computer program that learns about the semantics of assignment statements for the BASIC programming language, is described as an illustration this kind of causally-driven analogical reasoning and learning. The model maps and debugs inferences drawn from several commonly used analogies to assignment, in response to presented examples.

Keywords: Learning, concept formation, analogical reasoning, causal abstraction, incremental mapping, debugging, cognitive modeling.

1. Introduction

It has been often been said among AI researchers that learning one new thing requires knowing an enormous amount beforehand. One kind of learning for which this is most obviously true in learning by analogy. In this paper, I show exactly why this is true as it relates to one specific kind of learning by analogy, the formation of new concepts in an unfamiliar domain from analogies presented in a text or by a teacher.

In developing the model presented here, I concentrated specifically on analogies commonly used in textbooks introducing students to computer programming in the BASIC language. The model was motivated in part by observations of students behavior when first introduced first to the BASIC language using such analogies.

Some unresolved problems with earlier models of analogical reasoning are addressed in the model. Because of the close relationship between our everyday notions of analogy and similarity, several models of analogical reasoning in AI have been developed around forms of partial pattern matching. Algorithms like those of Evans and Winston [Evans 88, Winston 80] were based on the assumption that a best partial match could be found by accumulating evidence for each of a number of possible object-to-object mappings between representations of two situations, and then choosing the one that scored highest. In these systems, evidence for a match was essentially the number of relational connections preserved between corresponding objects for a given alignment of objects.

The work reported here was supported in part by the Advanced Research Projects Agency of the Department of Defense, and monitored by the Office of Naval Research under contract No. N00014-75-C-111.

The object alignment that placed the largest number of relations and attributes in correspondence was considered the best match and the thus "correct" analogical interpretation.

This approach has several major drawbacks as a central component of a process model for analogical learning. First, it presupposes that well defined, bounded representational models of the situations in both the *base* or familiar domain and the *target* domain are available as inputs. But, in fact, the required prior representations of the objects and relations in the target domain may be wrong or inconsistent with the analogy, or may not exist at all. The point of presenting an analogy to a student is to aid him in the *construction* of a representation of a target situation or to correct problems in a prior representation. Since matching cannot be used to construct such a representation where there was none before, it cannot form the basis of a general theory of learning by analogy.

Another problem with theories based principally on the matching of descriptions, particularly as the complexity of these descriptions increases, is that conceptual representations for real situations may contain many objects not taking part in a specific analogy. Winston [Winston 80, Winston 82] suggested that attention to important relations, such as those connected by causal links, can reduce the computational complexity of the matching process to some degree. Yet, even in strictly causal models, sub-systems can quite often be usefully expanded to greater and greater levels of detail [deKleer and Brown 81, Collins and Gentner 82], thereby introducing new objects and relationships which may or may not play roles in the analogy. A system taking as input incomplete descriptions of analogical situations, such as those presented in texts, but with a large body of background causal and other knowledge for "filling out" such descriptions, would still require methods for narrowing the focus of the comparison. In particular, it must be possible to find analogical relationships between situations without detailed specification and pairing of *all* of the objects potentially present in representations of each situation.

What is required to address these objections is to replace bottom-up matching with an approach based on analogical mapping, using a set of heuristics to delimit what is to be "imported" from a base to a target domain at a given time. One such heuristic is to map previously formed abstractions, such as those embodying causal and planning rules. It can be argued that such structures are necessary in any case for interpreting and planning in situations in a familiar domain. Focusing on such structures, and their associated special cases and known problems, allows for a much more top-down form of analogical reasoning. This is exactly what is required when prior knowledge of the target domain is severely impoverished.

2. Student Protocols Used as Guidelines

Analogies found in typical introductory texts generally include statements suggesting correspondences between one or several classes of objects in the domains to be related. To be useful, these stated correspondences must be combined with the presentation of target domain actions or situations described in terms of plausible situations in the familiar domain. This is illustrated in the following text, used by one author to introduce the notion of computer variable.

A	B	C	D	E	F	G	H	I
3				12				
J	K	L	M	N	O	P	Q	R
		7						
S	T	U	V	W	X	Y	Z	
					5			

To illustrate the concept of *variable*, imagine that there are 26 little boxes inside the computer. Each box can contain one number at any one time. [Albrecht 78]

This analogy can be paraphrased "A *variable* is like a *box* in that numbers can be *inside* variables in some ways similar to the way objects can be inside boxes." To be applied in learning about assignment, it helps if specific actions associated with variables are also introduced, as in the statement "To *put* the number 5 in the variable X, type 'X=5'." No matter how it is presented, however, this kind of given information must be combined with a student's ability to access knowledge of the "box" domain, including specific concepts developed from experience in that domain. In this simple case, a student must at least be able to interpret statements about *putting* objects in boxes.

In the protocols I collected while tutoring introductory BASIC, I observed a number of *plausible*, though often incorrect, explanations being generated by students in response to examples and problems presented in the "programming" domain. This must in general be the case, since analogies are almost by definition imperfect correspondences between situations. The errors I observed occurred even with extremely simple assignment statements. For example, statements like "X=Y" were misinterpreted as indicating that the "box" Y was to be "placed inside" the "box" X. Such examples make it clear that an important part of the process of developing new concepts by analogy must be the incremental debugging of the inferences derived from the analogy.

Sources of alternate hypotheses, including *additional analogies*, can be quite useful in this debugging process. The following protocols illustrate this quite clearly. After explaining to one subject (Perry, age 10) that there were many boxes in the computer, and that each one had a name that he could choose, the following occurred.

M: Suppose there's a box called X and we're going to store the number 5 in there. How would you do that?
P: the variables .. uh X .. X... no
M: You have to tell it...
P: Put the number 5 in the variable X.
M: You have to give it a command that will make it do that.
Now, here we have an example...suppose I type 'B=10'.
P: Oh, and if you want to store this 10 you write...

M: Now I typed that in, so now it's going to remember that, ok?
P: that B equals 10.
M: It's got a box called B, and inside it is 10.
P: So you write the box and then the number that you want to store in it?
M: Yes, and you put in an equal sign to tell it to do that.

Here, three common analogies to variables and assignment are mentioned: putting numbers in variables is like putting objects in boxes, computers use variables to *remember* things they are *told*, and assignment statements are like algebraic equalities. At first glance, this might seem more confusing than helpful. Yet, each analogy can be shown to play a useful, often complementary, role in developing a working understanding of assignment. Tutorial textbooks often use *at least* two of these analogies. If they are not presented explicitly as analogies, the normal language of the computer science domain suggests their presence. We often speak of computer *memory*, and the equal sign is used in a number of languages to denote assignment.

The real test of a tutorial analogy is how it affects one's understanding of new situations. The following shows one way in which having several analogies can be more helpful than having just one. As the dialogue above continued, I tried to illustrate a point about transferring values from one variable to another. I typed 'P=10' and then 'Q=P', and asked:

M: So, what's in P now?
P: Oh. Nothing.
M: Nothing?
P: 10! and then Q is also.
M: What do you think it is? Is it nothing or 10?
P: Let's find out. First let's see...
M: Well, what do you think it is?
P: If you have two boxes, and you moved...
You moved or it equals to? You moved
what's in P to Q so there's nothing in it,
or did you only put the same number in Q
that's in P? I think it's 10.
M: You think it's 10?
P: Because you don't say that, um, move P at all...
take P out. You only said that Q equals the
same as P. So if it equals, it has to be 10,
because if there's no 10 in it, Q wouldn't be
equal to it.

Here, Perry clearly made the inference that if "Q=P" was analogous to moving an object from one box to another, then the number that had been in P must now be in Q. Since objects, when moved, are no longer at their original location, P must now be "empty".

When pressed, Perry's uncertainty about this conclusion caused him to come up with an alternate explanation based on an entirely different line of reasoning, using a different analogy. If typing "Q=P" is treated as a statement to the computer that the two variables have equal values, based on the normal algebraic interpretation, then from the fact that P was originally 10 one could *conclude* that Q was also 10, while P's value stayed the same.

The use of algebra knowledge in the explanation Perry settled upon in this exchange did not stop him from further use of the box analogy. He continued to use both models in analyzing new situations. It was simply the case that algebra provided what seemed like a more satisfactory answer to this particular problem.

Such errors and their explanations played an important role in developing the process model implemented in CARL. Because these incorrect hypotheses represent plausible analogical extensions of situations in the base domains, it can be assumed that they are produced by the same analogical reasoning process that produces correct answers in other cases. It was by modeling errors observed in the protocol data, as well as correct inferences, that I was able to formulate the model of analogical reasoning presented here.

3. A Program that Learns about Assignment

Many aspects of the overall model are illustrated in the behavior of the computer program. CARL learns about various kinds of assignment statements by building and combining inferences drawn from several different analogies. The program formulates many of the hypotheses and a number of the errors I found students made as they tried to learn from the same analogies (see [Burstein 81] for other examples). The following is the English equivalent of one portion of a dialogue that CARL engages in (actual inputs are in an extended version of Conceptual Dependency [Schank and Abelson 77]):

```
>> The computer remembers a number by putting it
    in a variable.
>> A variable is like a box.
>> To put the number 5 in the variable named X,
    type 'X=5'.
>> How do you get the computer to remember the
    number 9?
CARL: Type 'X = 9'
>> Suppose I then type 'X=6'.
>> Now what's in the variable X?
CARL: 6 is in X and 9 is in X
>> No, the 9 replaces the 6 in X.
CARL: OK
>> Suppose I type 'Y=5' and then 'X=Y'.
>> Now what's in the variable X?
CARL: Y is in X
>> No, it means put the number in Y into X.
CARL: OK
>> Now what's in the variable Y?
CARL: Nothing.
>> Guess again.
CARL: 5 is in Y (by analogy to equality)
>> Correct.
CARL: OK
```

CARL develops, with the aid of a tutor, semantic representations for most common types of assignment statement. It concurrently forms rules about how to parse them, infer the logical results of their execution, and generate them as components of simple plans. In the process, it makes a number of the mistakes observed in the protocols of people learning the same material.

4. An Initial Structure Mapping Theory

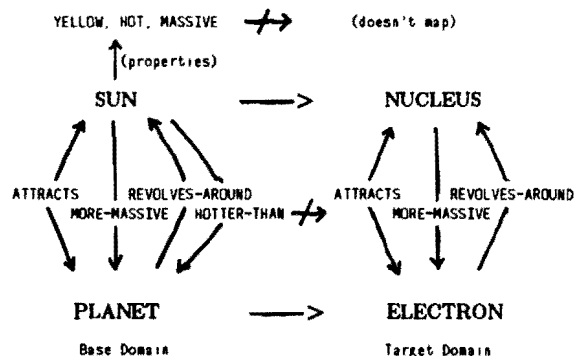
The analogical reasoning model in CARL was strongly influenced by some psychological studies of analogical learning. Gentner [Gentner 82, Gentner and Gentner 82] outlined a cognitive model of learning from scientific or "explanatory" analogies that dealt with some of the problems I have mentioned, though not others. The analogies she considered included such statements as:

The hydrogen atom is like the solar system.

Electricity flows through a wire like water through a pipe.

The model Gentner proposed for learning from such analogies, unlike the matching-driven models, did not require a complete prior representation of the target domain. However, it was underspecified as a process model. By her model, relations, or predicates connecting several objects or concepts, are mapped *identically* from one domain to the other under a prespecified object-object correspondence. After identical first-order relations have been mapped to relate corresponding objects in a target situation, second-order predicates, such as causal links between relations, are also mapped. Although this model does suggest a way to map new structures into an unfamiliar domain, it does not give a good account of how corresponding objects are first identified, nor does it constrain *which* relations are mapped. It also did not allow for mappings between non-identical relations, which I will argue is often necessary.

The need to constrain the set of relations mapped can be seen from Gentner's representation of the solar system model, and the mapping that she described to a model for the atom.



Here, the sun is related to a planet by the predicate HOTTER-THAN, as well as ATTRACTS and REVOLVES-AROUND, two predicates which are themselves causally related (not shown). Gentner claimed that the HOTTER-THAN relation was *not* mapped to the atomic model, in accord, I think, with most people's intuitions about this analogy. However, Gentner's specification of the mapping process could not predict this. It is also clearly the case that many other attribute comparisons, such as BRIGHTER-THAN, could also be part of a description of a solar system. Presumably, these relations would not be mapped either.

The explanation provided by Gentner for this was in terms of a general condition on the mapping process, essentially that "predicates are more likely to be imported into the target if they belong to a system of coherent, mutually constraining relationships, the others of which are mapped." [Gentner and Gentner 82] As the above example shows clearly, some heuristics of this form are needed.

In CARL, this general condition is reformulated as a top-down constraint on the relations considered for mapping. When a causally connected structure can be found in memory to support a described base domain situation, only relations taking part in that structure are considered for mapping. Within such a structured set of relations, simple attribute comparisons like HOTTER-THAN, LARGER-THAN, etc. are not mapped if the objects in the target domain cannot be compared on the same scale, and no corresponding attributes are suggested by the teacher. Thus, in applying the box analogy, CARL decides that the precondition that numbers "fit" inside variables should be dropped from the causal structure.

All of this is made easier in the context of a language understanding system that activates memory structures as a normal part of the interpretation process. When CARL is presented with a statement of a novel analogy, it uses the given object associations and base domain predicates in the analogy statement to form a partial description of a base domain situation that it can support as plausible with causal and goal/plan structures retrieved from memory.

The result of mapping a causally-connected structure found under these conditions is the formation of a new, parallel causal structure characterizing the target example. Objects in the target example are made to fill roles in that structure, using stated object correspondences between the domains when available. Other object correspondences are only formed by virtue of the roles correspondences between the mapped structures. So, for example, from a causal abstraction indicating that the result of putting a physical object in a container is the state "the object is INSIDE the container," CARL concludes that one result of an assignment is a parallel relation between variables and numbers. Thus, by corresponding roles, an indirect correspondence is formed between physical objects that go into boxes and numbers.

This causally directed mapping process is thus able to form a new structure where none existed before, while allowing correspondences between relations to be formed with some consideration of what is known of the objects and relations in the target domain. This approach to analogical concept formation parallels Carbonell's work on analogical problem solving [Carbonell 83]. Carbonell outlined a problem solving process whose first step was to be reminded of a solution to a similar problem. The process model he proposed then modified the components of the recalled plan to satisfy the needs of the new problem using a set of operators that preserved, as much as possible, the temporal and logical goal/subgoal structure of the original solution. Both Carbonell's model and mine were strongly influenced by Schank's theory of human memory organization [Schank 82], and the effects of that organization on the processes of interpreting, planning, and learning about new situations.

5. Mapping to Non-identical Relations

Another problem with Gentner's model was the claim that all relations are mapped "identically" between analogous situations. While this might be true in analogies between purely spatial descriptions of situations, including the standard geometric analogies dealt with by Evans, it is much too strong a claim in general. When analogies are formed between physically realizable situations and purely abstract ones, as in mathematics and computer programming, it is impossible to maintain the "identical predicate" mapping model.

Probably the most important thing implied by the analogy between boxes and variables is the fact that variables can "contain" things. That is, the relationship that exists between a box and an object inside the box is, in some way, similar to the relationship between a variable and the number associated with that variable. These relations are similar primarily because of the actions and plans each is involved in. One can put things in boxes, and assignment provides a way to to "put" numbers "in" variables as well. The principle function of the relation representing containment in variables is its role in abstract plans like STORE-OBJECT.

Students learning to program are generally aware that computers can manipulate numbers, and that the reason one learns to program is to be able to direct the computer's actions. This knowledge may be used to infer that the action of putting a number in a variable will be used in plans to manipulate numbers. Whether or not this occurs immediately, however, the fact that the analogy between boxes and variables relates physical objects to abstract concepts (numbers) suggests that the actual pre-conditions and side effects of assignment in the programming domain may be quite different from the conditions on placing objects in boxes. Though students only gradually discover how these situations differ, it is important that such differences not invalidate the analogy entirely. In a computational inference system, this must be reflected in the predicates representing the relations in each domain.

The problem from the standpoint of Gentner's model, is that the relationship which gets mapped from the "box world" to the "computer world" is exactly that of *physical containment*. The interpretation that results from copying this relation into the programming domain is that a number is physically INSIDE-OF a variable. Under normal circumstances, someone first learning to program will have no idea what computer variables are, but should know that numbers are not physical objects, and not expect that all of the inferences involving the relation INSIDE will apply when numbers are placed "in" variables.

This problem can be characterized as one of *levels of abstraction*. Depending on how much is known about the objects in the target domain when the analogy is presented, it may or may not be reasonable to map the most specific version of a relation from one domain to another. When mapping a relation identically leads to the violation of a constraint on one of the slots in that relation, then the relation meant in the target domain must be one sharing some of the properties of the base domain relation, and not others.

In CARL, when an attempt to map a relation directly results in such a constraint violation, a *virtual relation* is formed in the target domain that is a "sibling" of the corresponding base domain relation, or an ancestor at some higher level in the generalization hierarchy of relational

predicates. The constraints initially placed on the slots in new virtual relations are determined primarily from the classes of the objects related in the target domain examples presented.

When CARL is given the box analogy, it finds that mapping objects to numbers violates a normal constraint on the INSIDE relation. Instead, CARL forms a new predicate to relate variables and their "contents". This relation, hereafter called INSIDE-VAR, is initially given the constraints that the "contents" slot be a number, and the "container" slot be a variable, based on the objects in accompanying example "X=5". Inferences are associated with this new relation as they are successfully mapped from the box domain, learned independently in the new domain, or inherited from other analogies.

The final result of mapping the structure PUT-IN-BOX, describing the causal relations involved in putting an object in a container, looks roughly as follows. Notice particularly that the PTRANS predicate indicating *physical* transfer was also replaced by the more general predicate TRANS because the object "moved" in the target domain was not a physical object.

PUT-IN-BOX	----->	PUT-IN-VAR
role-variables:		
R-ACTOR (a HUMAN)		PIV-ACTOR (a COMPUTER)
R-BOX (a BOX)		PIV-BOX (a VARIABLE)
R-CONTENTS (a PHYSOBJ)		PIV-CONTENTS (a NUMBER)
actions:		
(PTRANS		(TRANS
actor R-ACTOR		actor PIV-ACTOR
object R-OBJ		object PIV-OBJ
from (unknown)		from (unknown)
to (INSIDE of R-BOX))		to (INSIDE-VAR of PIV-BOX))
preconditions:		
(*note		(*not*
(INSIDE or R-BOX is R-OBJ))		(INSIDE-VAR of PIV-BOX is PIV-OBJ))
(SMALLER than R-OBJ is R-BOX)		- dropped -
results:		
(INSIDE of R-BOX is R-OBJ)		(INSIDE-VAR of PIV-BOX is PIV-OBJ)

6. Overview of the Analogy Mapping Process

In general, then, CARL develops simple causal or inferential structures in a target domain by retrieving structures in memory from a familiar domain, and adapting them using a top-down mapping process that focuses on the causal/temporal links explicitly specified in those structures. The predicates mapped are subject to transformation within their abstraction hierarchy, as described above. Subsequent use of an analogy may occur when new examples are presented for which no explanation can be found in the target domain or when problems are presented, asking for plans or actions to achieve specific analogically stated goals. The latter, in this case, is generally a request for the generation of an assignment statement satisfying some specific goals or constraints.

In answering a question, CARL always looks first for memory structures in the domain it is learning about. Failing this, it tries known analogies. Thus, subsequent access to base domains is always for the purpose of mapping new, related structures -- related action situations, or more detailed, context-specific versions of previously mapped structures that account for additional preconditions or results.

The mapping process tries to form structures in the target domain, under the following general constraints:

- Corresponding predicates must be of the same class (action, relation, plan step, plan, goal, etc.).
- Corresponding predicates in the target structure are related together by causal or temporal links corresponding to those in the base domain structure.
- Corresponding case slots of analogically related predicates must consistently be filled by parallel roles of the two structures related.

CARL keeps a record called an AMAP detailing all of the object, role, and predicate correspondences developed. AMAPS are extended as needed to include new correspondences as they are found.

Because the AMAP uses *role* correspondences as well as object class correspondences, the relationships between objects of two domains can actually change quite subtly in dealing with new problems. Several bugs I observed in protocols depend on this distinction. In answering the question "How would you put one more in X?" I occasionally received the response "X=1". Also, when asking questions like "What is in X?" after typing "X=7" and then "X=6", I occasionally heard the answer "13".

These responses can clearly be interpreted in terms of the box analogy. For the first problem, it appears that the answer "X=1" was based on the fact that placing another object in a box that already contains some objects will cause it to contain "one more." However, both these solutions to both problems require that the relationship between contents of boxes and contents of variables be a mapping from a *set* of objects to the *cardinality* of that set. This is quite different from the simpler mapping of an object to a number. In modeling these responses, it was important that the analogy formation process relate objects primarily in terms of their functional roles in specific situations, and only secondarily determine how object features corresponded.

7. Incremental Analogical Reasoning

Even when analogies are based on simple actions, the specific inferences retrieved in support of new examples may vary considerably, depending on the context. For example, throwing a rock at a brick wall and throwing one at a glass window are immediately known to have very different consequences. Though an analogy to a thrown rock might imply indirectly that the specific inferences made in such alternate contexts will have correlates in a target domain, in practice, each such target situation must be explored before one can be said to have learned to model the new domain.

Extending analogies in this fashion is an error-prone process. In the protocols I examined, such errors appeared only when the context in the target domain made them potentially useful inferences. When I first introduced statements like "X=Y" to Perry, it was necessary that I explain that this meant X was given the value that Y had previously. Perry then inferred that Y must contain "nothing". CARL also extends analogies to such context specific inferences only when they form part of the interpretation of presented examples.

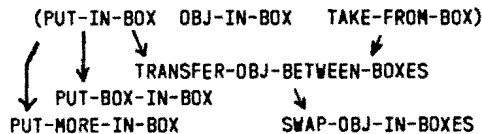
This behavior is modeled in CARL by retrieval and mapping of *related* causal structures from a base domain. CARL first uses the box analogy to support interpreting

" $X=Y$ " in terms of the action move one box into another. When this is corrected by stating that the *contents* of Y is moved, CARL tries mapping an object-transfer model, containing the "bug" that the object moved is no longer in Y . Information about the mapping of the prototype "put an object in a box," saved in an AMAP relating the two domains, is used both in finding the structures representing these more specific actions and subsequently to map these structures back to the programming domain.

The memory organization used in CARL for knowledge of simple action-based domains involving familiar objects is an extension of an object-based indexing system developed by Lehnert [Lehnert 78, Lehnert and Burstein 79] for natural language processing tasks. So that CARL can also retrieve a variety of special case situations, the memory retrieval process in CARL was augmented using discrimination networks based on the specification hierarchy model of episodic memory developed by Lebowitz and Kolodner [Lebowitz 80, Kolodner 80]. In addition, precondition and result indices were added so that actions and simple plans could also be retrieved in response to requests for the achievement of specific goals. Any or all of these forms of indexing may be used in finding a suitable structure to map. For familiar domains, the system assumes a large set of fairly specific causal inference structures exists in memory at the beginning of the learning process. No attempt is currently made to construct composite causal structures on the fly, though clearly that might be necessary with more complex analogies.

CARL contains the following structures describing the effects of some simple actions involving containers.

Situations using BOX as a CONTAINER:



Part of the specialization network
for things "INSIDE" boxes

In the computer run shown earlier, an initial mapping from the box domain was formed from the causal structure PUT-IN-BOX. Thereafter, specializations of that structure were available as new examples were presented. In addition, once the new containment relation was formed for variables, expectations were established for the other "primitive" situations involving containers. Thus, from the fact that variables can "contain" numbers, CARL expected that they might also be "put in" or "removed."

Many of the structures formed by the mapping process contained erroneous inferences. These structures were "debugged" locally, if possible, or simply thrown out. From those that were debugged, corrections were allowed to propagate downward in the situation hierarchy formed in the target domain from the initially formed prototype to variants of that structure mapped subsequently. For example, the fact that ' $X=5$ ' removes the old values of X , rather than "adding" to it, also applies to cases like ' $X=Y$ '. The inheritance mechanism in CARL that handles this is active only when new structures are formed, so it was important in teaching CARL that it be shown these bugs early on. This model seems to

suggest at least one reason why initial analogical prototypes are best kept as simple as possible, and problems in them corrected quickly.

Also as indicated in the first human protocol shown above, CARL develops parsing and generation rules for each class of assignment statement successfully represented. These rules are developed during the final stage of the analysis of each example.

8. Using Multiple Analogies

CARL often finds better explanations for assignment statements using the analogical relationship between assignment and equality than it does with the box model. In general, the box model does not help much in interpreting arithmetic expressions. However, with some exceptions, when assignment statements have the effect of associating values with variables that previously had none, the correct interpretation can be found by interpreting the statement as an equality. This is certainly true when all variables to the right of the equal sign have known values.

CARL first notices that algebra might be useful in learning about assignment when it sees the "=" sign in statements like " $X=5$ ". As it builds a new meaning for "=", it discovers the earlier definition. Parsing " $X=5$ " as an equality, CARL produces the an alternate explanation for the effect of that statement on the computer. Interpreting the statement in context, by analogy to what a person would do when hearing that statement, CARL concludes that the effect of *hearing* the statement is the storing of a new fact, "the value of the variable X is 5".

Since this inference can only have been made when the statement was typed into the computer, its causal/temporal effect on the machine can be related to the "physical" model of the same assignment statement developed using the box analogy. That is, the statement initiates a mental action (or sequence) leading to an association between a variable and a value. The model formed using the box analogy also resulted in a relationship between the "box" X and the number 5, so, by comparing the causal effects of each interpretation of the same action, CARL forms a mapping from the predicate VAR-VALUE in the algebra domain to the predicate INSIDE-VAR that it had previously constructed in the programming domain.

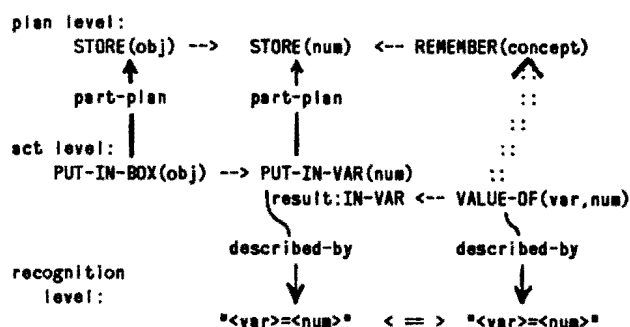
Once this association is formed, CARL can interpret the effects of many other assignment statements by parsing them as equalities, using algebra rules to determine effects on variables, and then mapping the statement interpretation and it's causal effects. The structures mapped are used both to replace specific inferences developed from other analogies when they are found to be in error, and to develop new structures.

It should be emphasized here that a causal/temporal element must be introduced in going from algebra to assignment. To relate the algebraic analogy to the box model correctly, it is therefore important that there be an agent *interpreting* the algebra statements and deriving new facts or relations from that process. It is only by the noticing that the relationship formed between a variable and its value as a *result* of this interpretation process is like the relationship between variables and numbers developed previously that CARL decides the the analogy may be useful. This analysis is supported by and makes use of the third analogy CARL knows of, the

analogy between the computer and a human information processor. It is because people can interpret algebra statements that CARL, and Perry, deemed it reasonable that the computer could as well.

The interactions between the three analogies used by CARL are summarized in the following diagram. It should be noted that the each analogy is represented and related at several levels of description, but that the functions served by each are quite different. The box model provides the initial causal model of the assignment domain. The algebra domain provides knowledge of numbers, the operations that can be performed with them, and the symbols for representing them. The human processor model is active primarily at the planning level, providing *reasons* for doing many of the operations that computers can perform.

Interactions between Three Analogies



9. Conclusions

In developing CARL, I have been concerned with a number of related issues in the learning of basic concepts in a new domain by a combination of incremental analogical reasoning, and the use of multiple analogical models. I have tried here to motivate the need for top-down use of abstractions in this process. This was found to be necessary in forming rules about assignment in CARL, both to limit the analogical reasoning required to create initial models of concepts in the new domain, and to allow for incremental debugging of the many errors that can result from the use of analogies. The process described here is heavily teacher-directed, but allows for fairly rapid development of a working understanding of basic concepts in a new domain.

Acknowledgements: I would like to thank Dr. Chris Riesbeck and Larry Birnbaum for many helpful comments on drafts of this paper.

References

- Albrecht, R., Finkel, L. and Brown, J.R.. *BASIC for Home Computers*. John Wiley & Sons, Inc., New York, NY., 1978.
- Burstein, Mark H. Concept Formation through the Interaction of Multiple Models. Proceedings of the Third Annual Conference of the Cognitive Science Society, Cognitive Science Society, August, 1981, pp. 271-274.
- Carbonell, Jaime G. Learning by Analogy: Formulating and Generalizing Plans from Past Experience. In *Machine Learning: An Artificial Intelligence Approach*, Michalski, R. S., Carbonell, J. G. and Mitchell, T. M., Ed., Tioga Publishing Company, Palo Alto, California, 1983, pp. 137-162.
- Collins, Allan and Gentner, Dedre. Constructing Runnable Mental Models. Proceedings of the Fourth Annual Conference of the Cognitive Science Society, Cognitive Science Society, August, 1982, pp. 86-89.
- de Kleer, J. and Brown, J. S. Mental Models of Physical Mechanisms and their Acquisition. In *Cognitive Skills and Their Acquisition*, Anderson, John R., Ed., Lawrence Erlbaum and Assoc., Hillsdale, NJ, 1981, pp. 285-309.
- Evans, Thomas G. A Program for the Solution of Geometric Analogy Intelligence Test Questions. In *Semantic Information Processing*, Marvin L. Minsky, Ed., M.I.T. Press, Cambridge, Massachusetts, 1968.
- Gentner, Dedre. Structure Mapping: A Theoretical Framework for Analogy and Similarity. Proceedings of the Fourth Annual Conference of the Cognitive Science Society, Cognitive Science Society, August, 1982, pp. 13-15.
- Gentner, D. and Gentner, D. R. Flowing waters or teeming crowds: Mental models of electricity. In *Mental Models*, Gentner, D. and Stevens, A. L., Ed., Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1982.
- Kolodner, Janet L. Retrieval and Organizational Strategies in Conceptual Memory: A Computer Model. Tech. Rept. 187, Yale University. Department of Computer Science, 1980. Ph.D. Dissertation
- Lebowitz, M. *Generalization and Memory in an Integrated Understanding System*. Ph.D. Th., Yale University, October 1980.
- Lehnert, W. G. Representing Physical Objects in Memory. Tech. Rept. 131, Yale University. Department of Computer Science, 1978.
- Lehnert, W.G. and Burstein, M.H. The Role of Object Primitives in Natural Language Processing. Proceedings of the Sixth International Joint Conference on Artificial Intelligence, IJCAI, August, 1979, pp. 522-524.
- Schank, R.C.. *Dynamic memory: A theory of learning in computers and people*. Cambridge University Press, 1982.
- Schank, R.C. and Abelson, R.. *Scripts, Plans, Goals and Understanding*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1977.
- Winston, P. "Learning and reasoning by analogy." *CACM* 23, 12 (December 1980).
- Winston, P. "Learning new principles from precedents and exercises." *Artificial Intelligence* 19 (1982), 321-350.

PROGRAMMING BY ANALOGY

Nachum Dershowitz*
 Department of Computer Science
 University of Illinois
 Urbana, IL 61801

ABSTRACT

Analogy is one tool that automatic programming systems can use to learn from experience, just as programmers do. We illustrate how analogies between program specifications can be used to debug incorrect programs, modify existing programs to perform different tasks, derive abstract schemata from given sets of cognate programs, and instantiate schemata to solve new problems.

An analogy between the specification of a given program and that of a new problem is used as the basis for modifying the given program to meet the new specification. Debugging is a special case of modification: if a program computes wrong results, it must be modified to achieve the intended results. For program abstraction, an analogy is sought between the specifications of the given programs; it may then be used to transform an existing program into an abstract schema that embodies the shared technique. By comparing the specification of the derived schema with a given concrete specification, and formulating an analogy between them, an instantiation of the schema may be found that yields the desired concrete program.

Key terms: learning, analogy, automatic programming, program modification, debugging, abstraction, instantiation, program transformations, program schemata.

Analogy pervades all our thinking, our everyday speech and our trivial conclusions as well as artistic ways of expression and the highest scientific achievements.

—George Polya

1. INTRODUCTION

Programming begins with a specification of what the envisioned program ought to do. It is the programmer's job is to develop an executable program satisfying that specification. Yet, only a small fraction of a programmer's time is typically devoted to the creation of original programs *ex nihilo*. Rather, most of his effort is normally directed at debugging incorrect programs, adapting known techniques to specific problems at hand, modifying existing programs to meet amended specifications, extending old programs for expanded capabilities, and abstracting ideas of general applicability into "subroutines."

The goal of research in "automatic programming" is to formalize methods and strategies used by programmers so that they may be incorporated in automatic, and interactive, programming environments. In our view, program development systems should incorporate formal tools for transforming and manipulating programs. In this paper, we illustrate how analogies might be used by such a system for that purpose.

The importance of analogical reasoning has been stressed by many, from Descartes to Polya. The use of analogy in automated problem solving was proposed in [Kling71]. Other works employing analogy as an implement in problem solving include [Brown76], [McDermott79], and [Winston80]. The use of analogies to guide the modification of programs was proposed in [MannaWaldinger75] and pursued in [DershowitzManna77] and [UlrichMoll77].

Programmers improve with experience by assimilating programming techniques that are encountered, and judiciously applying the ideas learned to new problems. One way in which knowledge can be applied is by modifying a known program to achieve some new goal. For example, a program that uses the "binary-search" technique to compute square-roots might be transformed into one that divides two numbers. We show how to modify programs by first finding an analogy between the specification of the existing program and that of the program we wish to construct. This analogy is then used as a basis for transforming the existing program to meet the new specification. Program debugging is a special case of modification: if a program computes wrong results, it must be modified to achieve the intended results.

All our programs are assumed to be annotated with an *output specification* (stating the desired relationship between the input and output variables upon termination of the program), an *input specification* (defining the set of legal inputs on which the program is intended to operate), and *invariant assertions* (relations that are known to always hold at specific points in the program for the current values of variables) demonstrating its correctness. The invariant assertions play an important role in deriving analogies.

The idea that programs should be constructed by a series of transformations has been widely promoted. Modification differs from such transformations in that correctness with respect to the original specification is *not* preserved. What we want is for the resultant program to be correct with respect to the *transformed* specification. Correctness-preserving transformations and specification-changing modifications are thus complementary. A scenario of computer-aided programming and debugging appeared in [Floyd71]. The HACKER system [Sussman75] constructed programs by trying out alternatives and attempting to debug them when necessary; other knowledge-based or plan-based

*This research supported in part by the National Science Foundation under Grant MCS-79-04897.

debugging systems have been constructed, as well. [KatzManna75] and [Sagiv76] describe debugging techniques based—like our method—on invariant assertions.

Program modification is not the only manner in which a programmer utilizes previously acquired knowledge. After coming up with several modifications of his first “wheel,” he is likely to formulate for himself (and perhaps for others) an abstract notion of the underlying principle and reuse it in new, but related, applications. Program “schemata” are a convenient form for remembering such knowledge. A schema may embody basic programming techniques and strategies (e.g. the “generate-and-test” paradigm or the “binary-search” technique) and contains abstract, uninstantiated symbols, in terms of which its specification is stated.

The abstraction of a set of concrete programs to obtain a program schema and the instantiation of abstract schemata to solve concrete problems may be viewed from the perspective of modification methods. This perspective provides a methodology for applying old knowledge to new problems. Beginning with a set of programs sharing some basic strategy and their correctness proofs, a program schema that represents their analogous elements is derived. Preconditions for the schema's applicability are derived from the correctness proofs. The resultant schema's abstract specification may be compared with a given concrete specification to suggest an instantiation that yields a concrete program when applied to the schema. If the instantiation satisfies the preconditions, the correctness of the new program is guaranteed.

To date there has been a limited amount of research on program abstraction. The STRIPS system [FikesHartNilsson72] generalized the loop-free robot plans that it generated; HACKER “subroutinized” and generalized the “blocks-world” plans it synthesized, executing the plan to determine which program constants could be abstracted. [Dershowitz-Manna75] suggested using the proof of correctness of a program to guide the abstraction process; that idea was pursued further in [Dershowitz81]. [Gerhart75] and others have advocated and illustrated the use of schemata as a powerful programming tool. A collection of such schemata, along with a catalog of correctness-preserving program transformations, could serve as part of an interactive program-development system.

In the following sections we trace the life-cycle of an example program. The example illustrates some of the kinds of transformations programs undergo and how analogy can be used as a guide. We begin with an imperfect program to compute the quotient of two real numbers. After the program is *debugged*, it is *modified* to approximate the cube-root of a real number. Underlying both the division and cube-root programs is the binary-search technique; by *abstracting* these two programs, a binary-search schema is obtained. This schema is then *instantiated* to obtain a third program, one to compute the square-root of an integer.

2. DEBUGGING

Consider the problem of computing the quotient q of two nonnegative real numbers c and d within a specified (positive) tolerance e . These requirements are conveniently expressed in the form of the following skeleton program:

```
P1: begin comment real-division specification
  assert 0 ≤ c < d, e > 0
  achieve |c/d - q| < e varying q
end
```

The *achieve* statement,

achieve $|c/d - q| < e$ varying q ,

contains the *output specification* which gives the relation between the variables q , c , d , and e that we wish to be attained at the end of program execution: the (absolute value of the) difference between the exact quotient c/d and the result q should be less than e . The clause **varying** q indicates that of the variables in the specification, only q may be set by the program; the other variables, c , d , and e , contain input values that remain fixed. The *input specification* defines the set of inputs on which the program is intended to operate. Assuming that we wish our program to handle the case when the quotient is in the range 0 to 1, that is when the numerator c is smaller than the denominator d , the appropriate input specification is contained in the **assert** statement,

assert $0 \leq c \leq d, e > 0$,

attached to the beginning of the program. For the problem at hand, we assume that no general real-division operator $/$ is available, though division by powers of two (“shifts”) is permissible.

Now let us imagine that a programmer went ahead and constructed the following program:

```
T2: begin comment suggested real-division program
  B2: assert 0 ≤ c < d, e > 0
  purpose |c/d - q| < e
  purpose q ≤ c/d < q + s, s ≤ e
    (q, s) := (0, 1)
  loop L2: suggest q ≤ c/d < q + s
    until s ≤ e
    purpose q ≤ c/d < q + s, 0 ≤ s < s1
      if d · (q + s) ≤ c then q := q + s fi
      s := s/2
    repeat
  suggest q ≤ c/d < q + s, s ≤ e
  E2: suggest |c/d - q| < e
end
```

The *purpose* statement,

purpose $|c/d - q| < e$,

is a comment describing the intent of the code following it. The statement

suggest $|c/d - q| < e$

contains the programmer's contention that the preceding code actually achieves the desired relation, i.e. the relation $|c/d - q| < e$ holds for the value of q when control reaches the end of the program. The comment

purpose $q \leq c/d < q + s, s \leq e$

indicates that the programmer's intention is to achieve the desired relation $|c/d - q| < e$ by achieving the subgoals $q \leq c/d \leq q + s$ and $s \leq e$. Achieving these relations is sufficient for $|c/d - q| < e$ to hold. For this purpose the programmer

constructed an iterative loop intended to keep the first relation invariantly true while making progress towards the second. The suggested invariant is contained in the statement

suggest $q \leq c/d < q + e$

at label L_2 . The goal of the loop body is

purpose $q \leq c/d < q + e, 0 < e \leq s_{L_2}$,

where s_{L_2} denotes the value of the variable s when control was last at the label L_2 . This means that the value of s is to be less than it just was at the head of the loop. The two loop-body statements are accordingly repeated (zero or more times) until the test $s \leq e$ becomes true, at which point the loop will be exited.

We know what the above program was intended for, and we know that it does not always fulfill those intentions. However, before we can debug it, we need to know more about what it actually does. This can be accomplished by examining the code and annotating the program with the discovered invariant relations (see [DershowitzManna81]). It is not difficult to derive the loop invariant $d \cdot q \leq c < d \cdot (q + 2 \cdot s)$. This remains true when the loop exit is taken; along with the exit test $s \leq e$, it implies that upon termination the output invariant $|c/d - q| < 2 \cdot e$ holds. Note that the desired relation $|c/d - q| < e$ is *not* implied.

Now that we know something about what the program does, we can try to debug it. Our task is to find a correction that changes the actual output invariant

assert $|c/d - q| < 2 \cdot e$

to the desired output invariant

suggest $|c/d - q| < e$.

We go about this by first looking for a way to transform the actual invariant into the desired one; we then try to apply the same transformation to the program, hopefully correcting the error thereby. Accordingly, we would like to find an analogy between the actual output invariant and the desired specification; we write

$$|c/d - q| < 2 \cdot e \iff |c/d - q| < e.$$

The obvious difference between the two expressions is that where the first has $2 \cdot e$, the second has just e . So, we can reduce the analogy to simply

$$2 \cdot e \iff e.$$

We can, therefore, transform the insufficient $|c/d - q| < 2 \cdot e$ into the desired $|c/d - q| < e$ by replacing e with $e/2$, i.e. by applying the transformation $e \Rightarrow e/2$.

So far we know that the transformation $e \Rightarrow e/2$, applied to the output invariant $|c/d - q| < 2 \cdot e$, yields the desired output specification $|c/d - q| < e$. That same transformation is now applied to the whole annotated program. The symbol e appears once in the program text: the exit clause $s \leq e$ accordingly becomes $s \leq e/2$. The symbol also appears four times in the invariants; for example, the input assertion $e > 0$ transforms into $e/2 > 0$ which is equivalent to $e > 0$. The transformed program is

```

P1: begin comment real-division program
B1: assert 0 ≤ c < d, e > 0
purpose |c/d - q| < e
purpose q ≤ c/d < q + 2·e, 2·e ≤ e
(q, s) := (0, 1)
loop L1: assert d·q ≤ c < d·(q + 2·s)
until s ≤ e/2
purpose q ≤ c/d < q + 2·s, 0 < e ≤ sL1
if d·(q + s) ≤ c then q := q + s fi
s := s/2
repeat
assert q ≤ c/d < q + 2·s, 2·s ≤ e
E1: assert |c/d - q| < e
end

```

(We have also changed the program's **purpose** statements to reflect reality.) It can be shown that a transformation such as $e \Rightarrow e/2$ preserves the relation between the program text and invariants, i.e. the transformed assertions are invariants of the transformed program.

3. MODIFICATION

Consider the following specification:

```

Q3: begin comment cube-root specification
assert a ≥ 0, e > 0
achieve |a1/3 - r| < e varying r
end

```

We would like to use the corrected real-division program as a basis for the construction of the specified program for computing cube-roots. (We assume, of course, that the cube-root operator is not primitive.) To this end, we first compare the specifications of the two programs. The output specification of the division program is

assert $|c/d - q| < e$,

while the output specification of the desired program is

achieve $|a^{1/3} - r| < e$ varying r .

The obvious analogy between the two is

$$\frac{q}{c/d} \iff \frac{r}{a^{1/3}},$$

i.e. where the former specification has q , the other has r , and where the former has c/d , the other has $a^{1/3}$. One way to obtain a cube-root program from the division program is via the transformations

$$\begin{array}{lll} q & \Rightarrow & r \\ u/v & \Rightarrow & u^{1/3} \\ c & \Rightarrow & a, \end{array}$$

where by $u/v \Rightarrow u^{1/3}$ we mean that every occurrence of the (general) division operator $/$ is replaced by the cube-root operator applied to what was the numerator. Transformations that involve specific functions such as division, are not, however, guaranteed to yield a correct program, since the program may be based on some property that holds for division, but not for extracting roots. Such transformations are heuristic in nature; they only suggest a possible analogy between the two programs. Indeed, when applied to the division program P_1 we get a program that computes a/d , not $a^{1/3}$. What must be done in such cases is to review the derivation

of the program, expressed by the programmer in purpose statements, and see where the analogy breaks down.

The purpose of the division program was $|c/d - q| < \epsilon$ which transformed into $|a^{1/3} - r| < \epsilon$ as desired. The programmer achieved $|c/d - q| < \epsilon$ by breaking it into the subgoals given in the statement

$$\text{purpose } q \leq c/d < q + 2 \cdot s, \quad 2 \cdot s \leq \epsilon,$$

part of which became the exit test for the loop and part became a loop invariant. These subgoals transform into

$$\text{purpose } r \leq a^{1/3} < r + 2 \cdot s, \quad 2 \cdot s \leq \epsilon,$$

which indeed imply the transformed goal $|a^{1/3} - r| < \epsilon$. The purpose of the loop body of the division program was

$$\text{purpose } q \leq c/d < q + 2 \cdot s, \quad 0 < s < s_{L_1}.$$

In other words, the loop body reaches the invariant while making progress towards the exit test by decreasing s . The loop-body subgoal of the transformed program, then, is

$$\text{purpose } r \leq a^{1/3} < r + 2 \cdot s, \quad 0 < s < s_{L_1}.$$

At this point the division program introduces a conditional with the

$$\text{purpose } q \leq c/d < q + s$$

and halves s .

It is here that the analogy breaks down. The division program achieves the above purpose in two cases, by testing if $d \cdot (q + s) \leq c$ or not. For example, if $d \cdot (q + s) \leq c$ does not hold, then $c/d < q + s$, as desired. On the other hand, the fact that $d \cdot (r + s) \leq a$ does not hold in the cube-root program tells nothing about $a^{1/3} < r + s$. We look, therefore, for a transformation that makes $d \cdot (r + s) > a$ imply $a^{1/3} < r + s$, or the equivalent to $a < (r + s)^3$. Matching what we have with what we want tells us that the implication would hold if we could transform $d \cdot (r + s) \Rightarrow (r + s)^3$. Thus, where the division program has the function $u \cdot v$, the cube-root program requires v^3 . We complete the analogy by adding the transformation

$$u \cdot v \Rightarrow v^3,$$

which is applied to the conditional test.

There remains one problem: a transformed program can only be expected to satisfy the output specification for those inputs that satisfy the transformed input specification. In our case, we can solve this if we can find an alternative manner by which to initialize the invariant $r \leq a^{1/3} < r + 2 \cdot s$ prior to entering the loop. To achieve the subgoal $r \leq a^{1/3}$, we can let $r = 0$. Then to achieve $a^{1/3} < r + 2 \cdot s = 2 \cdot s$, we can let $s = (a + 1)/2$. (This requires additional knowledge about cube-roots.) The complete cube-root program is:

```

Q3: begin comment real cube-root program
B3: assert a ≥ 0, c > 0
(r, s) := (0, (a + 1)/2)
loop L3: assert r ≤ a1/3 < r + 2 · s
until s ≤ ε/2
  if (r + s)3 ≤ a then r := r + s fi
  s := s/2
repeat
E3: assert |a1/3 - r| < ε
end

```

4. ABSTRACTION

At this point, we have two programs, P_1 for finding quotients and Q_3 for finding cube-roots. Both programs utilize the binary-search technique. It would be nice if one could extract an abstract version of the two programs that captures the essence of the technique, but is not specific to either problem. The resultant abstract program schema could be used as a model of binary search for the solution of future problems.

For this purpose, consider the complete analogy that we found between the specifications of P_1 and Q_3 :

$$\begin{array}{ll} q & \Longleftrightarrow r \\ u/v & \Longleftrightarrow u^{1/3} \\ c & \Longleftrightarrow a \\ u \cdot v & \Longleftrightarrow v^3 \end{array}$$

Since both u/v and $u^{1/3}$ are functions, we try to generalize them to an abstract function $\gamma(u, v)$. Similarly the generalization of $u \cdot v$ and v^3 is another function $\delta(u, v)$. Both q and r are output variables and are generalized to an abstract output variable z ; the input variables c and a are generalized to an abstract input variable x . This gives the following set of transformations for generalizing the division program:

$$\begin{array}{ll} q & \Rightarrow z \\ u/v & \Rightarrow \gamma(u, v) \\ c & \Rightarrow x \\ u \cdot v & \Rightarrow \delta(u, v) \end{array}$$

Applying these transformations to the specification

$$\text{achieve } |c/d - q| < \epsilon \text{ varying } q$$

of the division program yields

$$\text{achieve } |\gamma(x, d) - z| < \epsilon \text{ varying } z.$$

This will be the abstract output specification of the schema. Substituting the abstract functions γ and δ into their respective positions in the division program P_1 , does not, however, result in a schema that will work for all instantiations of γ and δ . This is because the original program relied upon facts specific to multiplication and division. We must therefore determine under what conditions the abstract schema does achieve its specifications.

To begin with, the transformed initialization assignment does not achieve the desired loop invariant. We therefore replace the loop initialization with the subgoal

$$\text{achieve } \delta(d, z) \leq x < \delta(d, z + 2 \cdot s) \text{ varying } z, s,$$

leaving—for the time being at least—the specifics of how to initialize the loop invariant unspecified. For the loop-body path to be correct, the truth of the invariant must imply that the invariant will hold next time around; this can easily be shown to be the case for any function δ . For the loop-exit path to be correct, we must have that the loop invariants, plus exit test, imply that the output invariant holds. For this to be the case, it suffices to establish the condition

$$\delta(w, u) \leq v \equiv u \leq \gamma(v, w).$$

In this manner, we have derived a general program schema for a binary search for the value of $\gamma(x, d)$ within a tolerance ϵ :

```

S4: begin comment binary-search schema
B4: assert  $c > 0$ ,  $\delta(w, u) \leq v \equiv u \leq \gamma(v, w)$ 
achieve  $\delta(d, z) \leq z < \delta(d, z + 2 \cdot c)$  varying  $z, c$ 
loop L4: assert  $\delta(d, z) \leq z < \delta(d, z + 2 \cdot c)$ 
until  $c \leq c/2$ 
  if  $\delta(d, z + c) \leq z$  then  $z := z + c$  fi
   $c := c/2$ 
repeat
E4: assert  $|\gamma(z, d) - z| < c$ 
end

```

Of course, for this schema to be executable, the function δ appearing in it must be primitive; otherwise, it should be replaced. Similarly, the unachieved subgoal

achieve $\delta(d, z) \leq z < \delta(d, z + 2 \cdot c)$ varying z, c

must be reduced to primitives.

5. INSTANTIATION

The binary-search schema just derived from the division program may be applied to the computation of the square root of an integer. The goal is to construct a program that finds the integer square-root z of a nonnegative integer a :

```

R5: begin comment integer square-root specification
assert  $a \in \mathbb{N}$ 
achieve  $z = \lfloor \sqrt{a} \rfloor$  varying  $z$ 
end

```

where the function $\lfloor u \rfloor$ yields the largest integer not greater than u .

We cannot directly match this goal with the output specification of the schema

assert $|\gamma(z, d) - z| < c$ varying z .

However, if we expand the goal $z = \lfloor \sqrt{a} \rfloor$, using the definition of $\lfloor u \rfloor$, we get the equivalent goal

achieve $z \leq \sqrt{a} < z + 1$, $z \in \mathbb{Z}$ varying z

(where $\mathbb{Z}$ is the set of all integers), i.e. z should be the largest integer not greater than $\sqrt{a}$. Since we know that the schema achieves the two output invariants

assert $z \leq \gamma(z, d) < z + c$,

we can compare these invariants with the above goal. This suggests the transformations

$$\begin{array}{lcl}
 \gamma(u, v) & \Rightarrow & \sqrt{u} \\
 z & \Rightarrow & a \\
 c & \Rightarrow & 1
 \end{array}$$

to achieve $z \leq \sqrt{a} < z + 1$. In addition, we will have to extend the program to ensure that the final value of z is a nonnegative integer.

The precondition for the schema's correctness is

assert $c > 0$, $\delta(w, u) \leq v \equiv u \leq \gamma(v, w)$;

instantiating it yields

assert $1 > 0$, $\delta(w, u) \leq v \equiv u \leq \sqrt{v}$.

This condition may be satisfied by taking $\delta(w, u)$ to be u^2 . This completes the analogy, and suggests the additional transformation

$$\delta(w, u) \Rightarrow u^2.$$

Applying the instantiation mapping to the schema, we obtain the partially written program:

```

R5: begin comment incomplete integer square-root program
B5: assert  $a \in \mathbb{N}$ 
achieve  $z^2 \leq a < (z + 2 \cdot c)^2$  varying  $z, c$ 
loop L5: assert  $z^2 \leq a < (z + 2 \cdot c)^2$ 
until  $c \leq 1/2$ 
  if  $(z + c)^2 \leq a$  then  $z := z + c$  fi
   $c := c/2$ 
repeat
assert  $|\sqrt{a} - z| < 1$ 
achieve  $z \in \mathbb{Z}$  protecting  $z \leq \sqrt{a} < z + 1$  varying  $z$ 
end

```

This program still contains two unachieved subgoals. The first can be achieved by assigning $(z, c) := (0, (a+1)/2)$. For the second, we may perturb the current value of z just enough to make it an integer. (The protecting clause means that the relation $z \leq \sqrt{a} < z + 1$, achieved by the instantiated schema, should not be clobbered when achieving the additional goal $z \in \mathbb{Z}$.) This can be done by assigning

if $\lceil z \rceil^2 \leq a$ then $z := \lceil z \rceil$ else $z := \lfloor z \rfloor$ fi.

An alternative approach to completing the above program is to insist that $z \in \mathbb{N}$ hold *throughout* execution of R_5 . This is the avenue pursued, for example, in the "structured programming" derivations in [Dijkstra76] and [Blikle78].

6. DISCUSSION

There are a few problems inherent in the use of analogies for program modification and abstraction. These include "hidden" analogies, "misleading" analogies, "incomplete" analogies, and "overzealous" analogies. Hidden analogies arise when given specifications (of the existing program and desired problem in the case of modification, and of the two or more existing programs in the case of abstraction) that are to be compared with one another have little syntactically in common. Since the pattern-matching ideas that we have employed are syntax-based, when the specifications are not syntactically similar, the underlying analogy would be hidden. In such a situation it is necessary to rephrase the specifications in some equivalent manner that brings their similarity out, before an analogy can be found. This is clearly a difficult problem in its own right; in general some form of "means-end" analysis seems appropriate.

At the opposite extreme, a syntactic analogy may be misleading. The same symbol may appear in the specifications of two programs, yet may play nonanalogous roles in the two programs. Two programs might even have the exact same specifications, but employ totally different methods of solution. Situations such as these would be detected in the course of analyzing the correctness conditions for the abstracted programs.

Knowing how a program was constructed can help avoid overzealously applying transformations to unrelated parts of a program. We have seen how the program derivation also helps complete an analogy between two programs, only part of which was found by a comparison of specifications.

REFERENCES

- [Blikle78] Blikle, A., "Towards mathematical structured programming," pp. 9.1-9.19 in *Formal Descriptions of Programming Concepts*, ed. E. J. Neuhold, North-Holland (1978).
- [Brown76] Brown, R., *Reasoning by analogy*, Artificial Intelligence Laboratory MIT, Cambridge, MA (Oct. 1976).
- [Dershowitz81] Dershowitz, N., "The evolution of programs: Program abstraction and instantiation," *Proc. 5th Intl. Conf. on Software Engineering*, pp. 79-88 (Mar. 1981).
- [DershowitzManna75] Dershowitz, N. and Manna, Z., "On automating structured programming," *Colloques IRIA on Proving and Improving Programs*, pp. 167-193 (July 1975).
- [DershowitzManna77] Dershowitz, N. and Manna, Z., "The evolution of programs: Automatic program modification," *IEEE Trans. Software Engineering* SE-3(6), pp. 377-385 (Nov. 1977).
- [DershowitzManna81] Dershowitz, N. and Manna, Z., "Inference rules for program annotation," *IEEE Trans. Software Engineering* SE-7(2), pp. 207-222 (Mar. 1981).
- [Dijkstra76] Dijkstra, E. W., *A discipline of programming*, Prentice Hall, Englewood Cliffs, NJ (1976).
- [FikesHartNilsson72] Fikes, R. E., Hart, P. E., and Nilsson, N. J., "Learning and executing generalized robot plans," *Artificial Intelligence* 3(4), pp. 251-288 (Winter 1972).
- [Floyd71] Floyd, R. W., "Toward interactive design of correct programs," *Proc. Information Processing Cong.*, pp. 7-10 (Aug. 1971).
- [Gerhart75] Gerhart, S. L., "Knowledge about programs: A model and case study," *Proc. Intl. Conf. on Reliable Software*, pp. 88-95 (Apr. 1975).
- [KatzManna75] Katz, S. M. and Manna, Z., "Towards automatic debugging of programs," *Proc. Intl. Conf. on Reliable Software*, pp. 143-155 (Apr. 1975).
- [Kling71] Kling, R. E., *Reasoning by analogy with applications to heuristic problem solving: A case study*, Ph.D. Dissertation, Stanford Univ., Stanford, CA (Aug. 1971).
- [MannaWaldinger75] Manna, Z. and Waldinger, R. J., "Knowledge and reasoning in program synthesis," *Artificial Intelligence* 8(2), pp. 175-208 (Summer 1975).
- [McDermott79] McDermott, J., "Learning to use analogies," *Proc. Intl. Joint Conf. on Artificial Intelligence*, pp. 568-576 (Aug. 1979).
- [Sagiv76] Sagiv, Y., *A study of the automatic debugging of programs*, Master's thesis, Weizmann Institute of Science, Rehovot, Israel (Aug. 1976).
- [Sussman75] Sussman, G. J., *A computer model of skill acquisition*, American Elsevier, New York, NY (1975).
- [UlrichMoll77] Ulrich, J. W. and Moll, R., "Program synthesis by analogy," *Proc. ACM Symp. on Artificial Intelligence and Programming Languages*, pp. 22-28 (Aug. 1977).
- [Winston80] Winston, P. H., "Learning and reasoning by analogy," *Comm. ACM* 23(12), pp. 689-703 (Dec. 1980).

REASONING BY ANALOGY IN SCIENTIFIC THEORY CONSTRUCTION

Lindley Darden
Department of Philosophy
University of Maryland
College Park, Maryland 20742

ABSTRACT

This paper discusses methods of reasoning for constructing new concepts and theories in science. The methods include reasoning by analogy, reasoning from failed analogy, reasoning piecemeal from more than one analogy, and reasoning using a shared abstraction. Cases from the history of science illustrate the methods, including a lengthy discussion of selection theories. Suggestions for implementation in AI systems are made.

KEY WORDS: analogy, theory construction, discovery, scientific reasoning, selection theories, artificial intelligence.

1. INTRODUCTION

As a historian and philosopher of science, I am interested in the process of the construction of new scientific theories. This paper explores methods of reasoning to new concepts and new theories in science; the methods of reasoning include reasoning by analogy and other patterns of reasoning similar to analogical reasoning. Suggestions are made as to how these methods could be implemented in artificially intelligent computer systems.

It is possible to reason to new concepts using only the data to be explained. For example, BACON (Bradshaw, Langley and Simon, 1980) postulates a new intrinsic property of an object by using data about how that object interacts with a number of different, other objects. Uniformity of behavior is postulated as due to an intrinsic property. Thus, data about relations is decomposed and used as an indication of the existence of a new property. Usually, however, sources other than the data themselves are needed to supply new concepts. Analogies are one source of such new concepts that can be built into new theories.

Philosophers of science have rarely discussed discovery in science. A prevailing view has been that the discovery of new theories is a matter of the individual psychology of the scientists, that there is no "logic of discovery." Karl Popper (1960) is a prominent proponent of this view and Carl Hempel (1966, p.14) argued that there can be no mechanical rules for producing the novel concepts found in theories. Although N. R. Hanson (1961) attempted to provide a logic of discovery, which he called "retroductive reasoning," he merely hinted at the way plausible hypotheses actually could be constructed. I will pursue two of Hanson's hints in this paper: reasoning by analogy and reasoning about types of theories.

The existence of an infallible logic of discovery is unlikely. Instead, a goal of "friends of discovery" (Nickles, 1980) should be to find patterns of reasoning that can produce plausible hypotheses. The plausible hypotheses can then be tested to see if they qualify as genuine "discoveries."

If we are going to understand how analogies function in patterns of reasoning in theory construction, a series of questions need to be answered.

These questions are:

1. What is an analogy?
2. Are there types of analogies?
3. How do we find analogies?
4. How do we distinguish good from bad analogies?
5. How do we use analogies to construct new scientific concepts and theories?
6. How can we test a proposed pattern of reasoning by analogy to see if it is adequate for constructing plausible hypotheses?

I will not be able to completely answer all of these questions in this paper, but I want to begin the task. Work in both philosophy of science and AI is relevant. Mary Hesse is the philosopher who has dealt most extensively with some of these questions. Most of the AI work that has dealt with discovery in science, such as BACON, DENDRAL, and METADENDRAL (Buchanan and Feigenbaum, 1978), has not used reasoning by analogy. Conversely, most of the AI work on reasoning by analogy, such as Evans (1968), Kling (1971), Genesereth (1980), Carbonelle (1982), has not been directed to understanding

scientific discovery. The section of Winston's (1980) work on the analogies between water flow and electricity brought together scientific reasoning and reasoning by analogy, though Winston's attention was on teaching and learning rather than on discovery.

I will draw on this literature, as well as ideas of my own, to devise four methods of theory construction. Each method will be illustrated with examples from the history of science. Prospects for implementation of the patterns will be discussed. If methods of reasoning in theory construction can be implemented in AI systems, then it would be possible to test the patterns to see if they, indeed, can function to yield plausible hypotheses. Philosophy of science and AI have the potential of being joined to produce an experimental philosophy of science.

2. METHODS OF THEORY CONSTRUCTION

In her book Models and Analogies in Science (1966), Mary Hesse discusses reasoning by analogy in theory construction. Hesse's key points are best illustrated by one of her examples. She compares sound waves and light waves. The properties which are similar she designates as the positive analogy; the dissimilar properties constitute the negative analogy; the neutral analogy involves the properties of the analogue which may or may not be present in the subject field. In her example, the properties of light are the subject of investigation. Light has properties that are similar to properties of sound waves and the neutral analogy of a medium for the waves to travel in provided a plausible, but subsequently disproved, hypothesis about the existence of the ether. The horizontal relations between the analogue and the subject are usually those of similarity, although occasionally specific properties may be identical. The vertical relations among properties of the analogue should be, according to Hesse, causal relations, in the weak sense of a "tendency to co-occurrence." (Hesse, 1966, p.77). Hesse indicated that the analogue may be important, not only in the original stages of theory construction, but also later: if the theory faces anomalies, the neutral analogy may be exploitable for further theory construction.

Hesse's representation scheme of lining up properties of the analogue and subject is much like frame-structured representation schemas used in AI systems. (Minsky, 1975; Thagard, forthcoming) One could construct a frame for sound waves with slots for the properties of sound waves, such as travels in a medium, and values for the slots, with the slot "medium" having the value "air." Another frame could be constructed for light. Hesse's insistence that the properties of the analogue must be causally connected is directed at a key problem with analogical reasoning: why, if some features of an analogue and a subject are similar, would we expect others to be? In other words, how can we

confidently use the neutral analogy to postulate plausible features in the subject? By restricting the features to those that have a "tendency to co-occurrence," Hesse attempts to solve this problem. Certainly if tight causal connections exist between features, then the likelihood that the neutral analogy can be transferred is increased. Unfortunately, Hesse's form of representation has the disadvantage that it does not make clear what the causal interconnections are between which properties. If one puts Hesse's method into a frame representation, then one could add slots to the frames to indicate which properties are causally connected and what the relations are. Winston's semantic net form of representation has the advantage of clearly showing the causal connections by representing entities as nodes and relations as arcs between the nodes.

Once such a causal network has been imposed on the representation of the analogue, then the implementation of reasoning by analogy is very straight-forward: match and transfer. But this process looks very canned and does not seem to resemble discovery. All the creative work went into choosing the analogue and representing the causal interconnections. The key questions become: how do we find good analogues and how do we focus on certain features of an analogue as relevant to the subject at hand?

An even deeper problem about the use of analogies in discovery exists. If discovery involves finding a similar analogue, how is anything new ever discovered? One seems to be employing the same structures again and again. We need some way of getting new structures, not merely reusing the old ones.

Hesse does not tell us what we are to do when postulated causal connections fail to be empirically confirmed, such as her own example of the postulation of the ether as a medium for light waves. The physicist Robert Oppenheimer, in an insightful paper of 1956, that Hesse does not cite in her 1966 book, discussed the role of analogies in discovering new theories. Analyzing the stages in the construction of the wave theory of light, Oppenheimer proposed that scientists came to the study of light phenomena with past experience of the nature of waves. But when the postulated medium for light waves was not found, it forced scientists to revise their previous view of necessary connections, to sever the notion of a wave from that of a medium. Thus, a portion of the causal network failed to transfer. This failure forced scientists to a creative new concept. But, humans came to this new concept of a mediumless wave with reluctance. It will be difficult to devise criteria for a system to use to enable it to choose between a failed analogy that should be discarded and a failed analogy that can be used to sever causal connections previously thought to be necessary.

From Hesse and Oppenheimer we can extract two methods of theory construction by analogy:

look for a relevantly similar analogue and exploit the neutral analogy; if this process fails, consider severing connections between properties that were previously thought to be necessary connections. But these methods are limited. A relevantly similar analogue may not exist. Also, the severing of what we thought were causally correlated properties is not a kind of reasoning that occurs frequently in science. Thus, these methods will have limited scope.

3. PIECEMEAL THEORY CONSTRUCTION

Little discussed in the analogy literature is the possibility of using more than one analogue, in a piecemeal or modular way, to construct a new theory out of old, but previously unrelated, pieces. Piecemeal theory construction using more than one analogy gives meaning to the cliché that creativity may result from putting old ideas together in new ways.

Charles Darwin's theory of natural selection illustrates theory construction by piecemeal relations. There are two conditions required for natural selection to occur. First, organisms must have hereditary variations. Secondly, more organisms must be produced than can survive. Coupling these two conditions gives natural selection: those organisms with variations advantageous in the struggle for existence will tend to survive and reproduce.

Darwin had two different sources for the two different conditions that make up his theory. Malthus's ideas about overpopulation in humans served as the source of Darwin's idea of a struggle for existence. The relation between Malthus's ideas and Darwin's is actually stronger than analogical: Darwin generalized to all organisms what Malthus proposed for humans. Thus, the relation is generalization and specialization rather than analogical.

The other source for Darwin's theory was the analogy to artificial breeding which had two functions. First, it provided evidence of the existence of heritable variations. Darwin argued that both domesticated organisms and organisms in the wild had numerous individual differences, many of which were passed on to offspring. Secondly, artificial breeding showed that selection of some of the hereditary variants and elimination of others could result in the formation of new varieties that perpetuated themselves. By analogy, natural selection could be expected to do likewise. But the agent of selection in nature had to be different from the human agents in artificial selection. The Malthusian idea, generalized, supplied the missing piece: the struggle for existence in nature served to select some variants and eliminate others. Darwin's theory thus had piecemeal relations to two other areas: Malthus's ideas and artificial selection.

Another case of the piecemeal construction

of a new concept is the discovery of viruses. This case differs from those previously discussed since the discovery is of a new entity at a new level of organization, not the construction of a new theory. Categories available in medical microbiology by the end of the nineteenth century were that of microorganism or chemical toxin. Microorganisms were visible in the light microscope, didn't pass through filters, were infectious, and grew on culture media. Chemical toxins were submicroscopic, were filterable, didn't multiply in the affected organisms, and didn't grow on media. When filterable, submicroscopic but infectious agents were discovered, they presented a real puzzle. Those favoring a microbe explanation suggested that these were just smaller than previously discovered microbes and, with suitable work, a medium could be found on which to grow them. Those favoring a chemical toxin explanation suggested that the agents weren't infectious, but were toxic in very small quantities, and thus it only appeared that they multiplied in the infected organism. Braver souls were willing to postulate a new entity, midway between a microorganism and a chemical substance, that was filterable, submicroscopic and infectious. Thus, a new concept was constructed piecemeal, in between two levels of organization. Subsequent work changed the essential characterization of the virus from a filterable infectious agent to an entity with structure visible in the electron microscope which is unable to multiply outside of other living cells. (Hughes, 1977).

The pattern of reasoning provided by this case is that of transfer of properties piecemeal between two levels of organization to create a new concept that shares some properties from each. After the new concept is established, then new properties of its own may be added as new data emerges. This pattern of reasoning has a similar structure to reasoning piecemeal from two different analogues; however, it is more constrained since the sources of the transferred properties must be related in a hierarchy of levels of organization. A new concept constructed by this interlevel, piecemeal analysis may be more plausible than one constructed by analogy, since concepts at new levels of organization have often been found, e.g., cells, enzymes, macromolecules

4. TYPES OF THEORIES

Thus far, we have been considering an analogy as a relation between two similar analogues. An alternative analysis of what an analogy is provides us with a fourth method of theory construction. Michael Genesereth has explicitly stated an alternative analysis of analogy which has been implicit in some AI work. According to Genesereth, "many analogies are best understood as statements that the situations being compared share a common abstraction." (Genesereth, 1980, p.208). Mary

Hesse, in addition to characterizing an analogy as a direct mapping between a subject and its analogue based on the similar properties, also mentioned formal analogies in which two things share only the same uninterpreted mathematical formalism. Genesereth's abstractions are not merely Hesse's formal analogies, since Genesereth's abstractions may have more semantic content than an uninterpreted formalism. He gave an example: "when one asserts that the organization chart of a corporation is like a tree or like the taxonomy of animals in biology, what he is saying is that they are all hierarchies. With this view, the problem of understanding an analogy becomes one of recognizing the shared abstraction." (Genesereth, 1980, p.208). (Prior AI discussion of analogy as a shared abstraction can be found in Kling, 1971; Kurt and Brown, 1979; and Winston, 1980)

Genesereth shares a view explicitly discussed by the psychologist Dedre Gentner in an excellent paper entitled "The Structure of Analogical Models in Science," (1980) in which she argued that the relations between the analogue and the subject are those of identity rather than similarity, but identity at an abstract level. When Hesse mentioned that in some cases the relations between the directly mapped properties were those of identity rather than similarity, those were identity relations at the specific, not the abstract, level.

The system that Genesereth envisions has various abstractions built into it, as well as criteria for choosing and applying an abstraction in a particular problem situation. Thus, the idea of analogy as shared abstraction supplies another method of theory construction: Explicitly devise abstractions of theories or other things that might serve as analogues in theory construction. In the first stages of theory construction, search through the set of abstractions to see if an appropriate one (according to some criterion) is found. If so, then instantiate it for the case in hand. If a single abstraction is inadequate, then try to use more than one piecemeal.

Before considering scientific examples of abstractions of theories, it is instructive to compare the direct mapping method that results from comparing two analogues and the shared abstraction method. Although humans can spontaneously see relations between two analogues, for an AI system to be able to do direct mapping, the two analogues must be represented in some way. The representation, such as a frame system suggested above for waves, already has an abstract form built into it. Thus, the analogical reasoning that the system is able to do seems to be a result of building a shared abstraction into the representation scheme. Consequently, the direct mapping method is not as different from the shared abstraction method as at first it seemed to be. The step of taking a concrete analogue and constructing a representation of it for use

in the analogical reasoning becomes a key component in the direct mapping method. It is difficult to see how an AI system, rather than the designer of the system, could carry out this abstracting step.

Theory construction involves not only the original construction of a theory, but also its subsequent development in the light of new evidence and anomalies. The direct mapping method holds the promise of being able to aid in anomaly resolution, since the analogue may contain negative or neutral components that were not used in the original theory construction. When anomalies arise, these additional properties may be used for further theory construction. If one is working only from the abstraction, which contains only the shared properties, then this additional source of information is lost. The direct mapping and shared abstraction methods could be combined to use the best features of each. Build a system that has both the detailed analogues and the shared abstractions represented. If the abstraction does not contain sufficient information to resolve an anomaly, then conduct the search at the more detailed level by searching through the negative or neutral properties in the detailed analogue to see if any can be of use.

5. ABSTRACT FORM OF SELECTION THEORIES

Although use of analogies and use of formal mathematical models are common methods in science, explicit use of semantically rich abstractions is not. Thus, case studies for the use of abstractions are lacking. But it is very interesting to ask whether abstractions could be devised and to look at the ways they map onto actual scientific theories. In devising an AI system to do theory construction, a key question becomes whether this method could have been used to produce known scientific theories, even though the discoverers did not use it.

Once Darwin had constructed the theory of natural selection, he established a new type of scientific theory. We may consider one way of abstracting the components of selection theories:

- I. An array of variants is generated.
 1. Number of variants
 2. Number of types of variants.
 3. Amount of difference between variants.
 4. Mechanism of generation of variants.
- II. Selection of a subset of variants occurs.
 1. Agent of selection
 2. Criteria of selection
- III. After selection, the pool of variants is different.

Each of these abstract components can be instantiated in different ways. Some of the

different possible instantiations yield alternative evolutionary theories that have actually been proposed historically.

For (I), the generator or the cause of the variants will determine the number of variants, the number of different types of variants, how much each variant differs from another. Different evolutionary theories have differed in their claims about such properties of the variants. Darwin proposed numerous small variations, which he called "individual differences," as the most important type of variation that natural selection acts on. Hugo de Vries, in his *Mutation Theory* (1903, 1904), proposed an alternative to Darwinian natural selection by proposing that large scale mutations occurred that could give rise to a new species in a single generation. Thus, the selection step played a minor role in de Vries's theory, serving merely to eliminate the least advantageous species. Mendelian genetics did not support de Vries's claim for the existence of such large scale mutations.

Additional alternatives to small variants are still being proposed. Stephen Gould (1977) recently suggested that large scale changes in developmental timing may be more important in evolutionary change than small scale point mutations. According to Gould, humans have many properties of baby apes: maybe a large scale change kept our ancestors from reaching ape maturity. Selection would still operate on such developmental changes. Thus, these views of Gould's differ from Darwin's with respect to (I), the generation of the variants.

There is a temporal relation between steps I, II and III. If step I yields certain types of variants, then step II may be unnecessary. For example, if the generator of variants had the capacity to produce only adapted variants, then the subsequent step of selection would not be necessary in either domestic or natural selection. If the second step is not needed, then the theory proposed is not a selection theory. Compare, for instance, so-called Lamarckian evolution, (Lamarck's theory was actually more complex): characters that are acquired as a result of an adaptation to an environment during the life of an individual are passed on to offspring. In such a theory, the generator of variants eliminates the need for selection since the criterion of selection (adaptation) has been incorporated into the mechanism of generation. Mendelian genetics has provided no evidence for such a mechanism for the inheritance of acquired characters.

Although it failed, such a so-called Lamarckian theory represents, nonetheless, an alternative type of theory to the selection type: a good analogy to human activity would be not domestic selection but human tool-making. Fashioning a tool to fit a need is analogous to an organism developing a

characteristic in response to an environmental demand. The criteria that would have operated in the selection process have been incorporated into the mechanism of generation of variants; the need for the selective step is eliminated. Thus, when faced with a problem of explaining the origin of new adapted somethings, either of two abstractions can be evoked: the selection type or the tool-fashioning type.

(Note at a metalevel: we have also found a way of obtaining a new abstraction: certain instantiations of parts of one abstraction may produce a new type of theory which obviates the need for the rest of the abstraction to be instantiated. A new abstraction with the tool-fashioning nature of the generator of the variants can be constructed; the new abstraction is no longer a selection theory abstraction.)

For step II, the selection mechanism has subproperties which include the agent of selection (that sounds a bit anthropomorphic; perhaps a more neutral term can be found) and the criteria on the basis of which some variants are selected. Contemporary controversies about evolutionary mechanisms between neutralists (see King and Jukes, 1969) and selectionists have focused on whether variants can become fixed in a population as a result of random processes rather than selection of adapted variants. Though random drift results in the pool of variants differing in succeeding generations, that pool does not share a set of characteristics as a result of a selective process. Thus, if there is no criterion for choosing among the variants and those that survive do so because of random processes, then the theory is not a selection theory. This case shows us that Step II could be made more abstract to read: a subset of the variants are retained. Selection or random choice among the variants would be alternative instantiations of this more abstract Step II.

(Note at a metalevel: once again we can construct a new abstraction by varying a step of the original abstraction. This time step II is changed and made even more abstract.)

Step III, a different array of variants, is the product of a causal mechanism operating to produce a different state of affairs at a subsequent time. In domestic and natural selection, the most important temporal dimension is the next generation of organisms; thus variability that is passed from parent to offspring, i.e., inherited variability, is the only variability important for selection. But in the abstraction of selection theories, no requirement for different generations is included. I have chosen to express the components sufficiently abstractly so that they apply to additional theories that do not include different generations of organisms but only a different pool of variants at a later time. The abstraction could be formulated at a lower level

and include transmission of selected characteristics to a subsequent generation; but fewer theories would be instances of the lower level abstraction. Another level of complexity is introduced in the biological case, since the variants can be considered either genes or phenotypes. Controversies about the level of selection (Lewontin, 1970; Hull, 1980) focus on this question of what is to count as the variant. Hull suggested a distinction between the "replicator" and the "interactor" in biological evolution; the abstraction discussed here suggests that "variator" should also be a key abstract component.

If steps I-III, considered as a temporal process, begin a second time, with the pool of variants (or their descendants) produced by the previous run as the starting point, then selection continues in the same direction. If the criterion of selection changes (as in environmental changes), then the direction shifts. Since the generator of variants and the criterion of selection can change independently in some instantiations, the range of variants produced is potentially very large.

In passing it is worth noting that John Holland (1975) constructed an abstract form of adaptive theories that differs from my abstract form of selection theories. Holland constructed his abstraction by drawing upon the contemporary theory of evolution. Its components include: the environment of the system undergoing adaptation, the adaptive plan "which determines successive structural modifications in response to the environment," and "a measure of the performance of different structures in the environment." (Holland, 1975, p.20) He has much more focus on the environment that my abstraction does; the environment becomes the agent and provides the criterion of selection within my abstraction. It is difficult to map my generator of variants into his scheme; the adaptive plan must contain the generator. Holland's abstraction may or may not be instantiated by a selective step; thus the adaptive theories which he is characterizing may or may not be selection theories. In fact, his abstraction also characterizes the type of theories I called "tool-fashioning": the adaptive plan has a generator that can fashion variants in response to the environmental demand; no selective step is necessary. Conversely, my abstraction may or may not be instantiated to result in adaptive theories, depending on the criterion of selection. It is interesting to note that two people, both drawing on the theory of evolution, construct markedly different abstractions. Could we devise an AI system that could construct different abstractions from the same particular theory?

6. EXAMPLES OF SELECTION THEORIES

A surprising number of different theories are specializations of the abstract type of selection theory given in steps I-III. A biological theory that fits this type beautifully is Jerne's (1955) natural selection theory of antibody formation, subsequently modified by Burnet (1957) to the theory of clonal selection for the production of antibodies. The problem is to explain how the body forms antibodies that are able to deactivate large numbers of invading foreign substances, called antigens, while not attacking the body's own substances. Jerne, in reflecting on the reasoning he used in the formation of the theory, said: "three mechanisms must be assumed: (1) a random mechanism for ensuring the limited synthesis of antibody molecules possessing all possible combining sites, in the absence of antigen, (2) a purging mechanism for repressing the synthesis of such antibody molecules that happen to fit auto-antigens, and (3) a selective mechanism for promoting the synthesis of those antibody molecules that make the best fit to any antigen entering the animal." (Jerne, 1966, p.301). Jerne's theory may be expressed in our steps I-III. Jerne proposed (I) a mechanism for producing an array of antibodies, (II) both negative selection to eliminate antibodies against bodily substances and positive selection for production of antibodies that fit an invading antigen, with the (III) result that the original array of antibodies is altered after selection. Jerne's theory proposed that natural antibodies circulating through-out the body would attach to an antigen, carry it to a phagocytic cell and cause that cell to produce more antibodies like the selected one. (Jerne, 1955, p.849).

Although Burnet endorsed much of Jerne's theory, he objected to the mechanism of production of antibodies and proposed an alternative at the level of the cell rather than the molecule. Burnet said: the "major objection is the absence of any precedent for, and the intrinsic unlikelihood of the suggestion, that a molecule of partially denatured antibody could stimulate a cell into which it had been taken to produce a series of replicas of the molecule... It would be more satisfactory if the replicating elements essential to any such theory were cellular in character *ab initio* rather than extracellular protein which can replicate only when taken into an appropriate cell... [this idea is developed here] from what might be called the 'clonal' point of view." (Burnet, 1957, p.67). Burnet proposed that the mechanism of variation was somatic mutation, namely changes within cells of the body. The selective mechanism operated by molecules on the cell surface recognizing antigens. After a type of cell recognized an antigen, it would be stimulated to reproduce into a clone of its type. Of this theory Burnet said: "Such a point of view is basically an attempt to apply the concept of population genetics to the clones of mesenchymal

cells within the body." (Burnet, 1957, p.68). Thus, Burnet explicitly appealed to the analogy between mutation with natural selection and his theory of somatic mutation and clonal selection.

Burnet argued against a prior theory of antibody formation that was of the tool-fashioning type. (Burnet, 1957, p.67). The template theory of antibody formation proposed that the body fashions an antibody on the invading antigen which serves as the template. The Jerne-Burnet selection theory, not the template theory, was subsequently confirmed and expanded. (Golub, 1981).

A number of other theories can be seen as selection theories. Additional historical research will be required to determine whether direct mapping to Darwin's theory of natural selection, use of the abstract type of selection theories, or other methods of theory construction played a historical role in the construction of these theories. Here I will merely call attention to the fact that their structure fits that of the abstraction for selection theories.

Darwin in The Power of Movement in Plants (1881) explained, for example, the coiling of a pea tendril around a wire as a result of natural complete circular movement of tendril being altered (selected) after encountering an object, so that subsequent movement was in the direction of the wire. The variant pool was complete circular movement, the object encountered provided the selective agent in that direction, and the altered pool was the subsequent movement only in that direction. (Ghiselin, 1969, p.196).

Operant conditioning works by selectively reinforcing some behaviors out of an original array to alter the subsequent behavior. The rat in a maze, for example, engages in random movements. Those in the direction of a food bar are reinforced, until the rat learns to move non-randomly. The original array of variants is thus random movement. Humans are the agents of selection, in the experimental case, and the criterion for selection is movement toward the food bar. The altered array of variants is the movement after the reinforcement. Memory plays the role of heredity in supplying the altered pool of variants on which subsequent selection steps act.

Various attempts to apply selection theories to social phenomena have been made as far back as social Darwinism in the nineteenth century and as recently as sociobiology in the last few years. Whether the mechanism for producing and transmitting types of social behavior is genetic or environmental has been hotly debated.

A final type of selection theory is evolutionary epistemology. Stephen Toulmin in Human Understanding argued for a view of the

development of knowledge as a result of "a dual process of conceptual variation and intellectual selection." (1972, p.200). Neither Toulmin nor others within this epistemological tradition have had much to say about the mechanism for the production of new concepts. I have been arguing that use of analogical reasoning may provide a means of constrained generation of plausible new ideas that then may be selected according to the criteria scientists use for justifying theories. Whether selection-type theories or tool-fashioning theories will ultimately be of use in understanding the growth of knowledge remains to be seen.

Recursive self-reference is proving to be a powerful technique in artificial intelligence. Having now come full-circle to my starting point, namely a discussion of the way analogies may be used as a mechanism to generate new ideas, it is no doubt time to stop.

7. CONCLUSION

In conclusion, I would like to suggest that AI and philosophy of science can work together in coming to understand the heuristics in discovery or patterns of reasoning in theory construction. AI is sometimes called experimental epistemology, and I believe it holds the promise of making philosophy of science an experimental discipline. If a pattern of reasoning can be built into an AI system that can construct plausible hypotheses, then that would serve as an experimental test of the pattern. Philosophers will have to be much more precise and detailed in their specifications of patterns of reasoning if those patterns are to be implemented in AI systems.

Of the methods of theory construction that I have discussed, the one that will probably be easiest to implement is theory construction by shared abstraction. Philosophers of science and people in AI could cooperate in finding types of theories, such as the selection type and tool-fashioning type that I have mentioned. Also, they could work together in developing criteria for when a particular type is to be evoked and instantiated in a particular problem domain.

The method that holds the greatest promise for producing something new or creative is the piecemeal construction using more than one analogue. New ideas often result from putting old ideas together in new ways. The implementation of piecemeal theory construction seems harder to me, but potentially very exciting.

On this optimistic note for future interfield interactions, let me close.

ACKNOWLEDGMENTS

I would like to thank students in my seminars on analogical reasoning at the Computer Science Department at Stanford and the Philosophy Department at Maryland for their many helpful comments on the ideas in this paper. I especially thank Bruce Buchanan, Russell Greiner, Lars Rodseth, Richard Keller, David Kohn, Louis Steinberg, Marcia Kraft, and Frederick Suppe for specific comments and suggestions. I appreciate the time and space provided for research and writing over the last two years by sabbatical leave and a General Research Board Grant from University of Maryland, a visiting scholar position with the Heuristic Programming Project at Stanford, a guest account on the SUMEX-AIM computing facility, a research fellowship from the American Council of Learned Societies funded by the National Endowment for the Humanities, and a visiting scholar position in the Department of History of Science at Harvard.

REFERENCES

- Bradshaw, Gary, Pat Langley, and Herbert Simon. (1980). "BACON.4: The Discovery of Intrinsic Properties." CIP Working Paper No. 420. Department of Psychology. Carnegie-Mellon University.
- Buchanan, Bruce, and Edward Feigenbaum. (1978). "DENDRAL and METADENDRAL." Artificial Intelligence 11:5-24.
- Burnet, F.M. (1957). "A Modification of Jerne's Theory of Antibody Production using the Concept of Clonal Selection." The Australian Journal of Science 20: 67-69.
- Carbonell, Jaime. (1982). "Learning by Analogy: Formulating and Generalizing Plans from Past Experience. In Machine Learning." Edited by R. Michalski, J. Carbonell, and T. Mitchell. Palo Alto, California: Tioga Publishing Co.
- Darden, Lindley. (1976). "Reasoning in Scientific Change: Charles Darwin, Hugo de Vries, and the Discovery of Segregation." Studies in the History and Philosophy of Science 7:127-169.
- Darden, Lindley and Nancy Maull. (1977). "Interfield Theories." Philosophy of Science 44: 43-64.
- Darden, Lindley. (1978). "Discoveries and the Emergence of New Fields in Science." In PSA, 1978, v. 1. Edited by P.D. Asquith and I. Hacking. East Lansing, Michigan: Philosophy of Science Association. Pages 149-160.
- Darden, Lindley. (1980). "Theory Construction in Genetics." In Scientific Discovery: Case Studies. Edited by T. Nickles. Dordrecht, Holland: Reidel. Pages 151-170.
- Darwin, Charles. (1859). The Origin of Species. Reprint of First Edition, Baltimore, Maryland: Penguin, 1968.
- Darwin, Charles. (1876). The Autobiography of Charles Darwin and Selected Letters. Francis Darwin. New York: Dover, 1958.
- Darwin, Charles. (1881). The Power of Movement in Plants. Reprinted: New York: De Capo Press, 1966.
- Evans, Thomas G. (1968). "A Program for the Solution of a Class of Geometric Analogy Intelligence Test Questions." In Semantic Information Processing. Edited by M.I. Minsky. Cambridge, Mass.: MIT Press.
- Genesereth, Michael R. (1980). "Metaphors and Models." In Proceedings of the First Annual National Conference on Artificial Intelligence. Menlo Park, California: American Association of Artificial Intelligence. Pages 208-211.
- Gentner, Dedre. (1980). "The Structure of Analogical Models in Science." Cambridge, Mass: Bolt Beranek and Newman, Inc. Report No. 4451.
- Ghiselin, Michael T. (1969). The Triumph of the Darwinian Method. Berkeley: University of California Press.
- Golub, Edward S. (1981). The Cellular Basis of the Immune Response. 2nd ed. Sunderland, Mass.: Sinauer Associates, Inc.
- Gould, Stephen J. (1977). Ontogeny and Phylogeny. Cambridge, Mass.: Harvard University Press.
- Hanson, Norwood R. (1961). "Is There a Logic of Scientific Discovery?" In Current Issues in the Philosophy of Science. Edited by H. Feigl and G. Maxwell. New York: Holt, Rinehart and Winston, 1961. Reprinted in Readings in the Philosophy of Science. Edited by B. Brody. Englewood Cliffs, New Jersey: Prentice-Hall, 1970. Pages 620-633.
- Hempel, Carl. (1966). Philosophy of Natural Science. Englewood Cliffs, New Jersey: Prentice-Hall.
- Herbert, Sandra. (1971). "Darwin, Malthus, and Selection." Journal of the History of Biology 4: 209-217.
- Hesse, Mary. (1966). Models and Analogies in Science. Notre Dame, Indiana: University of Notre Dame Press.

- Holland, J.H. (1975). Adaptation in Natural and Artificial Systems. Ann Arbor, Michigan: University of Michigan Press.
- Hughes, Sally Smith. (1977). The Virus: The History of A Concept. New York: Science History Publications.
- Hull, David. (1980). "Individuality and Selection." Annual Review of Ecology and Systematics 11: 311-332.
- Jerne, Niels K. (1955). "The Natural-Selection Theory of Antibody Formation." Proceedings of the National Academy of Sciences 42: 849-857.
- Jerne, Niels K. (1966). "The Natural Selection Theory of Antibody Formation; Ten Years Later." In Phage and the Origins of Molecular Biology. Edited by John Cairns, et al. Cold Spring Harbor, Long Island, New York: Cold Spring Harbor Laboratory of Quantitative Biology. Pages 302-312.
- King, J.L. and Jukes, T.H. (1969). "Non-Darwinian Evolution." Science 164: 788-798.
- Kling, R. E. (1971). "A Paradigm for Reasoning by Analogy." Artificial Intelligence 2: 147-178.
- Kohn, David (1980). "Theories to Work By: Rejected Theories, Reproduction and Darwin's Path to Natural Selection." Studies in the History of Biology 4: 67-170.
- Lamarck, J. B. (1809). Zoological Philosophy. Reprinted by Hafner, New York, 1963.
- Lewontin, R. C. (1970). "The Units of Selection." Annual Review of Ecology and Systematics 1: 1-18.
- Malthus, Thomas R. (1798). An Essay on the Principle of Population. London: J. Johnson. Reprint: An Essay on the Principle of Population and a Summary View of the Principle of Population. Edited by Antony Flew. Baltimore, Maryland: Penguin, 1970.
- Minsky, M. (1975). "A Framework for Representing Knowledge." In The Psychology of Computer Vision. Edited by P.H. Winston. New York: McGraw Hill. Pages 211-280.
- Oppenheimer, Robert. (1956). "Analogy in Science." American Psychologist 11: 127-135.
- Popper, Karl. (1979). The Logic of Scientific Discovery. New York: Harper Torchbooks.
- Thagard, Paul, forthcoming. "Frames, Knowledge and Inference." Pittsburgh Studies in the Philosophy of Science.
- Toulmin, Stephen. (1972). Human Understanding. Vol. I. Princeton: Princeton University Press.
- VanLehn, Kurt and Brown, John Seely. (1980). "Planning Nets: A Representation for Formalizing Analogies and Semantic Models of Procedural Skills." In Appitude, Learning, and Instruction. Vol II: Cognitive Process Analyses of Learning and Problem Solving. Edited by R. E. Snow, P.A. Federico, and W.E. Montague. Hillsdale, N.J.: Lawrence Erlbaum Assoc. Pages 95-137.
- Vries, Hugo de. (1903-4). The Mutation Theory. 2 vols. Trans. J.B. Farmer and A.D. Darbishire. New York: Kraus Reprint Company, 1967.
- Winston, Patrick. (1980). "Learning and Reasoning by Analogy." Communications of the Association for Computing Machinery 23: 689-703.

DISCOVERING PATTERNS IN SEQUENCES OF OBJECTS

Thomas G. Dietterich
Department of Computer Science
Stanford University
Stanford, CA 94305

Ryszard S. Michalski
Department of Computer Science
University of Illinois
Urbana, IL 61801

Abstract

A more general kind of sequence-prediction problem—the non-deterministic prediction problem—is defined, and a general methodology for its solution presented. The methodology, called SPARC, employs multiple description models to guide the search for plausible sequence-generating rules. Three different models are presented along with algorithms for instantiating them to discover rules. The instantiation process requires that the initial input sequence be substantially transformed to make explicit important features of the sequence. Four different data transformation operators are described. The architecture of a system called SPARC/E is presented, which implements most of the methodology for discovering sequence-generating rules in the card game *Elcuis*. Examples of the execution of SPARC/E are presented.

1. Introduction

Inductive learning—that is, learning by generalizing specific facts or observations—is a fundamental strategy by which we acquire knowledge about the world. This form of learning is rapidly becoming one of the central research topics in AI. Most research on computer models of inductive learning has addressed the problem of inducing a general description of a concept from a collection of independent instances of that concept (the so-called training instances). Thus, the research has dealt with learning concepts that represent a certain class of instances. The instances can be specific physical objects, interactions, actions, processes, and so on. The learned concepts are general descriptions of classes of such instances.

Learning problems of this type include

- learning a checkers evaluation function [Samuel, 1963, 1967] that assigns to a given class of board situations a certain value,
- learning descriptions of block structures [Winston, 1970],
- determining rules for interpreting mass spectrograms [Buchanan and Mitchell, 1978],
- formulating diagnostic rules for soybean diseases [Michalski and Chilausky, 1980], and
- discovering heuristics to guide the application of symbolic integration operators [Mitchell, Utgoff, and Banerji, 1983].

In Samuel's checkers program, for example, each training instance was a board situation represented as a vector of 16 attributes. The

learned concept was an evaluation function that computed the "value" of any board position for the side whose turn it was to move. No relationships between different board positions were considered. Similarly, Michalski's AQ11 program [Michalski and Chilausky, 1980] was given independent training instances, each describing a diseased soybean plant in terms of 35 multi-valued attributes. Each plant could have one of 19 possible soybean diseases. From several hundred training instances, the program inferred general diagnostic rules for each of these diseases.

This type of inductive learning can be called *instance-to-class generalization*. A review of several methods for such instance-to-class generalization can be found in [Michalski, Carbonell, and Mitchell, 1983]. A comprehensive review of learning research is given in [Dietterich, London, Clarkson, and Dronney, 1982].

Another type of inductive learning involves constructing a description of a whole object by observing only selected parts of it. For example, given a set of fragments of a scene, the problem is to hypothesize the description of the whole scene. A very important case of such *part-to-whole generalization* is where the "part" consists of a fragment of a sequence of objects (or a process evolving in time) and the problem is to induce the hypothetical description of the whole sequence (the process). Once such a description is found, it can be used to predict the possible continuations of the given sequence or process. This class of part-to-whole inductive learning problems we will call *prediction problems*.

This paper investigates the prediction problem for a sequence of objects characterized by a finite set of attributes. An elementary problem of this type is letter-sequence prediction, in which each object in the sequence is characterized by only one attribute: the name of the letter. For example, given a sequence of letters such as

A B X B C W C D V ...

the learning program must discover a "pattern"—that is, a rule that governs the generation of letters in the sequence. In this case, such a rule might state that the sequence is a periodically repeating subsequence of three letters in which the first two letters are successors of the letter appearing in the previous period, while the third letter is the predecessor of the corresponding letter in the previous period. Early papers by Simon and Kotovsky [1963, 1972, 1973] show that just a few relationships (such as successor, predecessor, and equality) are sufficient to represent most such patterns. Related work by Solomonoff [1964] and Hedrick [1976] has investigated grammatical approaches to describing letter sequences.

The sequence prediction problem becomes more difficult when the sequence consists not of simple objects with only a single relevant attribute (like the problem just described), but instead of objects with many relevant attributes. Further complexity is introduced if the pattern describing the sequence also involves a variety of relationships among these attributes. For example, the pattern may involve the periodicity of recurrence of certain properties or the dependence of the next object in the sequence on the properties of objects preceding

it at some arbitrary distance in the past. A sequence prediction problem exhibiting the above-mentioned complexities arises in the card game Eleusis [Abbott, 1977; Gardner, 1977]. Examples from this game will be used to illustrate the general methodology of discovering patterns in sequences described in this paper. The rules for Eleusis are briefly explained in section 2.1.

Before we formulate precisely this problem of discovering patterns in sequences, let us first explain why it is important for current AI research. There are three major AI problems that must be addressed in any solution to this discovery task: (a) the representation problem, (b) the problem of performing model-driven inductive learning with multiple models, and (c) the problem of reasoning about temporal processes. The specific representation problem of interest here is that of automatically determining an appropriate series of transformations of the initial sequence description so that the pattern can be found. The multiple-model inductive learning problem arises because no single model can provide sufficient guidance to the search for plausible descriptions in this domain. The relationship of this problem to reasoning about time is not as strong as the other two problems. However, since temporal processes include as a special case discrete-time linear sequences, some of the techniques developed for sequence prediction may be relevant to the more general problem.

In the next two sections, we discuss in detail the representation problem and the problem of multiple-model induction as they arise in this domain.

1.1. Task-oriented transformation of description space

The problem of transforming the initial problem description arises in many practical domains in which the given data (e.g., the training instances in inductive learning) are observations or measurements that do not include the information most relevant to the task at hand. For example, in character recognition, the input typically consists of a matrix of light intensities representing a character, but the relevant information includes position-invariant properties of letters such as the presence of a line on the left or right of a character, occurrence of line endings, closed contours, and so on (e.g., [Karpinski and Michalski, 1966]). These position-invariant properties can be made explicit by applying *task-oriented* transformations to the raw data.

An example of a learning program that performs task-oriented transformations is INTSUM (a part of the Meta-DENDRAL system, [Buchanan and Mitchell, 1978]). INTSUM is presented with raw training instances in the form of chemical structures (graphs) and associated mass spectra (represented as fragment masses and their intensities). For each fragment in the mass spectrum, INTSUM must determine the bonds that could have broken to produce that fragment. A simple mass spectrometer simulator is used to develop these hypothesized bond breaks. Each of the resulting transformed training instances has the form of a chemical structure and a set of bonds that broke when that structure was placed in the mass spectrometer. It is this information that is provided to the remaining parts of the Meta-DENDRAL system (programs RULEGEN and RULEMOD).

In character recognition programs and in Meta-DENDRAL, the data transformations are fixed in advance. Future learning systems, however, may not know the proper transformations *a priori*. These learning systems will need to select or invent appropriate task-oriented transformations for each learning situation.

This description-space transformation problem has been called by various authors the *data interpretation problem* [Dietterich, et al., 1982] or the *reformulation problem* [Amarel, 1968]. We prefer the term *task-oriented transformation problem*, since it emphasizes that the proper choice of data transformations depends upon the task being performed. In the sequence prediction problem discussed in this

paper, the desired sequence-generating rules are described in a language quite different from the language used to describe the raw sequence. The learning system determines appropriate data transformations from four general classes of transformations and applies them to the raw sequence to produce a transformed sequence amenable to pattern discovery.

The task-oriented transformation problem is part of a spectrum of problems faced by learning programs. The simplest learning algorithms (e.g., linear regression) determine the coefficients for a predetermined, fixed set of variables. Slightly more sophisticated are learning algorithms, such as the A^9 algorithm [Michalski and Kulpa, 1971] or the candidate elimination algorithm [Mitchell, 1978], that are able to determine which terms are relevant and how they should be combined (i.e., with operators such as $\wedge$ and $\vee$). Learning algorithms that perform interpretative transformations (e.g., Soloway [1981], Meta-DENDRAL [Buchanan and Mitchell, 1978]) augment these basic inductive algorithms by applying a set of predetermined transformations to the data prior to inductive generalization. Not yet developed are learning algorithms that could select description-space transformations under guidance of special heuristics. And very few researchers have addressed the problem of discovering new descriptors (predicates, functions, operators, etc.). Table 1-1 shows this spectrum of inductive learning problems.

-
1. Determine coefficients
 2. Select relevant variables and combine
 3. Apply predetermined transformations
 4. Select transformations under heuristic guidance
 5. Discover new descriptors
-

Table 1-1: Spectrum of learning problems in increasing order of difficulty

The method presented in this paper falls under category 4, since it searches four general classes of transformations and employs heuristics reflecting domain-specific knowledge.

1.2. Learning with multiple models

The second major problem that arises in sequence prediction is the problem of learning using multiple description models. This problem has not received much attention in previous AI research. Most existing systems employ a single model that provides guidance to the induction algorithm as it searches a space of possible descriptions. Many systems, for example, use conjunctive descriptions to represent concepts. By constraining the search to consider only conjunctive descriptions, the learning problem is greatly simplified. Michalski [Michalski and Kulpa, 1971] constrains descriptions to be in disjunctive normal form with *fewest* disjunctive terms. This constraint is satisfied (approximately) by having the induction algorithm find first one conjunction, and then another, and so on until all of the training instances are covered. Meta-DENDRAL [Buchanan and Mitchell, 1978] employs a fairly elaborate model of the operation of the mass spectrometer to guide its search for cleavage rules. In general, all of these systems use a single model, and very few authors have made their models explicit.

One researcher who has employed multiple models is Persson [1966]. He applied four different models to the problem of extrapolating number- and letter-sequences. Briefly, these models were

1. a model that computes the coefficients and the degree of a polynomial by applying Newton's forward-difference formula (the degree can be arbitrarily large);
2. an extended model that discovers exponential rules of the form AB^C , where A is a polynomial of degree 4 or less and B and C are polynomials of degree 1 or less (i.e., B and C are of the form $ax + b$);
3. a simple periodic model for periods of length 2 (i.e., intertwined sequences); and
4. a generalization of the Kotovsky and Simon model for Thurstone letter-series that can discover simple periodic and segmented sequence-generating laws.

These models are applied in an artificial learning situation in which the program is given a *sequence* of sequence-extrapolation problems. Thus, in addition to attempting to solve each individual sequence-extrapolation problem, Persson's program tries to predict the *kind* of sequence-prediction problem that it will next receive—that is, it tries to predict which model will best fit the next sequence-prediction problem. Hence, when the program is attempting to solve one of the base-level problems, it selects models to apply based on its predictions about the kind of sequence that it is expecting.

Persson's work shows the value of employing multiple description models to search for sequence-generating rules. The major limitation of Persson's approach, however, is that it is specific to number- and letter-sequence prediction. His methods cannot solve the more general problem described in this paper in which objects have multiple attributes and the task is to find a nondeterministic sequence-prediction rule.

Table 1-2 shows a spectrum of five model-based learning methods. The simplest approach is to use a single fixed model. This has been the common approach in AI thus far. The next step is to provide a learning program with a set of models from which it would choose the most appropriate ones. This is the approach used by Persson. The third level of sophistication would be to have the program generate a predetermined set of models, just as the learning program applies a predetermined set of data transformations. This could be improved by having the program decide which models to generate on the basis of special heuristics. Finally, an even more sophisticated program would be able to invent new models and apply them to guide the learning process.

1. Single model
2. Selection from a few models
3. Predetermined generation of models
4. Heuristically-guided generation of models
5. Discovery of new models

Table 1-2: Spectrum of model-based methods in increasing difficulty

The approach described in this paper searches a predetermined space of possible models in a depth-first fashion, and hence, falls under point 3 of this table.

It is the development of techniques for addressing these two problems—of selecting task-oriented transformations and of applying

multiple description models—that is the main theoretical contribution of this research. In the remainder of this paper, we

1. define the sequence prediction problem under consideration,
2. describe the methods used for representing and transforming the initial training instances,
3. present techniques for representing the models and sequence-generating rules, and finally,
4. provide the details of the program SPARC/E, which implements most of the described methodology. The program is illustrated by a few selected examples of its operation when applied to the inductive card game Eleusis.

2. Problem Statement

Suppose we are observing a process that generates some objects, one after another, and arranges them into a sequence. Suppose that the objects are generated from a known set and that there exists an underlying law that specifies at least some of the properties of every new generated object. We will call such a law a *sequence-generating rule*. It is assumed that the law is expressed in terms of properties that are either observable properties of objects present in the sequence up to the moment when a new object is generated or properties that can be derived from such observable properties by some known inference rules.

We are interested in the most general kind of sequence-generating law in which the law does not necessarily completely determine which objects can or cannot appear next in the sequence. The law merely states some properties that constrain the next object to be a member of a restricted set. Thus, such a generating rule is nondeterministic. The task of discovering such a generating law is a difficult learning task, requiring task-specific data transformations and model-guided induction. We will call this learning problem a *non-deterministic prediction problem* (NDP, for short). If the law guiding the generation of the sequence completely defines the next object at every point in the sequence, then the NDP problem reduces to a *deterministic prediction problem* (DP, for short). In the DP problem, it is assumed that there is no randomness in the generation of the next object. The next object is strictly a function of the past objects.

Many researchers have previously considered DP problems such as letter-sequence prediction, number-series extrapolation, economical prediction, and prediction of the behavior of a computer system. Most recently, the BACON system [Langley, 1980] has addressed a wide range of DP problems that arise in scientific discovery situations. BACON and most of its predecessors make strong use of the constraint that in a DP problem, *all* attributes of the next object in the sequence are determined by the previous objects in the sequence. The NDP problem is more difficult to solve, because only a partial description of the original sequence is sought. Consequently, many more plausible hypotheses must be considered during the inductive learning process.

Let us illustrate a simple NDP problem by an example. Suppose we are given a snapshot of an ongoing process that has already generated the objects (graphs) shown in Figure 2-1.

The observable properties of each graph are: the NUMBER OF NODES, the SHAPE of the graph (T-junction, square, bar, wheel,

triangle, star, diamond), the TEXTURE of each node (solid black, blank, and cross), and the ORIENTATION of the graph (applicable only to graphs that are elongated in some direction, expressed as degrees clockwise from vertical). Suppose we would like now to predict what could be the next object.

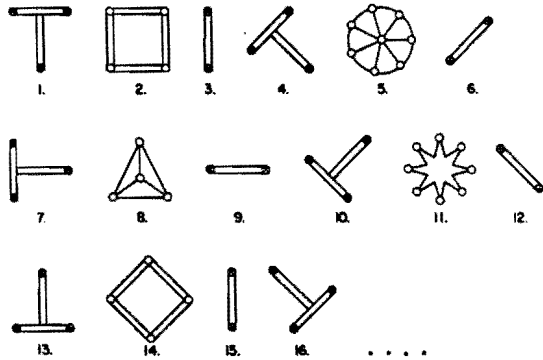


Figure 2-1: A simple NDP problem

By examining the given string in Figure 2-1, we can observe that it can be partitioned into segments of three graphs in length. The nodes of the graphs in each triplet have TEXTURE in the order <solid black, blank, cross>. The SHAPES of the graphs are always <T-junction, *, bar> (where * denotes any shape). We can also notice that the ORIENTATION of the T-junction changes by -45 degrees each time, while the ORIENTATION of the bar increases by $+45$ degrees each time. Finally, the NUMBER OF NODES in the center graph alternates between 4 and 8. If the above regularities indeed constitute the generating law, we can hypothesize that the next graph in the sequence will have 8 blank nodes, and then after that there will be a graph that is a slanted bar with crossed nodes and ORIENTATION of 225 degrees (slant downwards to left). Thus, with regard to the first predicted object, we know only two properties (NUMBER and TEXTURE of nodes), and with regard to the second predicted object, we know it completely. It is easy to see that the problem of letter-sequence prediction (or extrapolation) is a special case of the NDP problem where each object is a letter of an alphabet whose observable property is its name. It also has one derived property that is its position in the alphabet. (The order of letters in the alphabet is externally-provided domain knowledge.) Since each object (in this case a character) is defined completely by specifying its name (or its position in the alphabet), letter series prediction is necessarily a DP problem.

2.1. An exemplary NDP problem: the card game Eleusis

An interesting NDP problem occurs in the card game Eleusis, invented by Robert Abbott [Abbott, 1977; Gardner, 1977]. Eleusis is an inductive game in which players attempt to discover a "secret rule" invented by the dealer. The secret rule is the generating rule for a sequence of cards. Each player, in his or her turn, adds one card to the sequence, and the dealer indicates whether the card is a correct extension of the sequence (i.e., satisfies the secret sequence-generating rule). Players who play incorrectly are penalized by having additional cards added to their hands. The goal of each player is to get rid of all of the cards in his hand, which is only possible if correct cards are played. The cards played during the game are displayed in the form of a layout in which the correct cards form a "main line" and incorrect cards form "side lines" branching down from the main line at the card that they followed. Figure 2-2 shows a typical Eleusis layout for the sequence-generating rule "Play alternating red and black cards." In

this game, the 3 of hearts was played first, followed by a 9 of spades, and a Jack of diamonds. All of these were correct. Following the Jack, a 5 of diamonds was played. It appears on a sideline below the Jack, because it was not a correct extension of the sequence. (At this point a black card is required.) The 4 of clubs was then correctly played, and so on.

Main line:	3H	9S	JD	4C	JD	2C	10D	2C	5H
Side lines:			5D		AH	AS	8H		
					8H	10S	7H		
					QD		10H		

Figure 2-2: A sample Eleusis layout

Eleusis provides a good domain for studying the use of task-oriented data transformations to aid learning. Frequently, the generating law for an Eleusis sequence is stated in terms of descriptors that are not present in the initial sequence. In this example, for instance, the generating law is stated in terms of the color of the cards, but the original sequence supplies only the RANK and SUIT of each card. Table 2-1 provides some examples of generating laws from Eleusis. Note that the terms in which these laws are expressed (e.g., "strings of cards of the same suit", "alternating sequence") are quite different from terms such as RANK and SUIT that described the original sequence. To bridge this difference, appropriate description-space transformations have to be performed.

- If the last card was a spade, play a heart; if last card was a heart, play diamonds; if last was diamond, play clubs; and if last was club, play spades.
- The card played must be one point higher than or one point lower than the last card.
- If the last card was black, play a card higher than or equal to that card; if the last card was red, play lower or equal.
- Play alternating even and odd cards.
- Play strings of cards such that each string contains cards all in the same suit and has an odd number of cards in it.

Table 2-1: Some examples of sequence-generating rules in Eleusis

Eleusis also provides a good domain for studying the use of models for guiding the induction process. The space of possible Eleusis rules using descriptors such as SUIT, RANK, COLOR, FACEDNESS, PARITY, PRIMENESS, and RANK MODULO 3 is very large. In our description language, there are more than 10^{137} possible sequence-generating rules involving four or fewer conjunctive expressions¹. A breadth-first search of this space, such as is conducted by the candidate-elimination

¹This estimate is based on computing the space of all syntactically legal VL1 conjuncts containing the following set of descriptors (after each descriptor is listed the number of elements in its value set and the number of possible selectors that can be formed using those elements): SUIT (4,9), RANK (13, 91), COLOR (2,3), FACEDNESS (2,3), PARITY (2,3), PRIMENESS (2,3), RANKMOD3 (3,7), D-SUIT01 (4,9), D-SUIT02 (4,9), D-RANK01 (25,300), D-RANK02 (25,300), S-RANK01 (25,300), S-RANK02 (25,300), D-COLOR01 (2,3),

algorithm, would clearly be impossible. Fortunately, the rules used by people tend to cluster into certain classes that can be well-described by three models: periodic rules, decomposition rules, and DNF rules. Thus, a model-directed approach can be used to discover sequence-generating rules in Eleusis.

3. Overview of Solution

This section gives an overview of the approach taken to solving the NDP problem defined in section 2. The approach is a combination of bottom-up data transformation, top-down model specialization, and data-driven instantiation of the specialized models to fit the transformed data. These three processes can be briefly explained as follows:

1. Bottom-up data transformation involves applying various transformation operators to the initial sequence description to obtain a *derived sequence description*. We use four basic data transformations: adding derived attributes, segmenting, splitting, and blocking. Details of these are described in section 4.
2. Top-down model specialization involves specifying particular values for the parameters of general rule models to obtain a *parameterized model*. We use three general models: the disjunctive normal form model (DNF), the decomposition model, and the periodic model. Each of these models has one or more parameters. For example, both the DNF and decomposition models have a single parameter: the *lookback*, L (i.e., the number of objects back from the given object in the sequence that are assumed to determine the next object). The periodic model has two parameters: the *lookback*, L , and the *period length*, P , which indicates the length of the repeating period in the sequence. Details of the model specialization process are described in section 5.
3. The model-instantiation process attempts to fit the parameterized model to the derived sequence description to produce an instantiated parameterized model. A model that has been parameterized and instantiated serves as a sequence-generating rule. This process is described in section 6.

The above three steps are illustrated schematically in figure 3-1.

Model instantiation, as used in this paper, is an extension of the well-known AI technique of schema instantiation. Schema instantiation has been applied, for example, by Schank and Abelson [1975] to interpret natural language, by Englemore and Terry [1979] to interpret X-ray diffraction data in protein chemistry, and by Friedland [1979] to plan genetics experiments. Model instantiation differs from schema instantiation in the complexity of the instantiation process.

D-COLOR02 (2,3), D-FACEDNESS01 (2,3), D-FACEDNESS02 (2,3), D-PARITY01 (2,3), D-PARITY02 (2,3), D-PRIMENESS01 (2,3), D-PRIMENESS02 (2,3), D-RANKMOD3-01 (3,7), D-RANKMOD3-02 (3,7). The SUT and RANKMOD3 descriptors are cyclically ordered, while the RANK descriptors are interval descriptors. All others are nominal. In a block of three adjacent cards (with lookback $L=2$), the first seven descriptors appear three times—once for each card. Hence, the total number of possible conjuncts is $(9 \cdot 2! \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 7)^3 \cdot (9 \cdot 300 \cdot 300 \cdot 3 \cdot 3 \cdot 3 \cdot 3 \cdot 7)^2 = 2.11221 \cdot 10^{34}$. If there are four conjuncts in a rule, then we obtain $[2.11221 \cdot 10^{34}]^4 = 1.99 \cdot 10^{137}$.

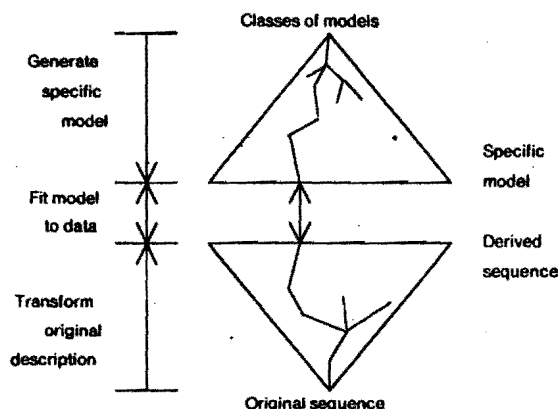


Figure 3-1: Schematic description of the rule discovery process

Model instantiation involves not only filling in predetermined slots or substituting constants for variables, but also synthesizing a logical formula of an assumed type. For example, in order to instantiate each of the three models described below, the program must synthesize a conjunction of predicates or a disjunction of such conjunctions that satisfies certain constraints. Model-instantiation methods share with schema-instantiation methods the advantage that they are efficient, and also effective with noisy and uncertain data. The constraints provided by the models (or schemas) drastically reduce the size of the space that the program must search.

The principal disadvantage of model- and schema-instantiation methods is that they require substantial amounts of domain knowledge to be built into the program. In order to keep this domain knowledge explicit and easily modified, we employ a ring architecture in the design of the learning program, as described in section 6. This architecture facilitates the application of the system to a variety of problems by simplifying the process of changing the domain-specific parts of the program.

4. Describing and Transforming Training Instances

Now that we have defined the problem to be solved (the NDP problem) and sketched the solution, we launch into the details of that solution. This section presents the description language for representing the original sequences and the transformation operators that can be applied to modify that representation.

4.1. Representing the initial sequence

A sequence of objects is represented as an indexed sequence²

$$\langle q_1, q_2, \dots, q_k \rangle$$

It is assumed that the only relevant relationship between two objects is their ordering in the sequence. Each object is described by a set of attributes (also called *descriptors*) $f_1, f_2, \dots, f_n$, which can be viewed as functions mapping objects into attribute values. To state that attribute f_i of object q_j has value r , we write

$$[f_i(q_j) = r].$$

²A summary of the notational conventions used in this paper appears in section 9.

This notation is called a *selector*. For example, if f_1 is *color* and r is *red*, then the selector

$$[\text{color}(q_j)=\text{red}]$$

states that the color of the j -th object in the sequence is red.

Each attribute is only permitted to take on values from a finite value set called the *domain*, $D(f_i)$, of that attribute. This constraint is part of the background knowledge that has to be given to the program. For example, in a deck of cards, the domain of the SUIT attribute is {clubs, diamonds, hearts, spades}. Additional knowledge about the domain set can be represented. In particular, the domain set may be linearly ordered, cyclically ordered (i.e., in a circular, wrap-around ordering), or tree ordered. We will see below how these domain orderings are applied to the problem of representing cards in an Eleusis game.

A complete initial description of a single object, q_j , called an *event*, is an expression giving the values for all of the attributes of q_j . This is usually written as a conjunction of selectors:

$$[f_1(q_j)=r_1][f_2(q_j)=r_2]\dots[f_n(q_j)=r_n].$$

It can also be represented as a vector of attribute values:

$$(r_1, r_2, \dots, r_n).$$

This vector notation suggests that each object description can be viewed as a point in the *event space* E :

$$E = D(f_1) \times D(f_2) \times \dots \times D(f_n)$$

This event space contains all possible events.

A complete description of the initial sequence is a sequence of conjunctions of selectors (or alternatively, a sequence of attribute vectors)—one conjunction for each object in the sequence. The space of all possible sequences can be generated by selecting all possible sequences of events chosen from E .

4.2. Transforming the Sequence

As we mentioned in section 1, it is often necessary to transform the initial sequence into a *derived sequence* in order to facilitate the discovery of sequence-generating rules. Such a data transformation can be viewed as a mapping T from one set of sequences S , containing objects Q , described by attributes F , to another set of *derived sequences* S' , containing *derived objects* Q' , and described by *derived attributes* F' .

$$T_{p_1, \dots, p_k} : \langle S, Q, F \rangle \rightarrow \langle S', Q', F' \rangle$$

where $p_1, \dots, p_k$ are parameters of the transformation that control its application. We have found four basic transformations to be especially useful for discovering sequence-generating rules: (a) *adding derived attributes*, (b) *segmenting*, (c) *splitting into phases*, and (d) *blocking*. Each of these is described in turn.

4.2.1. Adding derived attributes

The simplest transformation does not change the set of sequences, S , or the set of objects, Q , but only the set of attributes, F . For example, in Eleusis, the initial set F contains only two attributes: the RANK and SUIT of a card. These can be augmented by deriving such attributes as COLOR (red or black), FACEDNESS (faced or nonfaced), PARITY (odd or even), and PRIMENESS (prime or not prime in rank). The adding-derived-attributes transformation has no parameters.

4.2.2. Segmenting

The segmenting transformation derives a new sequence made up of a new set of objects, Q' , and described with a new set of attributes, F' . The new sequence is produced from the original sequence by dividing the original sequence into non-overlapping segments. Each segment becomes a derived object in the new sequence. The only parameter of the segmenting transformation is the segmentation condition that tells

how the original sequence should be divided into segments. Three types of segmentation conditions can be distinguished: (a) those that use properties of the *original objects* to determine where the sequence *should* be broken, (b) those that use properties of the *original objects* to determine where the sequence *should not* be broken, and (c) those that use properties of *derived objects* to determine where the original sequence *should* be broken.

For example, suppose the original sequence consists of physical objects described by attributes such as WEIGHT, COLOR, and HEIGHT. An example of each type of segmentation condition follows:

1. Break when
 $[\text{weight}(q_{i-1}) > 10][\text{weight}(q_i) \leq 10].$

According to this condition, the original sequence is to be broken (between q_{i-1} and q_i) at the point where the weight of an object changes from above 10 to under 10.

2. Don't break as long as
 $[\text{color}(q_i) = \text{color}(q_{i-1})][\text{weight}(q_i) > 10].$

This condition states that the original sequence will not be broken (between q_{i-1} and q_i) if the color stays the same and the weight remains above 10. It will be broken at any point where these conditions do not both hold.

3. Break so that $[\text{length}(q_i') = 2].$

This condition states that derived objects (q_i') should be subsequences of length 2 from the original sequence (i.e., pairs of adjacent objects from the original sequence).

The choice of attributes, F' , for describing the newly-derived objects, Q' , depends on the segmentation condition used to segment the sequence. For example, if the $[\text{length}(q_i') = 2]$ condition is used, attributes of interest might include the sum of the VALUES of the two original objects, the maximum VALUE, the minimum VALUE, and so on. The LENGTH of the segment would not be of interest, since by definition, it is a constant. However, if the $[\text{color}(q_i) = \text{color}(q_{i-1})]$ condition is used, the LENGTH of the segment could be quite interesting and should be derived. Also, the COLOR shared by all of the cards in the segment might be of interest. In our implementation, the user specifies which attributes should be derived. All user-specified attributes are derived unless the program can prove from the segmentation condition that those attributes would not have a well-defined value for each segment in the sequence or else would be trivially constant for all segments.

Often, a segmentation condition leads to the creation of incomplete segments at the beginning and end of the original sequence. These boundary cases can create difficulties during model instantiation, so they are ignored during rule discovery, but checked during rule evaluation.

4.2.3. Splitting

The splitting transformation splits a single sequence into a *sequence* of P separate subsequences: $\langle ph_1, ph_2, \dots \rangle$. Sequence ph_i starts with object q_i (the object at the i -th position in the original sequence) and continues with objects taken from succeeding positions at distance P apart in the original sequence. Each of the derived sequences is called a *phase*. P is the parameter of the splitting transformation that denotes the number of phases. Figure 4-1 shows the splitting operation for $P = 3$.

Original sequence: <q1 q2 q3 q4 q5 q6 q7 q8 q9>
 Derived sequence: <ph1 ph2 ph3>, where
 ph1: <q1 q4 q7>
 ph2: <q2 q5 q8>
 ph3: <q3 q6 q9>

Figure 4-1: Splitting transformation with P=3

The objects within each phase retain the linear ordering that they had in the original sequence. The phases themselves can be considered to be cyclically ordered so that ph_1 precedes ph_2 , which precedes ph_3 , and so on, until ph_p , which is followed by ph_1 again. Consider, for example, the following sequence:

<1 8 2 9 3 10 4 11>

The splitting transformation with P=2 would produce the sequence <ph1 ph2> where

ph1 = <1 2 3 4>
 ph2 = <8 9 10 11>

Since the splitting transformation simply breaks the original sequence of objects into subsequences, no new objects are created. Furthermore, no new descriptors are defined. The set of descriptors used to characterize the objects in each of the phases is the same as the set of descriptors used to characterize the objects in the original sequence.

The splitting transformation can be applied to break one sequence-prediction problem into several subproblems—one for each phase. This is how periodic rules are discovered.

4.2.4. Blocking

The blocking transformation converts the original sequence into a new sequence made up of a new set of objects B' and a new set of attributes F'. The new sequence is created by breaking the original sequence into overlapping segments called *blocks*. Each object b_i in the new sequence describes a block of L+1 consecutive objects from the original sequence, starting at object q_i (called the *head*) and proceeding backwards to object q_{i-L} (where L is the lookback parameter of the blocking transformation). Figure 4-2 shows the blocking operation for L=2 (Block length of 3).

Several attributes are derived to describe each block. For each attribute A applicable to the objects in the original sequence, the attributes $A_0, A_1, \dots, A_L$ are defined that are applicable to the objects in the derived sequence. $A_0(b_i)$ has the same value as $A(q_i)$; $A_1(b_i)$ has the same value as $A(q_{i-1})$; and so on until $A_L(b_i)$, which has the same value as $A(q_{i-L})$. In other words, the original attributes are retained in

Original sequence: <q1 q2 q3 q4 q5 q6 q7 q8>
 Derived sequence: <b3 b4 b5 b6 b7 b8>
 where b_i are derived objects defined as follows:
 b3: <q1 q2 q3>
 b4: <q2 q3 q4>
 b5: <q3 q4 q5>
 b6: <q4 q5 q6>
 b7: <q5 q6 q7>
 b8: <q6 q7 q8>

The underlined object in each block is the head object.

Figure 4-2: The Blocking Transformation with L=2.

the new sequence, but they are renamed so that they apply to whole *blocks* rather than to individual objects in the original sequence. The numerical suffix on the new names encodes the relative position of the original object q_i in block b_j .

For example, suppose we have the sequence <q1 q2 q3 q4 q5> with attributes RANK and SUIT, where

[rank(q1)=2][suit(q1)=H]
 [rank(q2)=4][suit(q2)=S]
 [rank(q3)=6][suit(q3)=C]
 [rank(q4)=8][suit(q4)=D]
 [rank(q5)=10][suit(q5)=H]

Now suppose we apply the blocking transformation to this sequence with L=2 to obtain the derived sequence of blocks <b3 b4 b5>. Then the descriptors RANK0, RANK1, RANK2, SUIT0, SUIT1, and SUIT2 will be derived with the values

[rank2(b3)=2][suit2(b3)=H]
 [rank1(b3)=4][suit1(b3)=S]
 [rank0(b3)=6][suit0(b3)=C]
 [rank2(b4)=4][suit2(b4)=S]
 [rank1(b4)=6][suit1(b4)=C]
 [rank0(b4)=8][suit0(b4)=D]
 [rank2(b5)=6][suit2(b5)=C]
 [rank1(b5)=8][suit1(b5)=D]
 [rank0(b5)=10][suit0(b5)=H]

This transformation leads to a highly redundant representation of the information in the original sequence. For example, the information about SUIT and RANK of the original object q_3 is repeated as SUIT0 and RANK0 of block b_3 , SUIT1 and RANK1 of block b_4 , and SUIT2 and RANK2 of block b_5 . However, this derived sequence of blocks facilitates the representation of the relationships between objects in the original sequence. Many sequence-prediction rules involve such relationships.

To represent relationships between objects, additional descriptors called *sum* and *difference* descriptors are defined. In the case of the above sequence, the descriptors S-RANK01, S-RANK02, D-RANK01, D-RANK02, D-SUIT01, and D-SUIT02 are created. The value of S-RANK01(b_i) is the sum of RANK0(b_i) and RANK1(b_i). The value of D-RANK01(b_i) is the difference between RANK0(b_i) and RANK1(b_i). Thus, in addition to the selectors shown above, the following selectors would also be derived for the new sequence:

[s-rank01(b3)=10][s-rank02(b3)=8]
 [d-rank01(b3)=2][d-rank02(b3)=4]
 [d-suit01(b3)=1][d-suit02(b3)=2]
 [s-rank01(b4)=14][s-rank02(b4)=12]
 [d-rank01(b4)=2][d-rank02(b4)=4]
 [d-suit01(b4)=1][d-suit02(b4)=2]
 [s-rank01(b5)=18][s-rank02(b5)=16]
 [d-rank01(b5)=2][d-rank02(b5)=4]
 [d-suit01(b5)=1][d-suit02(b5)=2]

From this representation, it is relatively easy to discover that [d-rank01(b_1)=2] is true for all blocks b_i .

Ordinarily, sum and difference attributes only make sense for attributes such as RANK whose domain sets are linearly ordered. We have extended the definition of difference to cover unordered and cyclically ordered domain sets as well. For an unordered attribute such as COLOR, whose domain set is {red, black}, D-COLOR01 takes on the value 0 if the COLOR0(b_i) = COLOR1(b_i) and 1 otherwise. For attributes with cyclically-ordered domain sets, such as SUIT (with values {clubs, diamonds, hearts, spades}), D-SUIT01 is equal to the number of steps in the forward direction that are required to get from SUIT1(b_i) to SUIT0(b_i). If SUIT1(b_i)=diamonds and SUIT0(b_i)=clubs, D-SUIT01(b_i)=3.

The sum and difference attributes make the ordering of the original sequence explicit in the attributes that describe each block.

Consequently, it is no longer necessary to represent the ordering between blocks. Hence, the model-fitting algorithms discussed below treat the derived sequence (of blocks) as an unordered set of events.

One difficulty with the above approach is that the numerical suffix notation is not very easy to read, especially when it is combined with a sum or difference prefix. Hence, we have developed an alternative representation that is more comprehensible. In this notation, selectors that refer to blocks, such as $[suit1(b1)=H]$, are written as selectors that refer to objects in the original sequence, such as $[suit(q1-1)=H]$. Similarly, selectors such as $[d-rank01(b1)=3]$ are written as $[rank(q1)=rank(q1-1)+3]$. This notation makes the meaning of the selectors clear without having to explicitly mention the blocks b_i . For purposes of implementation, the first notation is better because it enables the program to treat all sequences—including derived sequences—uniformly. However, the second notation is more understandable and hence will be used for the rest of this paper.

5. Representing Sequence-generating Rules and Models

A *sequence-generating rule* is a function g that assigns to each sequence of objects, $\langle q_1, q_2, \dots, q_k \rangle$, a non-empty set of *admissible next objects* Q_{k+1} :

$$g: \{\langle q_1, q_2, \dots, q_k \rangle\} \rightarrow \{Q_{k+1}\}$$

Q_{k+1} is the set of all objects that could appear as the next object in the sequence. For example, in the rule "Play a card whose rank is one higher than the previous card", $g(\langle \dots 4C \rangle) = Q_{k+1}$ is the set of cards $\{5C, 5D, 5H, 5S\}$.

The set Q_{k+1} may contain only one event, or it may contain a large set of possible events. If for all k , the sequence $\langle q_1, q_2, \dots, q_k \rangle$ is mapped by g into a singleton set, then the rule is a *deterministic* rule; otherwise, it is a *nondeterministic* rule. This paper addresses the problem of discovering a nondeterministic sequence-generating rule, g , given the sequence $\langle q_1, q_2, \dots, q_k \rangle$.

The sequence $\langle q_1, q_2, \dots, q_k \rangle$ can be viewed as the set of assertions

$$\begin{aligned} q_1 &\in g(\langle \rangle) \\ q_2 &\in g(\langle q_1 \rangle) \\ &\vdots \\ q_m &\in g(\langle q_1, \dots, q_{m-1} \rangle) \end{aligned}$$

These assertions are positive instances of the desired sequence-generating rule.

In Eleusis, negative instances are provided by the cards on the sidelines—that is, the cards rejected by the dealer for being incorrect. A sideline card q_3^- played after card q_3 provides a negative instance of the form:

$$q_3^- \notin g(\langle q_1, q_2, q_3 \rangle)$$

The goal is to find a description for g that is consistent with these training instances and satisfies some preference criterion.

The preference criterion in our methodology (and in all learning systems) attempts to evaluate a candidate rule in terms of its generality, predictive power, simplicity, and so on. These semantic properties are difficult to compute, however. Instead, virtually all learning systems employ syntactic criteria that correspond in some way to these semantic criteria. Syntactic criteria—such as the number of selectors in a conjunction and the number of conjuncts in a disjunction—will only correspond to the semantic criteria if the

representational framework is well chosen (See McCarthy [1958]). As we noted in the introduction, most previous AI research on learning has employed a single representational framework or model for describing the rules or concepts to be learned. In Eleusis, a single framework is insufficient. Instead, we have developed three basic models that were found to be useful: the DNF model, the decomposition model, and the periodic model. When these models are employed, syntactic criteria can be used to approximate semantic criteria during evaluation.

A *model* is a logical schema that specifies the syntactic form of a class of descriptions (in our case, sequence-generating rules). A model consists of *model parameters* and a set of *constraints* that the model places on the forms of descriptions. The process of specifying the values for the parameters of a model is called *parameterizing* the model. The process of filling in the form of the parameterized model is called *instantiating* the model. A fully-parameterized and fully-instantiated model forms a sequence-generating rule. Models can be instantiated using the original sequence, or, more typically, using a sequence derived by applying some of the data transformations discussed in the previous section.

All three models use the representation language VL22 as a building block for expressing sequence-generating rules. VL22 is an extension to the predicate calculus that uses the *selector* as its simplest kind of formula. The VL22 selector is substantially more expressive than the simple selector presented above in section 4.1. The simple selector has the form:

$$[f1(qj)=r]$$

whereas the VL22 selector has the form:

$$[f1(x1, x2, \dots, xn) = r1 \vee r2 \vee \dots \vee rm]$$

In the VL22 selector, attributes f_i can take any number of arguments $(x_1, x_2, \dots, x_n)$. Furthermore, the attributes f_i can take on any one of a set of values $\{r_1, r_2, \dots, r_m\}$. The $\vee$ denotes the *internal disjunction* operator. Thus, the selector

$$[rank(q1)=9 \vee 10 \vee J \vee Q \vee K]$$

indicates that the rank of object q_1 can be either 9, 10, J, Q, or K. The internal disjunction represents disjunction over the values of a single variable. In this case, it could be expressed alternatively as

$$[rank(q1) \geq 9],$$

since the domain of the RANK attribute is known to be linearly ordered with a maximum value of K (King). To aid comprehensibility, VL22 provides the operators $\langle$, $\rangle$, $\leq$, $\geq$, and $\neq$, in addition to the basic $=$ operator.

Examples of typical selectors include:

$$[rank(q1) \neq rank(q1-1)]$$

(paraphrase: the RANK of q_1 is different from the RANK of q_{1-1})

$$[suit(q1)=suit(q1-1)+1]$$

(paraphrase: the SUIT increases by one from q_{1-1} to q_1)

$$[rank(q1)+rank(q1-2)>10]$$

(paraphrase: the sum of the RANKs of q_1 and q_{1-2} is greater than 10)

Now that we have introduced the basic notation of VL22, each of the three rule models is presented in turn.

5.1. The DNF model

The DNF model supports the broad class of rules that can be expressed as a universally quantified VL22 statement in disjunctive

normal form. The DNF model has one parameter, the degree of lookback, L . An example of a DNF rule (with $L=1$) is:

$$\forall i ([\text{color}(q_i)=\text{color}(q_{i-1})] \vee [\text{rank}(q_i)=\text{rank}(q_{i-1})])$$

In general, a DNF rule is a collection of conjuncts of the form

$$\forall i (C_1 \vee C_2 \vee C_3 \vee \dots \vee C_k)$$

The universal quantification over i indicates that this description is true for all objects q_i in the sequence.

An additional constraint specified in the DNF model is that the number of conjuncts, k , should be close to the minimum that produces a description consistent with the data.

5.2. The Decomposition Model

The decomposition model constrains the description to be a set of implications of the form:

$$\begin{aligned} L_1 &\Rightarrow R_1 \\ L_2 &\Rightarrow R_2 \\ &\vdots \\ L_m &\Rightarrow R_m \end{aligned}$$

where the $\Rightarrow$ sign indicates logical implication.

The model states that the left- and right-hand sides, L_j and R_j , must all be VL22 conjunctions. The left-hand sides must be mutually exclusive and exhaustive—that is,

$$L_1 \vee L_2 \vee \dots \vee L_m \equiv \text{TRUE}, \text{ and } \forall j, k (j \neq k \Rightarrow (L_j \wedge L_k \equiv \text{FALSE})).$$

A decomposition rule describes the next object in the sequence in terms of characteristics of the previous objects in the sequence. For example, the rule

$$\forall i ([\text{color}(q_{i-1})=\text{black}] \Rightarrow [\text{parity}(q_i)=\text{odd}] \vee [\text{color}(q_{i-1})=\text{red}] \Rightarrow [\text{parity}(q_i)=\text{even}])$$

is a decomposition rule that says that if the last card was black, the next card must be odd, and if the last card was red, the next card must be even.

The decomposition model has a lookback parameter, L , that indicates how far back in the sequence the description "looks" in order to predict the next object in the sequence. The above rule has a lookback parameter of 1, because it examines q_{i-1} .

5.3. The Periodic Model

This model consists of rules that describe objects in the sequence as having attribute values that repeat periodically. For example, the rule "Play alternating red and black cards" is a periodic rule. The periodic model has two parameters: the period length, P , and the lookback, L . The period length parameter, P , gives the number of phases in the periodic rule. A periodic rule can be viewed as applying a splitting transformation to split the original sequence into P separate sequences. Each separate phase sequence has a simple description. The lookback parameter, L , tells how far back, within a phase sequence, a periodic rule "looks" in order to predict the attributes of the next object in that phase. The periodic model imposes the additional constraint (or preference) that the different phases be disjoint (i.e., any given card is only playable within one phase).

A periodic rule is represented as an ordered P -tuple of VL22 conjunctions. The j -th conjunct describes the j -th phase sequence. The rule

$$\langle [\text{color}(q_i)=\text{red}], [\text{rank}(q_i) \geq \text{rank}(q_{i-1})] \rangle$$

is a periodic rule with $P=2$ and $L=1$, which says that the sequence is made of two (interleaved) phases. Each card in the first phase is red; each card in the second phase has at least as large a rank as the preceding card in that phase. Hence, one sequence that satisfies this rule is $\langle 2H \ 3C \ 10H \ 5S \ AD \ 6S \ 6H \ 6C \rangle$.

A more complex periodic rule is the rule used to generate the sequence shown in Figure 2-1. It can be represented as

$$\langle [\text{texture-of-nodes}(q_i)=\text{solid black}] \& [\text{shape}(q_i)=\text{T-junction}] \& [\text{orientation}(q_i)=\text{orientation}(q_{i-1})-45],$$

$$[\text{texture-of-nodes}(q_i)=\text{clear}] \& \langle [\text{number-of-nodes}(q_i)=4], [\text{number-of-nodes}(q_i)=8] \rangle,$$

$$[\text{texture-of-nodes}(q_i)=\text{cross}] \& [\text{shape}(q_i)=\text{bar}] \& [\text{orientation}(q_i)=\text{orientation}(q_{i-1})+45] \rangle$$

Notice that this is a periodic rule with three phases and a lookback of 1. The middle phase of the period is itself a periodic rule with the NUMBER-OF-NODES alternating between 4 and 8.

5.4. Derived models

The three basic models can be combined to describe more complex rules. Basic models can be joined by conjunction, disjunction, and negation. For example, the rule "play alternating red and black cards such that the cards are in non-decreasing order" is a conjunction of the periodic rule

$$\langle [\text{color}(q_i)=\text{red}], [\text{color}(q_i)=\text{black}] \rangle$$

and the DNF rule

$$[\text{rank}(q_i) \geq \text{rank}(q_{i-1})].$$

5.5. Model Equivalences and the Heuristic Value of Models

The reader may have noticed that the decomposition and periodic models appear to be special cases of the DNF model. For instance, given that the clauses in a decomposition rule are mutually-exclusive and exhaustive, the decomposition rule

$$\begin{aligned} L_1 &\Rightarrow R_1 \& \\ L_2 &\Rightarrow R_2 \& \\ &\vdots \\ L_m &\Rightarrow R_m \end{aligned}$$

can be written as the DNF rule

$$[L_1 \& R_1] \vee [L_2 \& R_2] \vee \dots \vee [L_m \& R_m]$$

Similarly, if the clauses of a periodic rule are mutually-exclusive and exhaustive, then the periodic rule

$$\langle C_1, C_2, \dots, C_k \rangle$$

can be expressed as a decomposition rule of the form

$$\begin{aligned} C_1 &\Rightarrow C_2 \\ C_2 &\Rightarrow C_3 \\ &\vdots \\ C_{k-1} &\Rightarrow C_k \\ C_k &\Rightarrow C_1 \end{aligned}$$

Even when the constraints of mutual exclusion and exhaustion are violated, it is usually possible to develop some equivalent DNF rule for any periodic or decomposition rule. For instance, in the periodic rule

`< [color(q1-0)=red], [rank(q1-0)=even] >`
(paraphrase: play alternating red and even cards)

the different phases are overlapping. The above transformation into a decomposition rule

`[color(q1-1)=red] => [parity(q1)=even] &`
`[parity(q1-1)=even] => [color(q1)=red]`

does not work, because, for example, the sequence

`<3D 2D 4C ...>`

satisfies the second rule (the first if-then clause can be applied twice), but not the first rule (since the 4C is not red). However, it is possible to get around this particular problem by defining a new descriptor for each object in the original sequence, called POSITION, that has the value i for object q_i . With this descriptor, the above rule can be encoded as

`[position(q1)=odd] => [color(q1)=red]`
`[position(q1)=even] => [parity(q1)=even]`

Hence, it appears that all rules can be written as DNF rules.

Given this fact, it is reasonable to ask why multiple models should be used at all. The answer is that the primary value of multiple models is that they provide heuristic guidance to the search for plausible rules. Hence, though the DNF model is capable of *representing* all of these rules, it is not helpful for *discovering* them. In short, it is *epistemologically adequate* but not *heuristically adequate* (see [McCarthy and Hayes, 1969; McCarthy, 1977]). Each model directs the attention of the learning system to a small subspace of the space of all possible DNF VL22 rules. The next section shows how the constraints associated with each model are incorporated into special model-fitting induction algorithms.

6. Architecture and Algorithms

In section 3 we described the three basic processes involved in discovering sequence-generating rules: (a) transformation of the original sequence to obtain a derived sequence, (b) selection of appropriate models for the given sequence, and (c) fitting of the models to the derived sequence. In sections 4 and 5, the four data transformations and the three models were presented. This section covers the third step of fitting the specialized models to the transformed sequence. The model-fitting process is most easily understood in the context of the program architecture, so this section also discusses the architecture in detail.

6.1. Overview of the Program

The processes in the program (see Figure 6-1) are structured into four components—the three basic components mentioned above plus an evaluation step. The processes of transforming the initial sequence and of selecting and parameterizing a model are performed in parallel. Then, specialized model-fitting algorithms use the transformed sequence to instantiate the model to obtain a candidate sequence-generating rule. These candidate rules are then evaluated to determine a final set of rules.

The reason for performing data transformation and model selection in parallel is that these two processes are interdependent. For example, if a periodic model is selected (with period length P), then a splitting transformation (with number of phases P) needs to be applied to the sequence. These two processes can be viewed as simultaneous cooperative searches of two spaces: the space of possible data transformations and the space of possible parameterized models.

6.2. Overview of the Concentric Ring Architecture

In order for the learning program to be easily modified to handle entire classes of NDP problems, the program is structured as a set of concentric knowledge rings (see Figure 6-2). A knowledge ring is a set of routines that perform a particular function using only knowledge appropriate to that function. The procedures within a given ring may invoke other procedures in that ring or in rings that are inside the given ring. Under these constraints, the concentric ring structure forms a hierarchically organized system.

Ideally, the rings should be organized so that the outermost ring uses the most problem-specific knowledge and performs the most problem-specific operations and the inner-most ring uses the most general knowledge and performs the most general tasks. Such an architecture improves the program's generality because it can be applied to increasingly different NDP problems by removing and replacing the outer rings. In order to apply the program to radically different learning problems, all but the inner-most ring may need to be replaced.

The ring architecture is used here as follows. The outer-most rings perform user-interface functions and convert the initial sequence from whatever domain-specific notation is being used into a sequence of

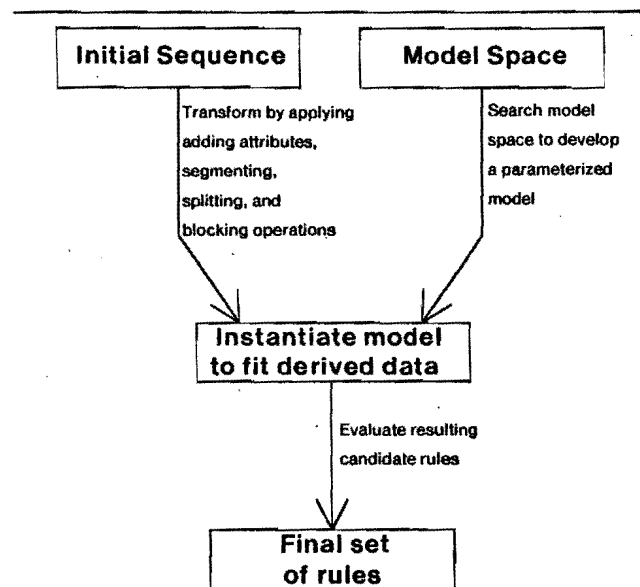


Figure 6-1: The Model-fitting Approach

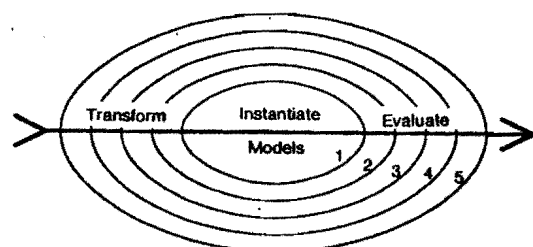


Figure 6-2: The knowledge ring architecture

VL22 events. The inner-most ring performs the model-fitting functions. It expects the data to be properly transformed so that the data have the same form as the models to which they are to be fitted. The intervening rings conduct the simultaneous processes of developing a properly parameterized model and transforming the input sequence into an appropriate form.

The intervening rings also evaluate the rules discovered by the inner-most ring using the knowledge available in each ring.

6.3. The Program SPARC (ELEUSIS version)

SPARC (Sequential Pattern ReCognition) is a general program designed to solve a variety of NDP problems using the ring architecture. So far, we have implemented only a more specific version of the program, called SPARC/E, tailored specifically to the problem of rule discovery in the game Eleusis. SPARC is made up of five rings, as shown in Figure 6-2. This section describes the functions of each ring in the SPARC/E version of the program. To illustrate these ring functions, we use the Eleusis layout shown in Figure 6-3. Recall that in an Eleusis layout, the main line shows the correctly-played sequence of cards (positive examples). The side lines, which branch out below the main line, contain cards that do not satisfy the rule—that is, incorrect continuations of the sequence (negative examples).

Main line:	3H	9S	4C	JD	2C	10D	8H	7H	2C
Side lines:		JD		AH	AS				10H
		5D		8H	10S				
				QD					

Figure 6-3: Sample Eleusis Layout

6.3.1. Ring 5: User Interface

Ring 5, the outer-most ring, provides a user interface to the program. It executes user's commands for playing the card game Eleusis, as well as commands for controlling the search, data transformation, generalization, and evaluation functions of the program. One command in Ring 5 is the INDUCE command that instructs SPARC/E to look for plausible NDP rules that describe the current sequence. When the INDUCE command is given, Ring 5 calls Ring 4 to begin the rule discovery process.

6.3.2. Ring 4: Adding Derived Attributes

Ring 4 applies the adding-derived-attributes transformation to convert the Eleusis layout into a sequence of VL22 events. This involves creating derived attributes that make explicit certain commonly known characteristics of playing cards that are likely to be used in an Eleusis rule: COLOR, PARITY, FACED versus NON-FACED cards, and so on. Figure 6-4 shows the layout from Figure 6-3 after it has been processed by Ring 4. The plusses and minuses along the right-hand side of the figure indicate whether the event is a positive example or a negative example of the sequence-generating rule. These derived events are passed to Ring 3 for further processing.

6.3.3. Ring 3: Segmenting the Layout

Ring 3 is the first Eleusis-independent ring. It applies the segmenting transformation to the sequence supplied by Ring 4. In the present implementation, the end points of each segment are determined by applying a segmentation predicate, $P(\text{card}_{i-1}, \text{card}_i)$ to all pairs of adjacent events in the sequence. When the predicate P evaluates to FALSE, the sequence is broken between card_{i-1} and card_i

VL22 event	positive or negative
[rank(card1)=3][suit(card1)=H]	
[parity(card1)=odd][color(card1)=red]	
[prime(card1)=N][faced(card1)=Y]	+
[rank(card2)=9][suit(card2)=S]	
[parity(card2)=odd][color(card2)=black]	
[prime(card2)=N][faced(card2)=N]	+
[rank(card3)=J][suit(card3)=D]	
[parity(card3)=odd][color(card3)=red]	
[prime(card3)=Y][faced(card3)=Y]	-
[rank(card3)=8][suit(card3)=D]	
[parity(card3)=odd][color(card3)=red]	
[prime(card3)=N][faced(card3)=Y]	-
[rank(card3)=4][suit(card3)=C]	
[parity(card3)=even][color(card3)=black]	
[prime(card3)=N][faced(card3)=N]	+
[rank(card4)=J][suit(card4)=D]	
[parity(card4)=odd][color(card4)=red]	
[prime(card4)=Y][faced(card4)=Y]	+
etc.	

Figure 6-4: Derived layout after Ring 4 processing.

to form the end of a segment. Typical segmentation predicates used are:

```
[rank(cardi)=rank(cardi-1)]
[rank(cardi)=rank(cardi-1)+1]
[color(cardi)=color(cardi-1)]
[suit(cardi)=suit(cardi-1)]
[parity(cardi)=parity(cardi-1)]
```

Other techniques for performing segmentation, such as providing a predicate that becomes TRUE at a segment boundary (see section 4.2.2), are not implemented in SPARC/E.

Ring 3 searches the space of possible segmentations using two search pruning heuristics. After each attempt to segment the sequence, it counts the number of derived objects (segments), k , in the derived sequence. If k is less than 3, the segmentation is discarded since there are too few derived objects to use for generalization. If k is more than half of the number of objects in the original sequence, the segmentation is also discarded because in this case many segments contain only one original object. Segmented sequences that survive these two pruning heuristics are passed on to Ring 2 for further processing.

One segmentation that Ring 3 always performs is the "null" segmentation—that is, it always passes the unsegmented sequence directly to the inner rings. Figure 6-5 shows a sample layout and the resulting derived layout after segmentation using the segmentation condition: $[\text{suit}(\text{card}_i)=\text{suit}(\text{card}_{i+1})]$. The derived objects (segments) are denoted by variables string_i . The negative event $[\text{suit}(\text{string}_2)=D][\text{color}(\text{string}_2)=\text{red}]$ $[\text{length}(\text{string}_2)=3]$ is obtained from the segment <5D 2D 4D>.

SPARC/E derives the descriptors COLOR, SUIT, and LENGTH to describe each derived object. The choice of which descriptors to derive involves three steps. First, LENGTH is derived whenever the segmentation transformation is applied. Second, any descriptor that is tested in the segmentation predicate (in this case, SUIT) is also derived. Third, any descriptor is derived whose value can be proved to be the same for all cards in each segment. In this case, COLOR is derived because, if SUIT is a constant, then COLOR is also a constant. Using this

The layout:

```

3H  5D  2D  7C  AC  9C  JH  6H  8H  QH  KS
   5S  4D      AH
           7S

```

The derived sequence:

description of derived object	positive or negative
[suit(string1)=H][color(string1)=red] [length(string1)=1]	+
[suit(string2)=D][color(string2)=red] [length(string2)=2]	+
[suit(string2)=D][color(string2)=red] [length(string2)=3]	-
[suit(string3)=C][color(string3)=black] [length(string3)=3]	+
[suit(string4)=H][color(string4)=red] [length(string4)=4]	+

Figure 6-5: Sample layout and segmented sequence.

segmentation, SPARC can use the DNF model to discover that the segmented sequence can be described as

[length(string1)=length(string1-1)+1]

That is, the LENGTH of each segment of constant SUIT (in the main line) increases by 1.

6.3.4. Ring 2: Parameterizing the Models

Ring 2 searches the space of parameterizations of the three basic models. Each model is considered in turn. For each model, Ring 2 develops a set of derived events based on each allowed value of the lookback parameter, L , and the number of phases parameter, P . The user can control which models should be inspected and what range of values for L and P should be investigated. By default, the program will inspect the decomposition model with $L = 0, 1$, or 2 , and the periodic model with $P = 1$ or 2 and $L = 0$ or 1 .

Specifically, Ring 2 performs the following actions depending on which model is being parameterized:

A. For the *decomposition* model with lookback parameter L , Ring 2 applies the blocking transformation to break the sequence received from Ring 3 into blocks of length L . After blocking, all of the attributes that described the original objects are converted into attributes that describe the whole block (as described in section 4 above). Furthermore, sum and difference descriptors are derived to represent the relationships between adjacent objects in the original sequence. The resulting derived events can be viewed as very specific if-then clauses of the following form.

Given an initial sequence of objects $\langle q_1, q_2, \dots, q_m \rangle$, let us look at block b_i which describes the subsequence $\langle q_{i-L}, \dots, q_{i-1}, q_i \rangle$. Let F_j denote the selectors of object q_{i-L+j} renamed so that they apply to b_i . For example, F_1 could be the selectors [suit1(b1)=H][rank1(b1)=3]—selectors that originally referred to object q_{i-L+1} . Let $d(F_j, F_k)$ denote all of the difference selectors obtained by "subtracting" event F_k from event F_j , and let $s(F_j, F_k)$ denote all of the summation selectors obtained by "summing" events F_j and F_k . For example, $d(F_0, F_1)$ could include the selectors [d-suit01(b1)=2][d-rank01(b1)=-3] obtained from "subtracting" F_1 from F_0 .

With these definitions, the derived events for the decomposition model have the form:

$$F_1 \& \dots \& F_L \Rightarrow F_0 \& d(F_0, F_1) \& \dots \& d(F_0, F_L) \& s(F_0, F_1) \& \dots \& s(F_0, F_L)$$

These derived events no longer need to be ordered since the ordering information is made explicit within the events. These events have the form of very specific if-then clauses. This facilitates the model-fitting process in Ring 1.

B. For the *DNF* model with lookback parameter L , the sequence derived in Ring 3 is blocked in a very similar manner, except that only the selectors describing q_i are retained in the description of block b_i . The derived events have the following form:

$$F_0 \& d(F_0, F_1) \& \dots \& d(F_0, F_L) \& s(F_0, F_1) \& \dots \& s(F_0, F_L)$$

These events are very specific conjuncts that are passed to the A^q algorithm in Ring 1, where they are generalized to form a DNF description.

C. For the *periodic* model with period length P and lookback L , Ring 2 performs a splitting transformation followed by a blocking transformation. First, the sequence obtained from Ring 3 is split into P separate sequences. Then each separate sequence is blocked into blocks of length $L+1$. The derived events have the same form as the events derived for the DNF model. Note that because the blocking occurs after the splitting, the lookback takes place only within a phase.

To provide an example of the function of Ring 2, Figure 6-6 shows some events from Figure 6-3 after they have been transformed in preparation for fitting to a decomposition model with $L=1$.

6.3.5. Ring 1: The basic model-fitting algorithms

Ring 1 consists of three separate model-fitting algorithms: the A^q algorithm, the decomposition algorithm and the periodic algorithm.

The A^q algorithm [Michalski and Kulpa, 1971] is used to fit the DNF model to the data. A^q attempts to find the DNF description with the fewest number of conjunctive terms that covers all of the positive examples and none of the negative examples. The algorithm operates as follows. First, a positive example, called the seed, is chosen, and the set of maximally-general conjunctive expressions consistent with all of the negative examples is computed. This set is

```

[rank1(b2)=3][suit1(b2)=H]
[parity1(b2)=odd][color1(b2)=red]
[prime1(b2)=Y][faced1(b2)=N]      =>

[rank0(b2)=9][suit0(b2)=S][parity0(b2)=odd]
[color0(b2)=black][prime0(b2)=N]
[faced0(b2)=N][d-rank01(b2)=+6]
[d-suit01(b2)=+1][d-parity01(b2)=N]
[d-color01(b2)=Y][d-prime01(b2)=Y]
[d-faced01(b2)=Y][s-rank01(b2)=12]      +

[rank1(b3)=9][suit1(b3)=S]
[parity1(b3)=odd][color1(b3)=black]
[prime1(b3)=N][faced1(b3)=N]      =>

[rank0(b3)=J][suit0(b3)=D][parity0(b3)=odd]
[color0(b3)=red][prime0(b3)=Y]
[faced0(b3)=Y][d-rank01(b3)=+2]
[d-suit01(b3)=+2][d-parity01(b3)=N]
[d-color01(b3)=Y][d-prime01(b3)=Y]
[d-faced01(b3)=Y][s-rank01(b3)=20]      -

```

Figure 6-6: Some events of Figure 6-3 transformed for decomposition $L=1$.

called a *star*, and it is equivalent to the G-set in Mitchell's [1978] version space approach. One element from this star is chosen to be a conjunct in the output DNF description, and all positive examples covered by it are removed from further consideration. If any positive examples remain, the process is repeated, selecting as a new seed some positive example that was not covered by *any* member of any preceding star. In this manner, a DNF description with few conjunctive terms is found. If the stars are computed without any pruning, then Λ^q can provide a tight bound on the number of conjuncts that would appear in the optimal DNF description with fewest conjunctive terms.

The decomposition algorithm is an iterative algorithm that seeks to fit the data to a decomposition model. The key task of the decomposition algorithm is to identify a few attributes, called *decomposition attributes*, from which the decomposition rule can be developed. A *decomposition attribute* is an attribute that appears on the left-hand side of an if-then clause of a decomposition rule. For example, the decomposition rule

```
[color(card1)=black] => [parity(card1)=odd] v
[color(card1)=red]   => [parity(card1)=even]
```

decomposes on COLOR. Hence, COLOR is the single decomposition attribute.

The algorithm uses a generate-and-test approach of the following form:

```
decompositionattributes := {}    The empty set

while rule is not consistent do
  begin

    generate a trial decomposition
    (based on positive evidence only)
    for each possible decomposition attribute

      test these trial decompositions against
      the data

      select the best decomposition attribute and
      add it to the set decompositionattributes

  end
```

The process of generating a trial decomposition takes place in two steps. First, a VL22 conjunction is formed for each possible value of the decomposition attribute. All positive events that have the same value of the decomposition attribute on their left-hand sides are merged together to form a single conjunction of selectors. This VL22 conjunction forms the right-hand side of a single clause in the decomposition rule. Within this conjunction, a selector is created for each attribute by forming the internal disjunction of the values in the corresponding selectors in the events. For example, using all of the events derived in Ring 2 for the sample layout in Figure 6-3, the decomposition algorithm generates the trial decomposition shown in Figure 6-7 for the $\text{PARITY}(\text{card}_1)$ attribute.

Since there are only two values (ODD and EVEN) for the decomposition attribute in the sequence shown in Figure 6-3, two conjunctions are formed. The first conjunction is obtained by merging all of the positive events for which $[\text{parity}(\text{card}_1)=\text{odd}]$. There are four such events. The first selector in that conjunction, $[\text{rank}(\text{card}_1)=9 \vee 4 \vee 2]$, is obtained by forming the internal disjunction of the values of $\text{rank}(\text{card}_1)$ in each of the four events.

The second step in forming a trial decomposition is to generalize each clause in the trial rule. The generalization is accomplished by applying rules of generalization to extend internal disjunctions and drop selectors. (See [Michalski, 1983] for a description of various rules of generalization.) Corresponding attributes in the different clauses of

```
[parity(card1-1)=odd] => [rank(card1)=9 v 4 v 2]
[suit(card1)=S v C][parity(card1)=even v odd]
[color(card1)=black][prime(card1)=Y v N]
[faced(card1)=N]
[d-rank(card1,card1-1)=+6 v -5 v -7]
[d-suit(card1,card1-1)=1 v 2 v 3]
[d-parity(card1,card1-1)=Y v N]
[d-color(card1,card1-1)=Y v N]
[d-prime(card1,card1-1)=Y v N]
[d-faced(card1,card1-1)=Y v N]
[s-rank(card1,card1-1)=12 v 13 v 9]

[parity(card1-1)=even] =>
[rank(card1)=J v 10 v 8 v 7]
[suit(card1)=H v D][parity(card1)=even v odd]
[color(card1)=red][prime(card1)=Y v N]
[faced(card1)=Y v N]
[d-rank(card1,card1-1)=7 v 8 v -2 v -1]
[d-suit(card1,card1-1)=0 v 1]
[d-parity(card1,card1-1)=Y v N]
[d-color(card1,card1-1)=Y v N]
[d-prime(card1,card1-1)=Y v N]
[d-faced(card1,card1-1)=Y v N]
[s-rank(card1,card1-1)=16 v 12 v 18]
```

Figure 6-7: Trial decomposition on the $\text{PARITY}(\text{card}_1)$ attribute

the decomposition rule are compared, and selectors whose value sets overlap are dropped. When these rules of generalization are applied to the trial decomposition for PARITY_1 , for example, the following generalized trial decomposition is obtained:

```
[parity(card1-1)=odd] =>
[suit(card1)=C v S][color(card1)=black]
[parity(card1-1)=even] =>
[suit(card1)=H v D][color(card1)=red]
```

This is a very promising trial decomposition. However, it has been developed using only positive evidence, and it has been generalized without considering that the generalization may have caused the rule to cover negative events. Hence, the trial decomposition must be tested against the negative events to determine whether or not it is consistent. It turns out that the generalized trial decomposition shown above is indeed consistent with the negative evidence.

After a trial decomposition has been developed for each possible decomposition attribute, the best decomposition attribute is selected according to a heuristic attribute-quality functional. The attribute-quality functional tests such things as the number of negative events covered by the trial decomposition, the number of clauses with non-null right-hand sides, and the complexity of the trial decomposition (defined as the number of selectors that cannot be written with a single operator and a single value). The chosen trial decomposition forms a candidate sequence-prediction rule.

If the candidate rule is not consistent with the data (i.e., still covers some negative examples), then the decomposition algorithm must be repeated to select a second attribute to add to the left-hand sides of the if-then clauses. This has the effect of splitting each of the if-then clauses into several more if-then clauses. For example, if we first decomposed on $\text{PARITY}(\text{card}_1)$ and then on $\text{FACED}(\text{card}_1)$, we would obtain four if-then clauses of the form:

```
[parity(card1-1)=odd][faced(card1-1)=N] => ...
[parity(card1-1)=odd][faced(card1-1)=Y] => ...
[parity(card1-1)=even][faced(card1-1)=N] => ...
[parity(card1-1)=even][faced(card1-1)=Y] => ...
```

The periodic algorithm is nearly the same as the decomposition algorithm. For each phase of the period, it takes all of the positive

events in that phase and combines them to form a single conjunct by forming the internal disjunction all of the value sets of corresponding selectors. Next, rules of generalization are applied to extend internal disjunctions and drop selectors. Finally, corresponding attributes in different phases are compared, and selectors whose values sets overlap are dropped if this can be done without covering any negative examples.

6.3.6. Evaluating the NDP rules

Once Ring 1 has instantiated the parameterized models to produce a set of rules, the rules are passed back through the concentric rings of the program. Each ring evaluates the rules according to plausibility criteria based on knowledge available in that ring. Ring 2, for example, checks to see that the rule does not predict an end to the sequence. It is assumed that a valid sequence can be continued indefinitely. Ring 3 checks the last (partial) segment to see if it is consistent with the rule. It is possible to induce a rule, using only the complete segments, that is not consistent with the final segment. Ring 4 tests the rule using the plausibility criteria for Eleusis. These criteria are:

1. Prefer rules with intermediate degree of complexity. In Eleusis, Occam's Razor does not always apply. The dealer is unlikely to choose a rule that is extremely simple, because it would be too easy to discover. Very complex rules will not be discovered by anyone, and, since the rules of the game discourage such an outcome, the dealer is not likely to choose such complex rules either.
2. Prefer rules with an intermediate degree of non-determinism. Rules with a low degree of non-determinism lead to many incorrect plays, thus rendering them easy to discover. Rules that are very nondeterministic generally lead to few incorrect plays and are therefore difficult to discover.

Rules that do not satisfy these heuristic criteria are discarded. The remaining rules are returned to Ring 5 where they are printed for the user.

7. Examples of Program Execution

In this section, we present some example Eleusis games and the corresponding sequence-generating laws that were discovered by SPARC/E. Each of these games was an actual game among people, and the rules are presented as they were displayed by SPARC/E (with minor typesetting changes).

The raw sequences presented to SPARC/E had only two attributes: SUIT and RANK. SPARC/E was given definitions of the following derivable attributes:

- COLOR (red for Hearts and Diamonds; black for Clubs and Spades)
- FACE (true if card is a faced, picture card; false otherwise)
- PRIME (true if card has a prime rank, false otherwise)
- MOD2 (the parity value of the card, 0 if card is even, 1 otherwise)
- MOD3 (the rank of the card modulo 3)

- LENMOD2 (When SPARC/E segments the main sequence into derived subsequences, it computes the LENGTH of each of the subsequences modulo 2)

Three examples of the program execution are presented: one showing the program at its best, one showing some of the shortcomings of the program, and one demonstrating weaknesses of the program. A few explanations are required. First, each rule is assumed to be universally quantified over all events in the sequence. This quantification is not explicitly printed. Second, when the value set of a selector includes a set of adjacent values (e.g., $[RANK(CARDI)=3 \vee 4 \vee 5]$, this is printed as $[RANK(CARDI)=3..5]$. The computation times given are for an implementation in PASCAL on the CDC CYBER 175.

7.1. Example 1

In this example, we show the program discovering a segmented rule. The program was presented with the following layout:

```
Main line:  AH 7C 6C 9S 10H 7H 10D JC AD
Side lines:      KD      6S      QD
                JH

continued:  4H 8D 7C 9S 10C KS 2C 10S JS
            3S      9H      QH
            6H      AD
```

The program only discovered one rule for this layout, precisely the rule that the dealer had in mind (1.2 seconds required):

```
RULE 1: LOOKBACK: 0 NPHASES: 1 PERIODIC MODEL
CRITERION=[COLOR(CARDI)=COLOR(CARDI-1)]:
PERIOD([LENMOD2(StringI)=1])
```

The rule states that one must play strings of cards with the same color. The strings must always have odd length. The CRITERION= gives the segmentation criterion that a segment is a string of cards all of the same color. CARDI refers to the I-th card in the original sequence. STRINGI refers to the I-th string in the segmented sequence. SPARC/E discovered this rule as a degenerate periodic rule with a period length, P, of 1. Actually, the rule that the dealer had in mind had one additional constraint: a queen must not be played adjacent to a jack or king. Rules containing such exception clauses cannot be discovered by SPARC/E.

7.2. Example 2

The second example requires the program to discover a fairly simple periodic rule. SPARC/E discovers three equivalent versions of it.

Here is the layout:

```
Main line:  JC 4D QH 3S QD 9H QC 7H QD
Side lines:  KC 6S      4S      10D
            7S

continued:  9D QC 3H KH 4C KD 6C JD 8D

continued:  JH 7C JD 7H JH 6H KD
```

The program discovered the following descriptions of this layout (0.49 seconds were required):

RULE 1: LOOKBACK: 1 NPHASES: 0
DECOMPOSITION MODEL

```
[FACE(CARDI-1)=FALSE] =>
  [RANK(CARDI) ≥ JACK]
  [RANK(CARDI) > RANK(CARDI-1)]
  [FACE(CARDI)=TRUE] V
```

```
[FACE(CARDI-1)=TRUE] =>
  [RANK(CARDI)=3..9]
  [RANK(CARDI) < RANK(CARDI-1)]
  [FACE(CARDI) =FALSE]
```

RULE 2: LOOKBACK: 1 NPHASES: 1 PERIODIC MODEL

```
PERIOD([RANK(CARDI) ≥ 3]
  [RANK(CARDI) ≠ RANK(CARDI-1)]
  [FACE(CARDI) ≠ FACE(CARDI-1)])
```

RULE 3: LOOKBACK: 1 NPHASES: 2 PERIODIC MODEL

```
PERIOD([RANK(CARDI) ≥ JACK]
  [RANK(CARDI) ≥ -RANK(CARDI-1)+20]
  [FACE(CARDI)=TRUE],

  [RANK(CARDI)=3..9]
  [RANK(CARDI) = -RANK(CARDI-1)+5..14]
  [FACE(CARDI)=FALSE])
```

Rule 1 is a decomposition rule with a lookback of 1. Rule 2 expresses the rule as a single conjunction. This is possible because FACE versus NON-FACE is a binary condition, and there are precisely two phases to the rule. Rule 3 expresses the rule in the "natural" way as a periodic rule of length 2.

Notice that, although the program has the gist of the rule, it has discovered a number of redundant conditions. For example, in rule 1, the program did not use knowledge of the fact that $[rank(cardi) \geq jack]$ implies $[face(cardi)=true]$, and therefore, it did not remove the former selector. Similarly, because of the interaction of the two conditions, $[rank(cardi) > rank(cardi-1)]$ is completely redundant. SPARC/E already has enough background knowledge about the meanings of its attributes to support these inferences. Additional routines need to be written to actually perform them (as is done in the INDUCE-2 program—see [Michalski, 1983]).

7.3. Example 3

The third example shows the upper limits of the program's abilities. During this game, only one of the human players even got close to guessing the rule, yet the program discovers a good approximation of the rule using only a portion of the layout that was available to the human players. Here is the layout:

Main line:	4H	5D	8C	JS	2C	5S	AC	5S	10H
Side lines:	7C	6S	KC	AH		6C		AS	
	JH	7H	3H	KD					
	4C	2C		QS					
	10S	7S							
	8H	6D							
	AD	6H							
	2D	4C							

The program produced the following rules after 6.5 seconds:

RULE 1: LOOKBACK: 1 NPHASES: 0 DNF MODEL

```
[RANK(CARDI) ≤ 5][SUIT(CARDI)=SUIT(CARDI-1)+1] V
[RANK(CARDI) ≥ 5][SUIT(CARDI)=SUIT(CARDI-1)+3]
```

RULE 2: LOOKBACK: 1 NPHASES: 1 PERIODIC MODEL

```
PERIOD([RANK(CARDI)=RANK(CARDI-1)-9]
  [RANK(CARDI) = -RANK(CARDI-1)+4,5,7,11,13,17]
  [SUIT(CARDI)=SUIT(CARDI-1)+1,2,3])
```

RULE 3: LOOKBACK: 1 NPHASES: 2 PERIODIC MODEL

```
PERIOD([RANK(CARDI)=ACE,2,8,10]
  [RANK(CARDI) = -RANK(CARDI-1)+1,8,9,10],

  [RANK(CARDI)=5..JACK][SUIT(CARDI)=SPADES]
  [RANK(CARDI)=RANK(CARDI-1)+-0..6]
  [RANK(CARDI) = -RANK(CARDI-1)+8..14]
  [SUIT(CARDI)=SUIT(CARDI-1)+0..2]
  [COLOR(CARDI)=BLACK][PRIME(CARDI)=PTRUE]
  [PRIME(CARDI)=PRIME(CARDI-1)]
  [MOD2(CARDI)=1][MOD2(CARDI)=MOD2(CARDI-1)+0]
  [MOD2(CARDI) = -MOD2(CARDI-1)+0][MOD3(CARDI)=2]
  [MOD3(CARDI)=MOD3(CARDI-1)+0]
  [MOD3(CARDI) = -MOD3(CARDI-1)+1])
```

The rule that the dealer had in mind was:

```
[SUIT(CARDI)=SUIT(CARDI-1)+1]
[RANK(CARDI) ≥ RANK(CARDI-1)] V
```

```
[SUIT(CARDI)=SUIT(CARDI-1)+3]
[RANK(CARDI) ≤ RANK(CARDI-1)]
```

There is a strong symmetry in this rule: the players may either play a higher card in the next "higher" suit (recall that the suits are cyclically ordered) or a lower card in the next "lower" suit. The program discovered a slightly simpler version of the rule (rule 1) that happened to be consistent with the training instances. Note that adding 3 to the SUIT has the effect of computing the next lower suit.

The other two rules discovered by the program are very poor. They are typical of the kinds of rules that the program discovers when the model does not fit the data very well. Both rules are filled with irrelevant descriptors and values. The current program has very little ability to assess how well a model fits the data. These rules should not be printed by the program since they are highly implausible.

8. Summary

We have presented here a methodology for discovering sequence-generating rules for the nondeterministic prediction problem. The main ideas behind this methodology are

1. the use of task-oriented transformations of the initial data and
2. the use of different rule models to guide the search for sequence-generating rules.

Four different task-oriented transformations (adding attributes, blocking, splitting into phases, and segmenting) and three models (DNF, periodic, and decomposition) have been presented.

The main part of the methodology has been implemented in the program SPARC/E and applied to the NDP problem that arises in the card game Eleusis. The performance of the program indicates that it can discover quite complex and interesting rules.

This methodology is quite general and can be applied to other nondeterministic prediction problems in which the objects in the initial sequence are describable by a small set of finite-valued attributes. The main strengths of the method are (a) that it can solve

learning problems in which the initial training instances require substantial task-oriented transformation and (b) that it can search very large spaces of possible rules using a set of rule models for guidance.

Many aspects of this methodology remain to be investigated. We have not considered NDP problems in which (a) the training instances are noisy, (b) the training instances have internal structure so that an attribute vector representation cannot be used, and (c) the sequence-generating rules are permitted to have exceptions. Application of this methodology to real world problems will probably also require the development of additional sequence transformations and rule models. Also, more heuristics need to be developed that can be used to guide the application of transformations and models.

The implementation of the methodology in program SPARC/E has demonstrated that the method can be used to discover many Eleusis secret rules. There are some shortcomings of the implementation, however. The program presently conducts a nearly exhaustive depth-first search of the possible models and transformations. Much could be gained by having the program conduct a best-first heuristically-guided search instead. Another weakness of SPARC/E is its poor ability to evaluate the plausibility of the rules it discovers. It is also not able to simplify rules by removing redundant selectors, nor is it able to estimate the degree of nondeterminism of the rule. Both of these can be implemented without too much difficulty by including inference routines that make more complete use of the background knowledge already available to the program. Finally, an important weakness of the program is its inability to form composite models. SPARC/E is not presently able to handle the NDP problem shown in Figure 1, because it involves a periodic rule in which one of the phases contains an imbedded periodic sequence (see section 5.3).

In addition to these specific problems, there are some more general problems that further research in the area of sequence-generating laws should address. First, in some real world problems, there are several example sequences available for which the sequence-generating law is believed to be the same. Such problems occur, in particular, in describing the process of disease development in medicine and agriculture. A specific problem of this type that has been partially investigated involves predicting the time course of cutworm infestation in a cornfield and estimating the potential damage to the crop (see [Davis, 1981], [Baim, 1983], and [Boulanger, 1983]). In this problem, several sequences of observations are available—one for each field—and there is a need to develop a sequence-generating law that predicts all of these sequences.

A second general problem for further research is to handle continuous processes. AI research has so far given little attention to this case.

References

- Abbott, R., "The New Eleusis," Available from the author, Box 1175, General Post Office, New York, NY 10116, 1977.
- Amarel, S., "On Representations of Problems of Reasoning About Actions," *Machine Intelligence 3*, Michie, D., (ed.), University of Edinburgh Press, Edinburgh, pp. 131-171, 1968.
- Baim, P. W., "Automated Acquisition of Decision Rules: Problems of Attribute Construction and Selection," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1983.
- Boulanger, A. G., "The Expert System PLANT/CD: A Case Study in Applying the General Purpose Inference System ADVISE to Predicting Black Cutworm Damage in Corn," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1983.
- Buchanan, B. G., and Mitchell, T. M., "Model-Directed Learning of Production Rules," in *Pattern-directed Inference Systems*, Waterman, D. A., and Hayes-Roth, F., (eds.), Academic Press, New York, 1978.
- Cohen, P., and Feigenbaum, E. A., *The Handbook of Artificial Intelligence*, Vol. III, Kaufmann, Los Altos, 1982.
- Davis, J., "CONVART: A Program for Constructive Induction on Time-Dependent Data," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, Illinois, 1981.
- Dietterich, T. G., London, R., Clarkson, K., and Dromey, G., "Learning and Inductive Inference," Chapter XIV in Vol. 3 of *The Handbook of Artificial Intelligence*, Cohen, P. R., and Feigenbaum, E. A., (eds.), 1982.
- Dietterich, T. G., and Michalski, R. S., "Inductive Learning of Structural Descriptions: Evaluation Criteria and Comparative Review of Selected Methods," *Artificial Intelligence*, 1981.
- Engelmore, R., and Terry, A., "Structure and Function of the Crayalis System," *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, 1979.
- Friedland, P. E., "Knowledge-Based Experiment Design in Molecular Genetics," Rep. No. HPP-79-29, Department of Computer Science, Stanford University, 1979.
- Gardner, M., "On Playing the New Eleusis, the game that simulates the search for truth," *Scientific American*, No. 237, pp. 18-25, October, 1977.
- Hedrick, C. L., "Learning Production Systems from Examples," *Artificial Intelligence*, Vol. 7, No. 1, pp. 21-49, 1976.
- Karpinski, J., and Michalski, R. S., "A System that Learns to Recognize Hand-written Alphanumeric Characters," *Proce Institute Automatyki, Polish Academy of Sciences*, No. 35, 1966.
- Kotovsky, K., and Simon, H. A., "Empirical Tests of a Theory of Human Acquisition of Concepts for Sequential Patterns," *Cognitive Psychology*, No. 4, pp. 399-424, 1973.
- Langley, P. W., "Descriptive Discovery Processes: Experiments in Baconian Science," Rep. No. CMU-CS-80-121, Department of Computer Science, Carnegie-Mellon University, 1980.
- McCarthy, J., "Programs with Common Sense," in *Proceedings of the Symposium on the Mechanization of Thought Processes*, National Physical Laboratory, pp. 77-84, 1958.
- McCarthy, J., and Hayes, P., "Some Epistemological Problems from the Standpoint of Artificial Intelligence," in *Machine Intelligence 4*, Meltzer, B., and Michie, D., (eds.), Edinburgh University Press, Edinburgh, pp. 463-502, 1969.
- Michalski, R. S., "A Theory and Methodology of Inductive Learning," *Artificial Intelligence*, Vol. 20, 111-161, 1983.
- Michalski, R. S., and Chilausky, R. I., "Learning by Being Told and Learning From Examples: An Experimental Comparison of the Two Methods of Knowledge Acquisition in the Context of Developing an Expert System for Soybean Disease Diagnosis," *Policy Analysis and Information Systems*, Vol. 4, No. 2, June 1980.

- Michalski, R. S., and Kulpa, Z., "A System of Programs for the Synthesis of Switching Circuits Using the Method of Disjoint Stars," *Information Processing 71*, pp. 61-65, North-Holland, 1971.
- Mitchell, T. M., "Version Spaces: An approach to concept learning," Rep. No. STAN-CS-78-711, December, 1978.
- Mitchell, T. M., Utgoff, P. E., and Banerji, R. B., "Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics," in *Machine Learning*, Michalski, R. S., Carbonell, J. G., and Mitchell, T. M., (eds.), Tioga, Palo Alto, 1983.
- Persson, S., "Some Sequence Extrapolating Programs: A Study of Representation and Modeling in Inquiring Systems," Rep. No. CS50, 1966.
- Samuel, A. L., "Some Studies in Machine Learning using the Game of Checkers," in *Computers and Thought*, Feigenbaum, E. A., and Feldman, J., (eds.) McGraw-Hill, New York, pp. 71-105, 1963.
- Samuel, A. L., "Some Studies in Machine Learning using the Game of Checkers II—Recent Progress," *IBAI Journal of Research and Development*, Vol. 11, No. 6, pp. 601-617, 1967.
- Schank, R., and Abelson, R., "Scripts, Plans, and Knowledge," *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pp. 151-157, 1975.
- Simon, H. A., "Complexity and the Representation of Patterned Sequences of Symbols," *Psych. Review*, Vol. 79, No. 5, 369-382, 1972.
- Simon, H. A., and Kotovsky, K., "Human Acquisition of Concepts for Sequential Patterns," *Psychological Review*, Vol. 70, pp. 534-546, 1963.
- Solomonoff, R. S., "A Formal Theory of Inductive Inference," *Information and Control*, Vol. 7, pp. 1-22, 224-254, 1964.
- Soloway, E. M., "Learning = Interpretation + Generalization: A Case Study in Knowledge-Directed Learning," Rep. No. COINS TR-78-13, Computer and Information Science Dept., U. Mass. at Amherst, 1978.
- Winston, P. H., "Learning Structural Descriptions from Examples," Rep. No. AI-TR-231, MIT, 1970.
- q_i or $q1$ The i -th object in an input sequence.
- q_i' The i -th object in a derived sequence.
- q_i^- An object that constitutes an incorrect extension of the sequence after object q_i .
- b_i or $b1$ The i -th block in a sequence derived by the blocking transformation.
- ph_i The i -th phase derived by the splitting transformation.
- F The starting set of attributes for a transformation.
- S The starting set of sequences for a transformation.
- Q The starting set of objects for a transformation.
- F' The set of derived attributes from a transformation.
- S' The set of derived sequences from a transformation.
- Q' The set of derived objects from a transformation.
- F_j The set of selectors describing object $q_{i,j}$ in block b_i .
- g The sequence-generating function that maps a sequence into a set of objects Q_{k+1} that can appear as continuations of the sequence.
- Q_{k+1} The set of objects that can appear as continuations of the sequence $\langle q_1, q_2, \dots, q_k \rangle$.
- P The number of phases parameter of the splitting transformation and the periodic model.
- L The lookback parameter of the blocking transformation and all three models.
- $[f_i(q_j)=r_k]$ or $[f1(qj)=rk]$ A simple selector, which asserts that feature f_i of object q_j has the value r_k .
- $[f1(qj)=r1 \vee r2 \vee r3]$ A selector containing an internal disjunction. It asserts that f_i can have the value r_1 or r_2 or r_3 .
- d prefix The d prefix on an attribute name indicates that it is a difference attribute. Hence, $D-RANK(q_i, q_{i-1})$ is equal to $RANK(q_i) - RANK(q_{i-1})$.
- s prefix The s prefix on an attribute name indicates that it is a summation attribute. Hence, $S-RANK(q_i, q_{i-1})$ is equal to $RANK(q_i) + RANK(q_{i-1})$.
- $d(F_i, F_j)$ The set of difference selectors obtained by "subtracting" selectors F_j from F_i .
- $s(F_i, F_j)$ The set of summation selectors obtained by "adding" selectors F_i and F_j .
- $\Rightarrow$ Logical implication.

APPENDIX

Notational conventions

The following notational conventions are employed in this paper. In general, lowercase letters denote objects in some sequence (q, ph, b) or index variables (i, j, k) or the lengths of sequences (m, n). Uppercase letters denote sets of objects, attributes, and so on (Q, F, S) as well as parameters of models and transformations (L, P). Small capitals denote attributes (COLOR, RANK) and their values (RED, KING).

- $\langle \rangle$ Angle brackets denote sequences of objects, e.g., $\langle 2 \ 4 \ 6 \ 8 \rangle$ and also periodic rules, e.g., $\langle color(q_i)=red \rangle, \langle color(q_i)=black \rangle$.

LEARNING FROM NOISY DATA

J.R. Quinlan
 N.S.W. Institute of Technology
 Sydney 2007 Australia

ABSTRACT

Inductive inference systems are intended to discover general classification rules from examination of a given set of examples described in terms of a collection of properties. This study looks at the effect of noise in these descriptions on the discovery of classification rules and on their accuracy. A modified form of an existing rule-building algorithm that can tolerate noisy descriptions is presented. After discussing the results of three sets of experiments, the paper makes several conjectures about the way that the classification task should be formulated when noise is expected.

Key terms: Learning, inductive inference, classification rules, decision trees, noise, classification errors, performance degradation.

INTRODUCTION

An inductive inference system is given a collection of sample objects whose class is known and postulates from these a classification rule that assigns any similar object to a class. The objects are dealt with as descriptions in terms of their properties and these may intermix measurements, Boolean and qualitative attributes. For example, a person's description might include his weight (a measurement), whether or not he is healthy (a Boolean value based on judgement), and the colour of his eyes (a subjective division into categories).

In real-world classification tasks, the description of an object will often contain errors. Some of the sources of these errors are faulty measurement, ill-defined thresholds (e.g., when is a person "tall"?), and subjective interpretation of a multitude of inputs (e.g., what criteria are used when describing a person as "athletic"?). Regardless of the source, however, these errors can be expected to affect the formation and use of classification rules in two ways. First, within the given set of objects itself, errors will tend to confuse any mechanism that looks for common properties of subsets. Suppose, for example, that the subset of red-haired people ought to be classified differently from others; this information may be difficult to discover if many red-haired people have been erroneously described as "fair" or "brunette" and vice versa. Secondly, problems may arise independently when the rule formed from

the given objects is used to classify another object. The classification rule is couched in terms of the description of an object and, if this description contains errors, the result given by the classification rule might well be incorrect also, even though the rule itself is correct.

This paper investigates the effect of description errors on both the formation of classification rules and their use to classify objects. The formalism is a simplified one in which objects are described in terms of a fixed collection of properties or attributes, each with its own small, unordered set of discrete attribute values. Classification rules take the form of decision trees that examine an object's values of some attributes in order to assign it to a class. Now, we wish to observe the effects of different levels of description errors and so we must be able to control them in some way. This has been handled here by introducing varying levels of noise into the correct description of an object. Specifically, suppose a noise level of n percent has been applied to some attribute A . Whenever an object's description is now generated, with n percent probability the true value of A for that object will be replaced by a randomly-chosen value. This process thus models description errors as the occasional non-repeatable substitution of an incorrect value for the true attribute value.

The effect of description errors is always to degrade the performance of the classification rules. In general, the decision tree produced when the given set of objects contains noise will not be the same as that produced in the noise-free case. When the tree is used to classify an object, it could then give a wrong result either because it is the wrong tree, or because the description of the object being classified is incorrect. The noise model allows the error rate in descriptions to be varied at will to see the effect on the performance of the rule, measured in terms of the proportion of occasions on which it gives the correct classification of an object.

The paper first describes an existing top-down algorithm for constructing classification rules in the form of decision trees and shows how it has been modified to deal with erroneous descriptions of this type. Three experiments are then discussed in which the level of noise applied to attributes is varied and the degradation of performance of the classification rule is observed.

The results of these experiments suggest several conjectures relevant to induction with noisy data.

ALGORITHM FOR DEVELOPING CLASSIFICATION RULES

The basic aim of an induction system is to construct a rule for classifying objects from knowledge of a training set of objects whose classes are given. In our formalism all objects are described by a fixed collection of attributes, each with its own set of discrete values. Each object belongs to one of two classes, denoted 0 and 1 respectively, although the extension to more than two classes is straightforward. A rule takes the form of a decision tree: each interior node consists of a test of an attribute with one subtree for every possible value of that attribute, and each leaf signals an outcome of the classification rule.

If there are no two objects that have the same value for each attribute and yet belong to different classes, we say that the attributes are adequate for the classification task. Previous work [Quinlan, 1983] reported on an induction system called ID3 developed for this case. ID3 constructs a decision tree using a top-down divide-and-conquer approach: select an attribute, divide the training set into subsets characterized by the possible values of the attribute, and follow the same procedure recursively with each subset until no subset contains objects from both classes. These single-class subsets correspond then to leaves of the decision tree and can be labelled with that class.

The method depends for its practicality on making a good choice of attribute to test at each stage. ID3 takes an information-theoretic approach as follows. For a set S of objects, the information required for a classification rule is given by

$$M(S) = -p \log_2 p - (1-p) \log_2 (1-p)$$

where p is the proportion of objects in S that belong to class 1. Let A be an attribute with n possible values $A_1, A_2, \dots, A_n$. If attribute A were tested, S would be partitioned into subsets $S_1, S_2, \dots, S_n$ so that S_i contained all objects from S with value A_i of A . The expected information requirement for a classification rule of this form can be expressed as

$$B(S, A) = \sum_{i=1}^n W_i \cdot M(S_i)$$

where the weight W_i is the proportion of the objects in subset S_i . When given a training set S containing objects from both classes, ID3 bases the decision tree for S on an attribute A chosen to minimize $B(S, A)$ and continues recursively.

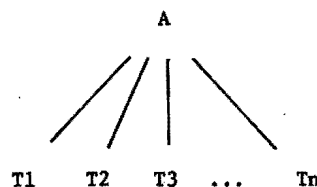
The presence of description errors complicates this situation somewhat. If the descriptions of the objects contain noise, it may be that the attributes are no longer adequate for the classification task. That is, we may end up with one or more training (sub)sets containing objects

with identical descriptions but different classes, and no amount of testing of attributes will produce single-class subsets. Instead of being able to give a categorical result for the classification rule, the best we may be able to do is return a probability that a given object belongs to a class. Rather than being labelled with a class, then, each leaf of the decision tree will contain the probability that objects corresponding to that leaf belong to class 1, with values 0 and 1 in those cases where a categorical decision is still possible.

In building a decision tree, we may now elect to stop testing attributes when it appears that no further tests are relevant to its class. The generalized procedure for building a decision tree for the set S of objects can be stated as follows:

If $M(S) \leq B(S, A)$ for all attributes A , the decision tree for S is the leaf p , where p is the proportion of objects in S that belong to class 1.

Otherwise, choose an attribute A that minimizes $B(S, A)$ and return as the decision tree



where T_i is the decision tree for the subset S_i of S whose objects have the i th value of the selected attribute A .

Although it looks reasonable, this approach leads to nonsensically large decision trees. To illustrate the problem, let S be a training set containing objects of both classes and let us suppose that no further testing of attributes will lead to a better decision tree for S . Let A be a random binary-valued attribute. Unless the proportion of class 1 objects with each value of attribute A is exactly the same as the proportion of class 1 objects in all of S (a most unlikely event, statistically speaking), $B(S, A)$ will be less than $M(S)$ and it will appear that testing attribute A is worthwhile.

The obvious expedient of requiring that $B(S, A)$ be less than $M(S) - h$ for some threshold h does circumvent the problem. This introduces another complication, though, because values of h large enough to prevent irrelevant attributes being tested also manage to exclude useful tests. As a result, using a threshold causes the procedure to generate trees that make more classification errors than previously when the training set contains no noise.

The solution to this problem comes from the chi-square test for statistical independence. Consider again a set S of m objects and an

attribute A with n values. Let $T[i,c]$ denote the number of objects in S with value i of attribute A that belong to class c. If an object's class is independent of its value of attribute A, the number $T'[i,c]$ that we would expect to find is given by

$$T'[i,c] = T[i,*] \cdot T[* ,c] / m$$

where * denotes summation over a parameter, e.g. $T[i,*]$ is $T[i,0] + T[i,1]$. Then the statistic

$$\sum_{i=1}^n \sum_{c=0}^1 (T[i,c] - T'[i,c])^2 / T'[i,c]$$

is approximately chi-square with n-1 degrees of freedom. If this value exceeds a tabulated value, the hypothesis that an object's class and value of A are independent over S (and therefore that A is irrelevant for classifying S) can be rejected with known confidence [Hogg and Craig, 1970, pp 313ff].

This test is incorporated into the tree-building procedure by stipulating that no attribute A will be considered for testing unless we can reject, with a high degree of confidence, the hypothesis that the value of A is independent of the class over the given set S. There is one minor difficulty in that the chi-square test is unreliable for very small values of the expectations T' , so the common practice of using the test only when all values of T' are at least 4 has been followed. For these experiments, the chi-square test was used with a confidence level of 99 percent and resulted in a dramatic reduction in the size of the tree when noise levels were high, with no loss of accuracy in the noise-free case.

THE PROBLEM

Since this study was intended to be an empirical investigation with practical applications, it was important to use a real (as opposed to invented) classification task -- an artificial one would almost inevitably contain built-in assumptions that might influence the results. For this reason, the data used were taken from an existing difficult classification problem for a chess end-game. The universe consists of 551 objects described in terms of 39 binary-valued attributes. About 74 percent of the objects belong to class 1 and the smallest known correct decision tree for the task contains 175 nodes. One of the attributes is redundant in the sense that a correct rule can be constructed without reference to it.

EXPERIMENT 1

The first experiment investigated the effect of noise on forming and using a decision tree when all objects were included in the training set. In each trial, the whole collection of objects was corrupted to a predetermined level and used to form a decision tree. A different corrupted copy of all objects was then generated and each object classified by the decision tree.

The results obtained were compared with the known classes of the objects and a mean error computed as the sum of the absolute difference between the real and computed classes (remembering that the computed class can be a probability such as 0.8) expressed as a percentage of the total number of objects. Each experiment was repeated 20 times with different corruptions of the training set and test cases to give a more reliable average together with an indication of its variance.

The form of the experiment was as follows. Each attribute in turn was individually corrupted with noise levels 5, 10, 15, 20, 30, ... 100 percent, giving in all twelve average errors showing the effect of noise in that attribute alone. The same noise levels were then applied to all attributes simultaneously. Finally, the class information in the training set was corrupted to the same levels. These trials thus gave an indication of misclassification error under twelve levels of noise for each attribute in isolation, for all attributes together, and for the class information in the training set.

As would be expected, the effects of the noise vary markedly from attribute to attribute. Figure 1 shows the degradation of performance as noise is added to attribute 2 (the most sensitive), attribute 32 (the least sensitive and, as it happens, the redundant attribute), and the average over the 39 attributes. This agrees with our intuition that, in classification problems, some attributes are more important than others. In fact, we can define the importance of an

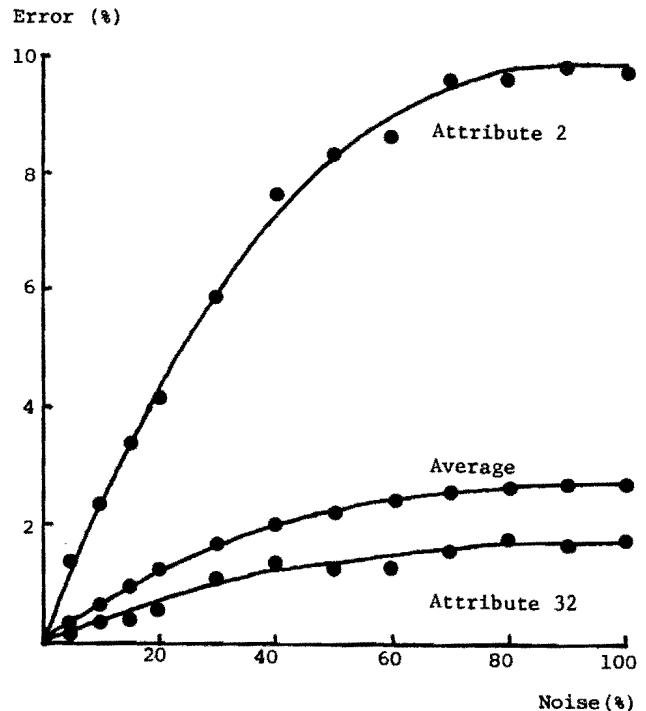


Figure 1. Classification error as a function of noise: attributes 2, 32, and average for all attributes (Expt. 1)

attribute without reference to noise as the average classification error that results if the attribute in question is deleted altogether from the data, so preventing its use in the decision tree. Under this definition the importance of attribute 2 is 8.0 percent, attribute 32 is 0 (since it is redundant), and the average of all attributes is 1.0 percent. The ranking of attributes by importance agrees well with their ranking by error caused when the attribute is corrupted by 100 percent noise. Notice that, in Figure 1, the error with 100 percent noise for attributes 2, 32 and the average is very close to their respective importances plus 1.8 percent.

The results shown in Figure 1 take the form of saturation curves. If each point is normalized by being expressed as a fraction of the maximum error observed at any noise level, the three cases seem to fit a single curve as demonstrated by Figure 2.

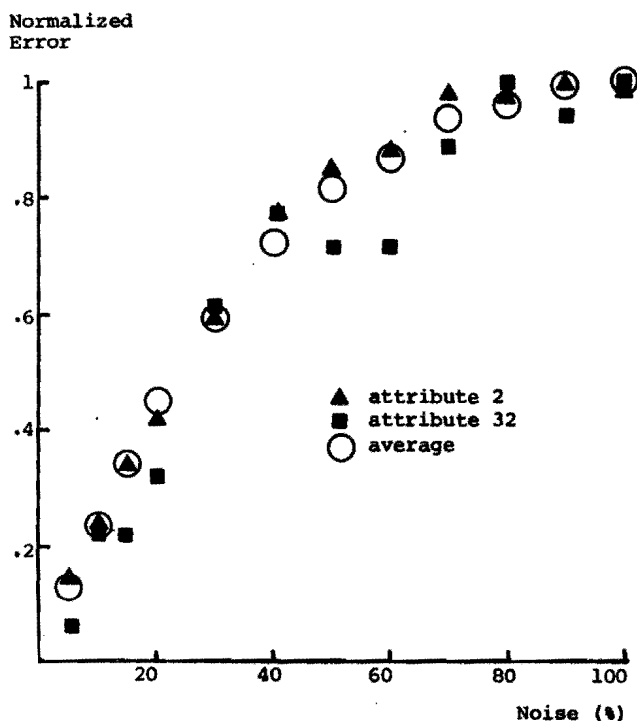


Figure 2. Normalized error for single-attribute noise (Expt. 1)

When noise is used to corrupt the class information in the training set, quite a different picture emerges. If class is treated as another attribute, its importance would necessarily be 50 percent for every two-class classification task. (This is because, if the class information in the training set is replaced by a random variable, the resulting classification rule would also be random and so would be expected to be correct half the time.) Figure 3 shows experimental agreement with this result, but also demonstrates clearly that the degradation varies nearly

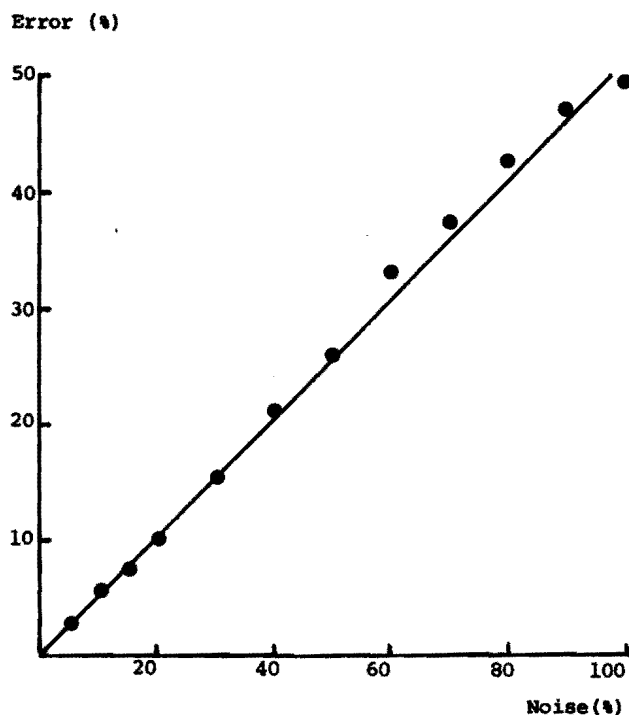


Figure 3. Classification error from noise in class information (Expt. 1)

linearly with noise in the class information.

Figure 4 shows the effect of noise applied simultaneously to all attributes. Notice that the error does not tend to 50 percent. It might seem that, if the description of all objects was random, we would get the same result as destroying class information. But in this case the training set still contains class frequency information

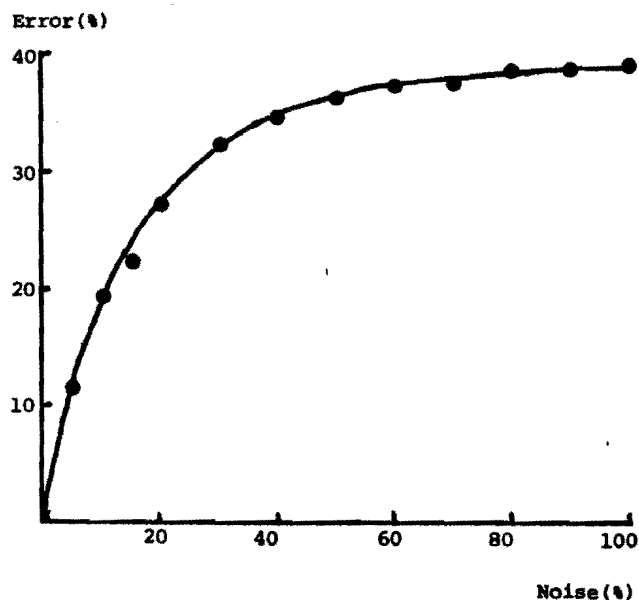


Figure 4. Classification error from noise in all attributes simultaneously (Expt. 1)

which will be reflected in the resulting decision tree. Since 74 percent of the 551 objects belong to class 1, the decision tree derived with random descriptions should give an error of $2 \times .74 \times .26$ or about 38.5 percent, as reflected in the figure. After an initial increase, the size of the decision tree decreases as noise is increased, averaging only 2 nodes when the descriptions are completely random, i.e. when all attributes are corrupted simultaneously with 100 percent noise. This demonstrates the utility of the chi-square test mentioned earlier as, without it, the average number of nodes for decision trees obtained from these random descriptions exceeded 500 nodes, even though they gave a similar average error. A saturation effect is relevant here as well, since the degradation for a given noise level is less than the sum of the errors caused by that level of noise in individual attributes.

EXPERIMENT 2

The first experiment treated noise as a ubiquitous commodity, applying both to the construction and use of classification rules. Since there was noise in the training set, the decision tree itself contained errors, and further errors were introduced as a consequence of faulty attribute values when the tree was used to classify an object. The second experiment was designed to separate the effects of noise on forming a classification rule and on using that rule. A correct decision tree was first constructed using all objects in their original, noise-free state. This correct tree was then used to classify objects corrupted in a manner similar to the previous experiment.

As before, each trial was repeated 20 times. The same twelve noise levels were applied to each attribute individually and then to all attributes together. The measure of degradation was again the average classification error over all 551 objects.

Once more, the effect of noise varied markedly from attribute to attribute with good correlation between the sensitivity of attributes to noise and their importance as defined above. Attribute 2 showed most degradation, and the effects of noise on this attribute and the average over all attributes appears in Figure 5. No saturation effect is evident, degradation increasing linearly with the noise level.

The broken lines on Figure 5 show the corresponding degradations for attribute 2 and the average obtained from the first experiment. Very surprisingly, the degradation in classification performance when noise is excluded from the training set becomes considerably greater in both cases as the noise reaches high levels. The effect is not uniform, though, and the maximum degradation experienced with the less important attributes is somewhat lower than before.

This experiment demonstrates that noise has a different effect when the underlying

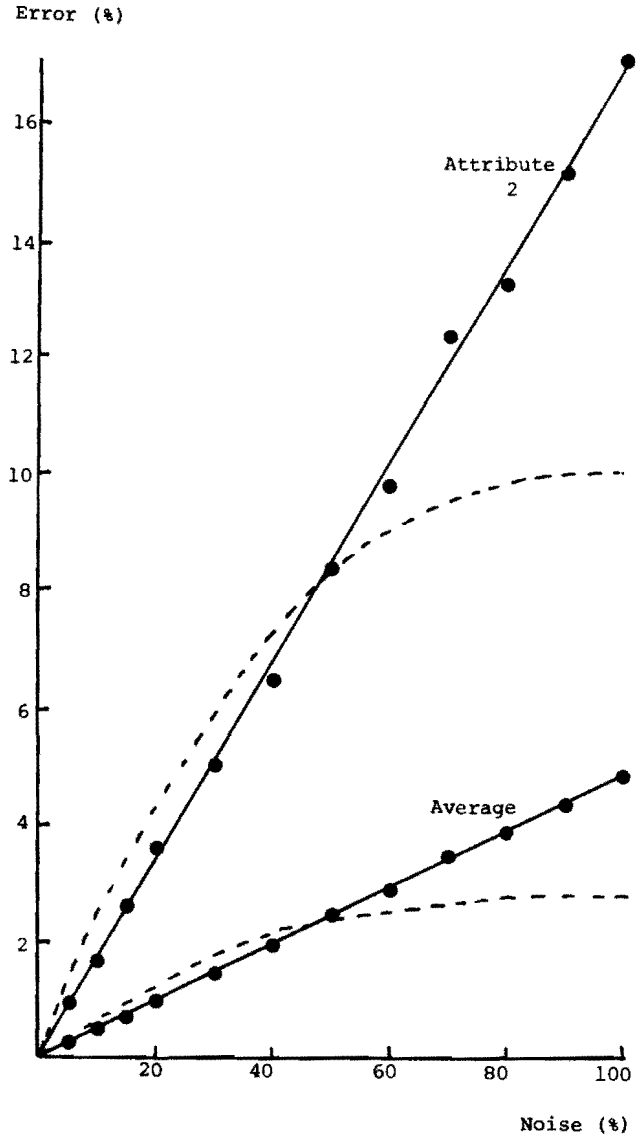


Figure 5. Classification error from single-attribute noise (Expt. 2)

classification rule is correct than that observed when the rule itself was formed from a noisy training set. The form of the degradation is different (linear versus saturation) and, even though the classification rule is correct, higher levels of noise produce a significantly higher average misclassification rate than previously.

EXPERIMENT 3

The first two experiments both assumed that the whole universe of 551 objects was available for construction of a classification rule. This third experiment was intended to check that the same results applied if only a subset of the objects was used as a training set. In this case, the resulting decision tree was tested on both known objects (those in the training set) and

unseen objects that were not taken into account when the tree was formed.

Each trial was structured as follows. A subset containing 111 of the 551 objects was selected randomly (a different subset each time) and corrupted by the imposition of noise. A decision tree was formed using the 111 corrupted objects as a training set. A new corrupted copy of all 551 objects was then generated and each object tested using the decision tree. Each trial was again repeated 20 times. The corruptions used mimicked those of the first experiment: 5, 10, ...100 percent noise applied to each attribute in turn, all attributes simultaneously, and to the class information in the training set.

Even when the training set is noise-free, the fact that it is not complete means that the decision tree generated from it will not correctly classify all objects. There is thus a base error before any effects of noise, found to be 13.2 ± 0.8 percent (or, to put it another way, a classification rule formed from about 20 percent of this universe correctly classifies about 87 percent of the universe).

There was a considerably higher variance on the results for these experiments and correspondingly poor curve fits. The effect of noise on attribute 2 again produced the highest degradation, with a peak value of 21 percent. The results of noise on the average single attribute, all attributes together, and the class information appear in Figure 6. Generally speaking, the findings appear similar to those of the first experiment except that the values are displaced by the base error. Notice that, as with the first experiment, the error due to class noise approaches 50 percent linearly while the error due to noise in all attributes together saturates towards 39 percent.

FINDINGS

The results of the experiments, especially the first and second, suggest several policy-level implications for the use of inductive inference machinery in noisy environments. These are best described as conjectures because their applicability across classification problems and across rule-inducing systems has not yet been established.

- (1) It is important to eliminate noise regarding class membership of the objects in the training set.

The importance of the class information is 50 percent, as discussed under the first experiment. This is much higher than the importance of any attribute in this example -- the maximum value was 8 percent for attribute 2. Experiment 1 confirmed that noise in the class information degrades performance linearly, and in particular the average classification error should then be half the class information noise rate even when all objects are included in the training set.

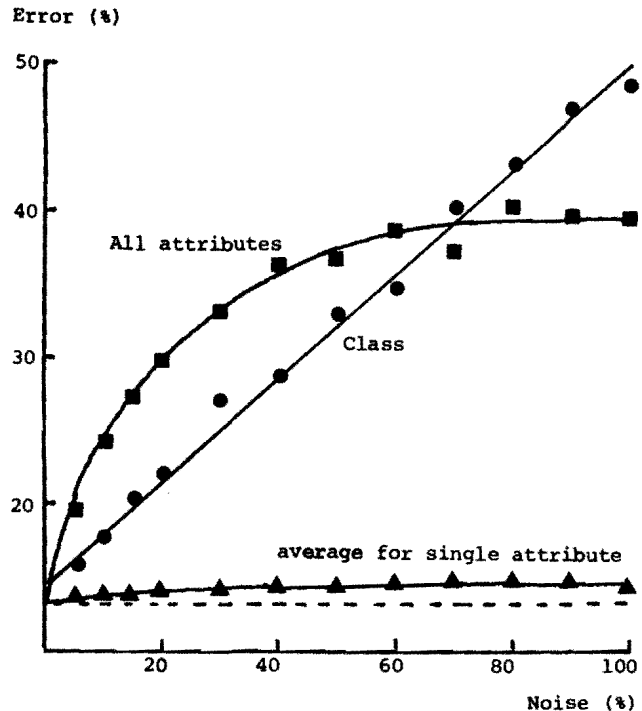


Figure 6. Classification errors (Expt. 3)

- (2) It is not worthwhile expending effort to eliminate noise from the attribute values of objects in the training set if there is going to be a significant amount of noise when the resulting classification rule is used in practice.

This observation comes from Figure 5. If the noise in the average attribute is greater than 45 percent, the performance of the correct decision tree (formed from a noise-free training set) falls below that of the tree obtained when the same level of noise was also present in the training set, and a similar result holds for the most important attribute (attribute 2). It would seem that the tree-forming algorithm tends to avoid using noisy attributes, but this feature is nullified if the training set is noise-free and so gives no indication of the noise to be expected in practice. Even with the lower levels of noise, the performance of the correct tree is only marginally superior to that of the tree formed from the training set containing noise.

- (3) We are better off dispensing altogether with noisy, less important attributes.

Consider Figure 1 again. Since attributes 2 is redundant, its importance is zero. That is, if we exclude this attribute altogether, the decision tree will still perform correctly. On the other hand, including the attribute with even slight levels of noise allows the attribute to be used in the decision tree and leads to classification errors when it is used. Again, the average

importance of the attributes is 1 percent, so dispensing with this "average" attribute will cause only 1 percent classification errors. The figure shows that we would produce fewer classification errors if this average attribute were excluded when its noise level exceeded 15 percent. Finally, we would even improve performance if we excluded attribute 2 with its highest importance (8 percent) if the noise level in this attribute exceeded 46 percent. In short, the less the importance of an attribute, the more readily we should discard it altogether when it is corrupted by even low levels of noise.

- (4) The payoff in noise reduction increases with the importance of the attribute.

All the experiments have confirmed a strong relation between the degradation in performance as an attribute is corrupted and the importance of that attribute. Remember that "importance" of an attribute is not a subjective term, but a value that can be computed over any given training set, noisy or otherwise. Therefore, if additional resources were available to minimize errors in describing objects, the importance model would indicate the attribute to which the extra effort could most profitably be applied.

- (5) Noise is less of a problem for approximate classification rules.

A classification rule is approximate if it is simpler than the corresponding correct rule. This can come about either because it was formed from a training set containing only a small fraction of all objects or because there is a substantial amount of noise in one or more attributes. For example, the decision tree formed from 111 of the 551 objects contained 49 nodes, about the same as the tree formed from all objects when all attributes were corrupted with 50 percent noise. Figures 4 and 6 show that the marginal performance degradation with increasing noise in both cases is significantly less than the corresponding marginal rates for the correct decision tree. If either the necessity of producing a rule from a small training set or of using several inherently noisy attributes limits the classification accuracy of the rule, noise in the other attributes will not result in much additional degradation (but see 1 and 3).

CONCLUSION

The work reported here falls into three parts. First, a tree-building procedure has been developed that can cope with noisy data. Second, the procedure has been implemented (in C) and extensive trials carried out on a real classification problem with controlled addition of noise. Finally, as a consequence of the experimental results obtained in these trials, conjectures have been advanced about the use of noisy attributes and the expected benefits of reducing noise.

This study has been almost entirely empirical. The next step is obviously to try to understand, at a more abstract level, some of the observed phenomena, e.g. the different degradation curve shapes for noise in attribute and class information. It would also be interesting to find out whether similar results are obtained with alternative systems for constructing classification rules.

ACKNOWLEDGMENTS

I would like to thank John Potter of N.S.W.I.T. for several helpful discussions on the statistical aspects of the modified rule-building algorithm.

REFERENCES

- Hogg, R.V. and Craig, A.T. Introduction to Mathematical Statistics (3rd Edition), New York: Macmillan, 1970.
- Quinlan, J.R. "Learning efficient classification procedures and their application to chess endgames", in Machine Learning, Michalski, Carbonell and Mitchell (Eds.), Palo Alto: Tioga Press, 1983.

INDUCTIVE RULE GENERATION IN THE CONTEXT OF THE FIFTH GENERATION

Donald Michie
Machine Intelligence Research Unit
University of Edinburgh
Edinburgh EH8 9NW

ABSTRACT

The Japanese Fifth Generation Computer Systems Report (Moto-oka, 1982), setting out the explicit goal of dominating information technology world-wide, has brought home to countries further advanced in most of these skills than the Japanese how essentially fragile is their lead.

This paper advances some propositions which are in danger of being overlooked as we rush into action.

PART 1 : SEVEN PROPOSITIONS

It is convenient to list the following propositions concerning Intelligent Knowledge Based Systems (IKBS).

Proposition 1.

With the possible exception of image-processing, VLSI research is relevant to IKBS only in so far as VLSI design-automation offers to IKBS an interesting and important application area.

Proposition 2.

Novel architectures are usually conceived as departing in kind from the von Neumann scheme. But even quantitative shifts of emphasis (e.g. towards very large cheap memories) may introduce qualitatively new issues.

Proposition 3.

The distinction between languages and relational database systems is becoming artificial.

Proposition 4.

Language debates have so far ignored IKBS' most pressing requirement, namely a means of generating programs that are analysable and executable by machine and human brain alike.

Proposition 5.

For every feasibly computable function it is often assumed that a compact specification in some form of logical language is always what the program-writer (human or machine) really needs. But it is sometimes possible to generate an extensional specification which offers him/it an entire new way of proceeding.

Proposition 6.

Automating the construction of knowledge-bases is pre-requisite to the viability of any future IKBS industry. Feasibility of computer induction has been shown in more than one laboratory. Vigorous follow-up is indicated.

Proposition 7.

IKBS will find itself end-stopped unless machine-executable concept-description languages are implemented and made available, based on a new view of programming: the Principle of Symmetry.

We now venture a (limited) definition of IKBS:

"Intelligent knowledge-based systems are capable of improving their own knowledge bases *via* precept, example and discovery."

As a human counterpart for the IKBS we select the apprentice, - an abnormally articulate one, able to expound the master's teaching, once assimilated, better than the master can.

PART 2 : INDUCTIVE LEARNING

A number of computer programs have been developed which "learn" by generating rules from examples, i.e. by induction (Michalski and Chilausky, 1980; Quinlan, 1979, 1982; Shapiro and Niblett, 1982; Niblett and Paterson, 1982; and papers in Michalski, Carbonell and Mitchell, 1983).

As a result of a long-established and sustained interest in machine learning (see the *Com-*

puter Journal, Michie, 1982a) the Edinburgh group has become a practised "machine-induction shop", one of only two. The other is R.S. Michalski's group in the University of Illinois. We can now routinely synthesise machine-executable descriptions (in the form of PASCAL programs) for classification tasks too complex for human experts either to program, to verbalise complete strategies for, or even reliably to perform. According to the style adopted, the machine-generated descriptions can be made to take either "Martian" or human form. In the second case, in spite of its synthetic origin, we call a machine-made description a "concept". A recently developed programming technique (Michie and Shapiro, 1982) has added the following capability: when a machine-made concept is "run" on a trial case, the system not only classifies the case but also displays an explanatory narrative in intelligible English as to how it arrived at the classification. This is a help both to the end-user and to the knowledge-engineer himself.

A brief description of methods and principles used by one of the Edinburgh learning programs, ACLS, is given below. The program is a development from Quinlan's ID3, in turn derived from Hunt, Marin and Stone's (1966) CLS (Concept Learning System).

ACLS (Analog Concept Learning System)

A "training set" of examples of classifications within a chosen field is presented to ACLS as a list of records. All fields of the record except the last are entered with the values of those attributes considered by the user to be relevant to the classification task. These are known as "primitive attributes". The last field is entered with the name of the decision class to which the record is to be assigned (*e.g.* CIRCLE, ELLIPSE, TRIANGLE, SQUARE, POLYGON, OTHER; or MALIGNANT, BENIGN, IMAGINARY; or ALLOWED, DISALLOWED, SPECIAL-CASE; or just TRUE, FALSE).

From these example records, ACLS derives a classification rule in the form of a decision tree, branching according to the value of the attributes. ACLS also generates a PASCAL conditional expression logically equivalent to the decision tree, and this PASCAL code can be run to classify new test examples. Whenever a new example is found which refutes the current rule, ACLS can be asked to restructure the rule so as to accommodate the new case, and to display, output or store it as before.

The set of attributes should be chosen by an expert in the given problem domain to be sufficient for classifying the data. In choosing the set of attributes the maxim is: if in doubt, put it IN. Any attribute which is in reality redundant will be set on one side by ACLS and not included in the tree (or corresponding PASCAL code) which it generates.

IT IS NOT NECESSARY THAT THE EXPERT SHOULD BE ARMED WITH A CLEAR MENTAL PICTURE OF THE RULE WHICH HE HIMSELF USES WHEN PERFORMING THE GIVEN TASK.

His role is simply to supply the list of primit-

ives and to act as an oracle by assigning example data to what he considers to be their correct classes. ACLS observes his behaviour as a classifier, and from this constructs the simplest rule it can, using the primitives supplied, which will assign the same example data to the same classes as he does. As the training set grows with the addition of fresh examples, so the ACLS-synthesised rule grows more sophisticated. As it does so it tends to approximate to a form on which the expert is likely to comment: "That looks like the way I think I probably do it".

THE EXPERT IS THUS ENABLED TO TRANSFER TO THE MACHINE A JUDGMENTAL RULE WHICH HE ALREADY HAD IN HIS HEAD BUT HAD NOT EXPLICITLY FORMULATED.

We list below four closely related questions under investigation with the aid of ACLS and similar programs:

- (1) the possibility that a learning program, such as the one described above, can be used to extend the mental grasp of the human user;
- (2) the use of the machine-synthesised programs as instructional texts;
- (3) measurement of the "knowledge" (see later) contained in machine-synthesised programs;
- (4) development of a human-intelligible "concept description language" as a mandatory vehicle for output from induction systems.

For a number of reasons, chess is specially qualified as a test domain for experiments of this kind. However, in order to confirm the generality of observed results, after basic work has been done in this measurable "test-tube" world, ACLS-type programs can be used to investigate knowledge enhancement in the domain of high school algebra, robot vision, industrial engineering fault diagnosis, chemistry, etc. Here, properly controlled experiments are less possible, so that it is desirable that knowledge enhancement in the context of chess be looked at before use of this powerful, but as yet poorly understood, instrument spreads too far into social practice. In this connection the US Department of Justice's Federal Bureau of Investigation and the US Army's Construction Engineering Research Laboratory have already purchased and installed version-1 ACLS packages for the IBM PC. This version has no built-in constraints of the type envisaged in item (4) and hence could in theory be used as a generator of potentially dangerous software in the sense of Kopec and Michie (1983).

Knowledge measurement

Tests using an information theory approach to measure the knowledge-content of machine-produced ("synthetic") programs are under way in Illinois and Edinburgh (Michie, 1977, 1982b). Relevant theoretical work has also recently started at Stanford (Yao, 1982).

Use is made of machine-oriented definitions, such as the "knowledge-content" of a program, "computational advantage", "penetration", "grasp"

and rate of knowledge acquisition. "Difficulty" of a problem can also be expressed within the same numerical calculus.

The current objective is to build the measurement system referred to above into rule-synthesis programs. These programs will then be enabled not only to produce rules which can be executed by machine but at the same time will be able to give the user a numerically based evaluation of each rule as it is synthesised. Recent results have suggested the possibility that the very act of synthesising and displaying rules in this style may help a trained practitioner in the given domain greatly to extend his skill and grasp.

In the perspective of knowledge-measurement alternative categories of solution program are possible:

- (1) too calculation-intensive to be machine-feasible
- (2) too store-intensive to be machine-feasible
- (3) machine feasible: too calculation-intensive to be brain-feasible
- (4) machine-feasible: too store-intensive to be brain-feasible
- (5) machine-feasible and potentially brain-feasible.

Knowledge engineering is the art of writing programs to fall into category (5), or, more precisely, that sub-category of (5) which is not only potentially brain-feasible by reason of its store-time trade-off properties, but also satisfies certain necessary structural properties (see later). Only programs of category (5) are sufficiently memory-sparing to be understood and sufficiently processor-sparing to be hand checkable. They lie within the human window.

Can machine optimality be combined with brain-feasibility? This is determined in each case by the shape of the trade-off curve of the given function. No general guarantee can be given that the two goals can be combined in any particular case. Moreover, from the point of view of brain feasibility, there are other problems.

"Super-programs"

Recent results have shown that programs constructed by systems such as Quinlan's ID3 can be, in one sense, "super-programs" and at the same time quite incomprehensible to people. By a super-program we mean one that is at least twice as attractive in terms of execution cost as the best which could be written by a programmer.

The Figure shows the execution costs on a CYBER 175 for three different programs written to solve the same difficult problem. The first implements a standard algorithmic approach. The second was the fastest pattern-coded solution which J.R. Quinlan, author of ID3 and also an outstanding programmer, could achieve after many months of work. The third was a machine-generated program, produced

by ID3 itself, equipped with a file of primitive descriptors and a file of example data. The last clearly qualifies as a super-program (see figure). Further, however hard they try, chess experts cannot understand it. Even though it constitutes a complete and correct description, it does not qualify as a concept description.

Work triggered by this observation has shown the reason for this. It lies in the way the decision tree generated by ID3 (or the corresponding PASCAL expression) is structured. We have therefore developed a method known as "structured induction". This is a hierarchical approach in the sense that the rules, or concepts, generated by the program using the original primitive attributes, themselves become the primitives for the next pass through the data (Shapiro and Niblett, 1982). Shapiro has subsequently applied the approach successfully to the ending King and Pawn *versus* King and Rook (Pawn on a7), generating a complete and self-documenting classifier for a space of over 200,000 positions, using some 350 expert-supplied examples. Since human experts are not self-documenting, no codified classificatory theory of the domain previously existed. The classifier thus constitutes a substantial and original contribution to end-game theory.

The method described constrains the output of the inductive generator to take the form of decision structures which people can recognise as concepts. The constraints have been embodied in the syntax of a rule language CDL-1 (Concept Description Language - 1), for which a compiler is now operational on the VAX-750.

<u>Lost-3-ply experiments</u>	
(CDC Cyber 72 at Sydney University)	
49 attributes	
(mixture of	35 pattern based
	4 low-level predicate
	10 high-level predicate)
715 distinct instances	
177-node decision tree found in 34 seconds	
Classification method	CPU time (msec)
Minimax search	285.0
Specialised search	17.5
Using decision tree	3.4

Upper part of Figure relates to the inductive synthesis of a recogniser for the "lost-3-ply" property in the chess ending king and knight v. king and rook.

Lower part compares run-time performance of two hand-made and one machine-synthesised recogniser.

(from Quinlan, 1982)

PART 3 : A KNOWLEDGE REFINERY

Expert systems now have commercial promise, surprisingly in view of the limitations of method discussed in the main paper. In one or two cases they are even making money. A system of over 2,000 rules called RI is used at Digital Equipment Corporation for configuring customers' needs (see McDermott, 1982). It already outperforms their best technical salesmen. But in knowledge refining, the focus is not on the product which expert systems were originally designed to deliver, namely interactive advice in conversation with a client, but on the unexpected by-product, the finished knowledge base itself. Expert system languages can be used:

- to get knowledge into the machine
- to test it
- to fill gaps
- to extend it
- to modify it.

Finally the knowledge can be put back into the human world in unrecognisably improved shape. This important observation was first made in the following manner:

Ivan Bratko, a Yugoslav computer scientist and chess master, was on a visit to Edinburgh. He chose for study an elementary piece of chess knowledge, namely, how to mate with king and rook against king, from any legal starting position. Although an elementary mate, it is not easy to program, taking (unless artificial intelligence methods are used) approximately 2-3 months of a graduate student's time. Furthermore, the resulting program is never completely correct in its play.

Using one of the Edinburgh advice languages, Bratko was able to complete the programming in two weeks. Subsequently he proved, both exhaustively by computation and also by symbolic proof, that the program was complete and correct. He then translated the program rules back into English. This produced six rules only: they were complete and correct, unlike grandmaster codifications which turn out to contain errors in those cases which have been tested by exhaustive computation (Clarke, 1977; Bratko, 1978; Kopec and Niblett, 1980), and moreover to consist mainly of gaps.

The improvement in the knowledge representation when back-translated from an advice program was unexpected. However, when the possibility was drawn to the attention of workers in other laboratories, they confirmed that they too were able to see indications of the same phenomenon. For example, MYCIN, developed in Feigenbaum's laboratory, was the first expert system of the modern type (Shortliffe, 1976). Although its knowledge is not broad enough to make the program of significant utility to clinicians, the system is heavily used by medical students, who find the knowledge-base more complete and easier to follow than a text book.

The following are some other cases where back-translation from an advice program has produced an improved text for the human learner.

Internal medicine

The internal medicine project is a collaboration between Dr. Jack Myers, the equivalent of a grandmaster in internal medicine, and a computer scientist, Harry Pople (Pople *et al.*, 1977). The aim is to codify Myers' clinical knowledge in the form of an expert system. Clinicians are already using print-outs of parts of the knowledge-base.

Organic chemical synthesis

In organic chemical synthesis at the University of California, organic chemists use the knowledge base from Wipke's program SECS (Wipke, 1977). In the domains covered by the program, they regard it as superior to conventional sources.

More striking cases occur where the system has inbuilt learning capabilities. When using these programs, the expert conveys his concepts to the system by means of examples and not through direct programming, just as he would if communicating with a student or colleague. The following two examples come into this category.

Soybean pathology

Until recently, the best classification of soybean diseases was that by the plant pathologist, Dr. Jacobsen, who collaborated in a study directed by R.S. Michalski in the University of Illinois (Michalski and Chilausky, 1980). Michalski's program, AQ11, is capable of generalising over examples: thus, in this case the machine was part-author of the resulting classificatory system, which gave 99% correct allocation of diseased plants as against approximately 83% for the old Jacobsen system. With great difficulty, using the machine to test, recycle and debug his theory, Jacobsen refined his own system so that it achieved a 93% success rate. Unable to improve it further, he then decided to accept the machine-generated taxonomy. He now uses this as reference material in his work.

Mass spectroscopy

This is the domain of the DENDRAL project at Stanford. The inductive learning module of the program, Meta-DENDRAL, produced rules (previously lacking) for the interpretation of mass spectra of the family of organic chemicals known as the mono- and poly-keto-androstanes. These rules formed the basis of a paper subsequently published in the chemical literature (Buchanan *et al.*, 1976).

The question arises: does a different program have to be written for each of these application areas? The knowledge put in by the expert must of course be domain-specific, but an induction program which assists his input of knowledge should typically show generality, as in the following example.

Lymphatic cancers

In the differential diagnosis of lymphatic cancers Ivan Bratko successfully used an Edinburgh

learning program derived from Ross Quinlan's ID3, which was in turn based on Earl Hunt's CLS (Concept Learning System) developed in the 1960's. The Edinburgh program was developed using the domain of chess. Strategies for chess end-games were conveyed by examples. The resulting correct and complete theory of King and Pawn against King is itself of interest to chess masters. Bratko then used the same program, in collaboration with a clinician, to construct a diagnostic scheme for the lymphatic cancers. In this experiment, with help only in extracting data from medical case histories, Bratko was able to perform as if he were a clinical expert, although knowing nothing about medicine (Bratko and Mulec, 1980).

Requirements for a refinery

What are the necessary ingredients for knowledge refining?

- (1) Knowledge-engineering software: *i.e.* "advice languages", "inference engines" (that part of an expert system able to make inferences from data supplied, and to retrace and display the lines of reasoning), and "induction modules" (that part of an expert system able to generate rules from examples).
- (2) A good development environment, *e.g.* UNIX or INTERLISP.
- (3) Trained knowledge engineers familiar with the above material.
- (4) One or more experts. Even with inductive knowledge generation automated to the limit of current technique, an expert is still needed to choose the repertoire of primitive measurements or attributes to evaluate as relevant to the given problem domain.

Originally designed to deliver interactive advice, it now appears that expert systems have a by-product which is even more important - the finished knowledge-base itself.

To recapitulate, expert systems can be used:

- to get knowledge into the machine
- to test it
- to debug it
- to fill gaps
- to extend it
- to modify it

and finally

- to put the knowledge back into human hands in improved form.

Thus, the possibility now exists of superseding (as the automobile has superseded the horse) a craft which has been going on for thousands of years - namely, the writing of manuals and texts on how to do things.

BIBLIOGRAPHY

- Bratko, I. (1978). Proving correctness of strategies in the ALI assertional language. *Information Processing Letters*, 7, 223-230.
- Bratko, I. and Mulec, P. (1980). An experiment in automatic learning of diagnostic rules. *Informatica*, 4, 18-25.
- Buchanan, B.G., Smith, D.H., White, W.C., Gritter, R., Feigenbaum, E.A., Lederberg, J. and Djerassi, C. (1976). Applications of Artificial Intelligence for chemical inference, XXII, Automatic rule-formation in mass spectrometry by means of the Meta-DENDRAL program. *J. Amer. Chem. Soc.*, 98, 6168-6178.
- Clarke, M.R.B. (1977). A quantitative study of King and Pawn against King. *Advances in Computer Chess 1* (ed. M.R.B. Clarke), pp. 108-18. Edinburgh: Edinburgh University Press.
- Hunt, E.B., Marin, J. and Stone, . (1966). *Experiments in Induction*. New York: Academic Press.
- Kopec, D. and Michie, D. (1983). *Mismatch between Machine Representations and Human Concepts: Dangers and Remedies*. Brussels: EEC Directorate-General for Science, Research and Development.
- Kopec, D. and Niblett, T. (1980). How hard is the play of the King-Rook-King-Knight ending? *Advances in Computer Chess 3* (ed. M.R.B. Clarke), pp. 57-81.
- McDermott, J. (1982). XSEL: A computer salesperson's assistant. *Machine Intelligence 10* (eds. J.E. Hayes, D. Michie and Y-H. Pao), pp. 325-337. Chichester: Horwood, and New York: Halsted Press.
- Michalski, R.S., Carbonell, J. and Mitchell, T. (eds.) (1983). *Machine Learning: an Artificial Intelligence Approach*. Palo Alto: Tioga.
- Michalski, R.S. and Chilausky, R.L. (1980). Knowledge acquisition by encoding expert rules *versus* computer induction from examples: a case study involving soybean pathology. *Int. J. for Man-Machine Studies*, 12, 63-87.
- Michie, D. (1977). A theory of advice. *Machine Intelligence 8* (eds. E.W. Elcock and D. Michie), pp. 151-158. Chichester: Horwood, and New York: Halsted Press.
- Michie, D. (1982a). Experiments on the mechanisation of game learning: 2. Rule-based learning and the human window. *Computer Journal*, 25, 105-113.
- Michie, D. (1982b). Measuring the knowledge-content of expert programs. *Bulletin of the Institute of Mathematics and its Applications*, 18, 216-220.
- Michie, D. and Shapiro, A. (1983). Research memorandum, in preparation.

- Moto-oka, T. (ed.) (1982). *Fifth Generation Computer Systems*. Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, October 1981). Amsterdam, New York and Oxford: North Holland.
- Niblett, T.B. and Paterson, A. (1982). *ACLS MANUAL*. Edinburgh, UK, and Champaign, USA: Intelligent Terminals Ltd.
- Pople, H.E., Myers, J.D. and Miller, R.A. (1977). DIALOG: a model of diagnostic logic for internal medicine. *Proc. 5th Inter. Joint Conf. on Artif. Int.* (IJCAI-77). Pittsburgh: Computer Science Department, Carnegie-Mellon University. (This program was subsequently re-named INTERNIST. It is now called CADUCEUS.)
- Quinlan, J.R. (1979). Discovering rules by induction from large collections of examples. *Expert Systems in the Microelectronic Age* (ed. D. Michie), pp.33-46. London, Paris and New York: Gordon and Breach. See also Quinlan's chapter in Michalski, Carbonell and Mitchell (1983).
- Quinlan, J.R. (1982). Semi-autonomous acquisition of pattern-based knowledge. In *Introductory Readings in Expert Systems* (ed. D. Michie), pp.192-207. New York, London and Paris: Gordon and Breach.
- Shapiro, A. (1983). The role of inductive learning in expert systems. *Ph.D. Thesis*. Edinburgh: Machine Intelligence Research Unit.
- Shapiro, A. and Niblett, R.B. (1982). Automatic induction of classification rules for a chess end-game. *Advances in Computer Chess 3* (ed. M.R.B. Clarke), pp.73-92. Oxford: Pergamon.
- Shortliffe, E.H. (1976). *Computer-Based Medical Consultations: MYCIN*. New York: Elsevier, North Holland.
- Wipke, W.T. (1977). Computer-assisted 3-dimensional synthetic analysis. *Computer Representation and Manipulation of Chemical Information* (eds. W.T. Wipke, S.R. Heller and E. Hyde), pp. 147-174. London and New York: Wiley Interscience.
- Yao, A. (1982). Theory and applications of trap-door functions. *Proc. 23rd Annual Symposium on Foundations of Computer Science* (Chicago), pp. 80-91. Los Angeles: IEEE.

Knowledge Acquisition and Learning in EXPERT

Casimir A. Kulikowski
Department of Computer Science
Rutgers University
New Brunswick, New Jersey 08903

ABSTRACT

Learning techniques are frequently de-emphasized in building expert systems because it is assumed that a human expert will provide the knowledge base. Yet, rarely are such knowledge bases complete or accurate, and much testing and refinement is needed before expert performance can be achieved. A data base of problem-solving examples with independently validated conclusions can be very useful to:

- a) induce reasoning rules using partial knowledge of causal and hierarchical relationships among data and hypotheses; and
- b) suggest improvements (generalizations or specializations) in the rules provided by the human expert. Two such systems have been developed to help construct and refine rules for the EXPERT consultation scheme. They have been successfully used to construct and validate knowledge bases in several medical domains.

SEEK: A System for Empirical Experimentation with Expert Knowledge

Expert knowledge for solving complex medical consultation problems changes frequently as new advances in clinical research appear, and as practical experience with the management of diseases is accumulated. Just as the human expert must keep up-to-date within a specialty, so must any of the expert systems that we develop. We have designed the SEEK system [1,2,3] to enable the builder of an expert consultation model to quickly and easily update and improve the performance of the reasoning model, based on results from medical cases with well-established diagnostic and therapeutic end-points.

SEEK prompts the model builder to enter the description of reasoning rules in the form of a table, where each conclusion can be defined in terms of classical medical terminology (major and minor findings, as well as necessary and exclusionary requirements). Each conclusion can have levels of confidence attached to it, such as definite, probable or possible, to account for the uncertainties and ambiguities in the reasoning. SEEK automatically converts the tabular description into

EXPERT rules, and runs the cases in the data base against the resulting model.

When disagreements between the expert's conclusions and those of the model are detected by SEEK, it suggests a set of experiments that can be carried out to modify the model's rules that have the potential of improving performance. These experiments usually consist of suggestions to generalize or specialize the logical scope of a rule or group of rules. Rather than carry out these experiments automatically, SEEK presents the model designer with a ranking of the experiments, and the designer must interact with the system to make the final choice. The experiments are generated by rule refinement heuristics which make use of the performance of the prototype reasoning model to instantiate specific advice on the modification of predicates or uncertainty levels in the rules. Experiments are typically tried out on an incremental basis, modifying one rule or a set of related rules at a time, and comparing the new with the previous performance. Because the overall performance of a model depends on the interaction of many rules, modification of a few of them will only change part of any faulty results, so various heuristics in SEEK have been developed to propose and evaluate the priority of the different specializations and generalizations based on expected improvements in performance.

Once an experiment has been selected and carried out, SEEK gives further advice on how to refine the rules. The designer will typically go through a sequence of experiments before settling on a final structure for the model. And even then, as new and more complex cases are incorporated within the data base, SEEK can be used repeatedly to see if they warrant any changes in the reasoning rules.

Because EXPERT models are compiled, we are able to run many experiments against quite substantial data bases (many hundreds of cases), keeping all in computer memory, which is needed to carry out the kind of global performance assessment that we are doing. In the past year, SEEK has been tested with a large knowledge base in the area of rheumatology developed by our

collaborators in rheumatology at the University of Missouri - Columbia (Drs. Gordon Sharp and Donald Lindberg). Heuristic rules that propose the experiments have shown their value in suggesting changes that resulted in significant diagnostic performance improvements for a large knowledge base (about 1000 observations, 400 hypotheses and 1000 rules). SEEK has also proved itself in helping build a large dermatology model.

Knowledge-based Induction of Rules

When the knowledge base designer lacks the time to construct a preliminary or prototype set of reasoning rules, it may be useful to try to directly induce them from the data bases of problem solving cases. Initially, it is typical for the human expert to specify the reasoning primitives (observations and hypotheses) and their interrelationships (mostly causal, taxonomic and instantiation forms). These relations can be used with a variety of learning heuristics to extract reasonable rules of inference from the data base of cases. We have taken this approach for models where the expert was able to provide us with such a structure: in glaucoma and rheumatology. Our learning program [4] uses a blackboard representation, with multiple knowledge sources to handle the different conclusions that the model is to arrive at, and the formation of the rules from the data that pertain to them.

Each knowledge source represents the a-priori knowledge of one physiological system or process and its mechanisms of dysfunction. It imposes a structure on the training data which implicitly defines permissible clusters of data and intermediate hypotheses. Competing interpretations can be proposed by having different knowledge sources view the data as manifestations of different physiological processes. A knowledge source has both declarative and procedural components. The declarative part is a static relational network of elements, each of which represents a particular state of dysfunction. They are: observations most closely attributable to a state; taxonomic links to subsystems; and causal rules linking one state to another. The procedural part consists of three functions: PROPOSE, EXPECT and GOAL. The PROPOSE function is a constrained generator of new candidate rules. It is invoked when the system chooses as its focus of attention an element which is part of the knowledge source network. It translates prior knowledge about all the permissible associations that include the focused element into pre-candidate inference rules, with the element as the consequent or right-hand side of the rules. It then proposes a

new candidate rule for each rule which is supported by pre-existing rules and which explains a portion of the present training cases. Pre-candidate rules with the same right-hand side element are generated in a general-to-specific order, so concise rules are found first whenever possible. The EXPECT function looks at new candidate rules and decides whether sufficient evidence exists for some of their elements to predict activity in other elements in the knowledge source. These are model-directed expectations flowing down taxonomic links or along causal links. The latter allow the system to re-focus attention to a different knowledge source, when the links connect one knowledge source to another. The GOAL function looks for ways of satisfying a causal rule so as to significantly strengthen one of the candidate rules supporting it. It is used as a mechanism for breaking deadlocks, when the system is unable to choose among competing graph structures of candidate rules.

The knowledge sources communicate through a global blackboard where the candidate rules are recorded. Each node on the blackboard is a rule with pointers to previously generated rules that are needed to support it. In addition, there are lists of positive and negative training instances which match the rule graph, and a numerical evaluation function.

The process by which the rules are learned can be viewed as a type of hypothesize-and-test procedure. The knowledge sources use the structure of prior causal and taxonomic knowledge to generate hypotheses about reasonable candidate rules, test these against the patterns of observations from the data base, and place those which are supported onto the blackboard. Once this has been done by all the knowledge sources, a support set of rules is produced to subsequently confirm or deny expectations of causal connectivity generated by the knowledge sources. Focus of attention heuristics activate the knowledge sources that contain causal consequences of an expectation, and these are tested against patterns in the data base and the blackboard. The result is an extended set of candidate rules, which is then pruned by various heuristics to produce a final covering set of rules for the training cases.

This system has been tested with subsets of the CASNET [5] glaucoma knowledge base and the rheumatology knowledge base [3]. We were able to generate rules that were comparable to those produced independently by the human experts, and in several instances produced more efficient rules for covering the cases in our data base.

In relation to a system such as SEEK, this approach represents a preprocessing or

alternative approach to developing a prototype model. We are now investigating the role of additional medical constraints on the strategies of rule generation.

Acknowledgements

The author wishes to thank Sholom Weiss, Peter Politakis and George Drastal for their collaboration and research within the Rutgers Research Resource on Computers in Biomedicine. This work was supported in part by grant RR-643 from the National Institutes of Health, Biotechnology Resources Program, Division of Research Resources.

References

- [1] Weiss, S. and Kulikowski, C. , Data Base Aids for Designing Consultation Systems, Proceedings of the Third World Conference on Medical Informatics (MEDINFO '80), pp. 794-797. 1980.
- [2] Politakis, P. and Weiss, S.M. , A System for Empirical Experimentation with Expert Knowledge, Proc. Fifteenth Hawaii International Conference on Systems Sciences, pp. 649-657. 1982
- [3] Politakis, P. and Weiss, S.M., Using Empirical Analysis to Refine Expert System Knowledge Bases, Journal of Artificial Intelligence (in press, 1983).
- [4] Drastal, G. and Kulikowski, C.A., Knowledge Based Acquisition of Rules for Medical Diagnosis, Journal of Medical Systems, 1982.
- [5] Weiss, S.M., Kulikowski, C.A., Amarel, S., and Safir, A., A Model-Based Method for Computer-Aided Medical Decision-Making, Artificial Intelligence, v.11, pp. 145-172, 1978.

HIERARCHICAL MEMORIES: AN AID TO CONCEPT LEARNING

Claude Sammut and Ranan Banerji

St. Joseph's University, Philadelphia, PA 19131

ABSTRACT

Marvin is a concept learning program which learns concepts by focussing on an example object for the concept and forms hypotheses, which it tests by showing objects to the trainer and asking him questions about their belonging to the concept being learned. Descriptions of concept are stored as sets of Horn clauses. The description of one concept may use the names of other concepts, so the interdependencies of concepts can be learned. A concept may consist of sets of objects, relationships between objects, functions which transform objects, or any other predicate.

Key Terms: Learning, machine learning, induction, artificial intelligence, predicate logic, automatic programming.

1. INTRODUCTION

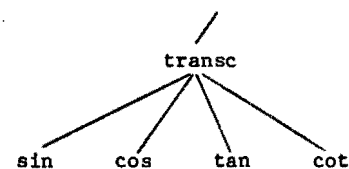
When we characterize learning as a search process, we assume the following:

1. There is a language which is used to describe concepts.
2. A state in the search space is a collection of descriptions written in this language. Given a set of examples, a goal state is a set of descriptions such that each positive example satisfies some description and no negative example satisfies any description.
3. There is a set of generalization rules for transforming states into new states.
4. A strategy for using the set of examples as a guide to choose a useful sequence of transformations which leads one to the goal state.

The problem of learning is to search through the space with a sequence of generalizations until a path to the goal is found. Learning systems such as LEX [Mitchell, 1982], INDUCE [Michalski, 1982] and MIS [Shapiro, 1981] have a fixed set of generalization rules. In programs such as CONFUCIUS [Cohen, 1978] and Marvin [Sammut, 1981], we have investigated learning systems in which the generalization rules change depending on the state, i.e. on the set of descriptions learned. Put another way, while in previous work the transformations operated on single sentences in the state, in our work the rules take the whole state into consideration. This makes for a

significant difference in the efficiency of the descriptions learned and, under many circumstances on the efficiency of the search process itself.

To contrast these two approaches, let us first look at a fixed language system such as LEX. This program is intended to learn how to perform symbolic integration. Expressions in LEX's language represent algebraic expressions. The system's generalization rules describe the way in which one expression may be transformed into a more general one. We reproduce below, part of the grammar for LEX's description language.



Note that the tree structure provides an ordering among expressions in the description language, indicating what kinds of expressions are more general than others. Thus, the designers of LEX have supplied it with a lot of background knowledge to enable it to learn to integrate. The tree not only provides the program with the simple concepts it needs (viz. real number, transcendental function, etc), it also shows the relationship between concepts. This relationship fixes the generalization rules once and for all. [Utgooff, 1983] explains how this "bias" increases the efficiency (indeed, the possibility) of learning. In our opinion this explanation is completely valid, unless the initial bias is unsuited for the descriptions one needs to learn.

Utgooff and Mitchell have pointed out this need for updating the bias and have shown how certain terms can be incorporated into the language and the partial order. However, there are further needs of language enrichment which can not be incorporated into the structure that they suggested. There is often a need for enriching the language by adding new disjunctions. The language is set up for generalizing the term $\sin(x)$ to $\text{transc}(x)$. It can be extended to include generalizations like $\sin(x) \text{ OR } \cos(x)$ without much trouble, say by calling the generalized term "subtrig(x)". However, terms like $\sin(x)\cos(x)$ can only be generalized to $\text{transc}(x)\cos(x)$, $\sin(x)\text{transc}(x)$ or $\text{transc}(x)\text{transc}(x)$. The tree structure prevents generalizations like $\sin(x)\cos(x) \text{ OR } \tan(x)\cot(x)$. The structure of the language has to be changed if one needs terms

like that.

Contrast with this the language used by Marvin, which allows the easy extension of the language by using sets of Horn clauses as descriptions, just as MIS does. The program is designed to emulate teacher/student learning. Unlike LEX, it is not given a hierarchy of concepts initially. The relationships between concepts are built up as the trainer presents new concepts. As in human teacher/student interactions, it is expected that simple concepts will be taught before complex ones. For example, addition is taught as a concept before multiplication is taught since the description of multiplication may use addition. The concepts which Marvin learns and the relationships between them form a model of the world in which the program exists.

Learning is not always aided by a benevolent teacher. Circumstances may force situations where the learning system tries to learn a more complex concept before learning a simpler concept upon which the complex one depends. Scientific investigation is an example of learning where the scientist (student) must deal with an indifferent trainer (nature). The history of science consists almost entirely of an account of how we have built models of the world, only to have to rebuild them when new concepts were later admitted, often with great reluctance, to be true. Shapiro's program, MIS, was an attempt to develop a system which could propose theories and, when necessary, trace through the derivation of a theory to discover an error if the theory was proved to be wrong.

We wish to investigate this kind of learning further. However, there are two difficulties with Shapiro's approach. Firstly, MIS uses a language determined entirely by the trainer. As we have already said, this limits the flexibility of the learning system. Secondly, MIS creates generalizations without taking advantage of the concept hierarchy to improve efficiency. Since Marvin was designed to avoid these difficulties, the new system is being built on the foundations already laid in this work.

However, before we discuss the problems that occur when we extend Marvin in this way, it may be worthwhile to discuss Marvin itself.

2. MARVIN

Marvin learns concepts by playing a game of "20 questions" with the trainer. The trainer shows the program an object which is an example of the concept which is to be learned (the target concept). Marvin uses this example to construct a hypothesis or trial concept by generalizing the description of the object so that it specifies a class of objects. A generalization is said to be consistent if the concept it describes is less general than the target.

To find out if a generalization is inconsistent, Marvin constructs object which are contained in the trial and shows them to the trainer. If the trainer tells Marvin that one of the objects is

not contained in the target then the generalization is inconsistent. When this occurs, Marvin tries to create a new trial which is more specific than the previous one. This is a restriction instead of a generalization. The program has been designed so as to minimize the number of objects which must be presented to the trainer in order to determine that a generalization is inconsistent.

When the program has learned a concept, it stores the description in its memory. A generalization is made when Marvin recognizes an object as belonging to a concept it has already learned. Marvin continues to make as many generalizations as it can. When it has exhausted all possibilities, the learning process stops and the final trial is stored in memory.

A concept is represented by a set of Horn clauses of the form:

$P \leftarrow Q \ \& \ R \ \& \ S.$

P, Q, R and S are predicates and the standard interpretation is applied: P is true if Q and R and S are true. The clause will be called a definition of the concept P. An alternative interpretation for a clause is: Q & R & S represents an instance of P, so Q & R & S may be generalized to P. Thus, a concept is also a generalization rule. For example, we may define a playing card as something having a suit and a face-value (called "value" below).

```
playing_card(X) <-
    suit(X, S) &
    any_suit(S) &
    value(X, V) &
    any_value(V).
```

"any_suit" may be defined as:

```
any_suit(X) <- atom(X, hearts).
any_suit(X) <- atom(X, diamonds).
any_suit(X) <- atom(X, clubs).
any_suit(X) <- atom(X, spades).
```

"atom(X, hearts)" means that "hearts" is an atomic value. "any_value" and other concepts such as "same_value" may be defined in the same way:

```
any_value(X) <- atom(X, king).
any_value(X) <- atom(X, queen).
. . . . .
any_value(X) <- atom(X, ace).
```

```
same_value(X, Y) <-
    suit(X, S) &
    suit(Y, S) &
    any_suit(S).
```

Marvin's memory is a database of such clauses. Predicates are also used to describe objects, for example,

```
suit(a, s) &
atom(s, hearts) &
value(a, v) &
atom(v, queen)
```

describes an object "a" which is a queen of hearts.

The predicates representing an object are called primary predicates. When an object is described to Marvin, the program searches its memory to determine if there is a subset of the primary predicates which matches the right hand side of a clause stored in memory.

In this example, "atom(s, hearts)" matches the right hand side of one clause in "any_suit" so we can infer that any_suit(s) is true. Similarly, any_value(v) can also be inferred. Since

any_value(v) & any_suit(s)

is true, then playing_card(a) is true.

Definition 1: Given a set of clauses:

$S^1 \leftarrow M^1$

$S^k \leftarrow M^k$

and a conjunction of predicates, C, we say that C implies C' if there is an M^i and a substitution, σ , such that

$\sigma M \leq C$ and $C' = C + \sigma\{S^i\}$

We treat atomic predicates in a conjunction as members of a set. "<=", "*", "+" and "-" mean subset, intersection, union and difference.

Definition 2: Given a conjunction, C^0 , we define a sequence $C^0..C^n$ such that C^{i+1} is implied by C^i and C^n does not imply anything. We write $A(C^0) = C^n$, representing the set of all predicates implied by C^0 .

If, in the original description of the object we replace "atom(s, hearts)" by the concept it implies i.e. "any_suit(s)", then we obtain the more general description:

suit(a, s) &
any_suit(s) &
value(a, v) &
atom(v, queen)

The description can be generalized further by replacing "atom(v, queen)" by any_value(v)".

The replacement operation can be described more formally as follow. We may think of any conjunction of predicates as a trial concept. In particular, the object shown as an example by the trainer is our first trial concept.

Definition 3: If T^i is a trial concept and M is a subset of T^i , and there exists a clause in memory $S \leftarrow M'$ such that with the substitution, σ , $M = \sigma M'$ then we define a replacement operation which will create a new trial, T^{i+1} , such that

$T^{i+1} = T^i - M + \sigma S$

T^{i+1} is a generalization of T^i . If there is more than one clause in the definition of the concept S the T^{i+1} is more general than T^i , since S describes a larger set than is described by M.

Suppose we are trying to teach Marvin the concept "pair":

pair(a, b) <-
playing_card(X) &
playing_card(Y) &
same_value(X, Y).

by giving it the example, T^0 :

pair(a, b) <-
value(a, v) &
atom(v, ace) &
suit(a, sa) &
atom(sa, spades) &
value(b, v) &
suit(b, sb) &
atom(sb, hearts).

Note that the values of a and b refer to the same object. The program may begin generalizing this description and eventually produce the following trial, T^1 :

pair(a, b) <-
value(a, v) & (1)
suit(a, sa) & (2)
value(b, v) &
suit(b, sb) &
any_value(v) & (3)
any_suit(sa) & (4)
any_suit(sb).

which is a consistent generalization.

Definition 4: If $T^0..T^k$ is a sequence of trials such that T^i generalizes to T^{i+1} then we say T^0 satisfies T^k .

Definition 5: Trial T is consistent with the target T' if any object which satisfies T also satisfies T' .

Since that values of cards "a" and "b" refer to the same object, equality of values has been preserved. Whenever two objects with the same description are entered, they are given the same name so that equality may be considered as part of the concept. By generalizing further, Marvin tries to remove this equality. For example, a new trial, T^{i+1} , produced from T^i may be:

pair(a, b) <-
card(a) &
value(b, v) &
suit(b, sb) &
any_value(v) &
any_suit(sb).

The numbered statements in T^1 form the set, M,. These imply "card(a)", i.e. S in the definition above. Only (3) has not been replaced because it is still part of the description of "b". T^{i+1} is too general since the value of card "a" is no longer related to value of card "b". When an inconsistent generalization is produced, we must find a way of creating another, more specific trial.

A generalization results in a set of predicates in a trial being replaced by a single, more general, predicate. The replacement operation throws away some specific information contained in the set, M , in favour of the more general statement, S . If the new trial is consistent then the information lost was not important. However, if the generalization is inconsistent then too much information was lost. To make T_{i+1} more specific, Marvin re-examines the predicates in M to determine which ones contain essential information. In the example, (1), "value(a, v)" is necessary to maintain the condition that the two cards have the same value. Once this has been determined, the program tries to remember other concepts which contain a statement like (1).

Definition 6:

Let $T_1 = T - M_1 + S_1$
and $T_2 = T - M_2 + S_2$
where $M_1 * M_2 \neq \emptyset$

then $T_1 + T_2$ is a restriction of T_1 .

same_value(X, Y) is S_2 in definition 6, and may be added to the trial in an attempt to make a consistent trial T_{i+2} .

To test a trial T_{i+1} , Marvin shows the trainer an instance of T_{i+1} . If the program can create an object which does not belong to the target then T_{i+1} is inconsistent. The object will satisfy all the predicates in T_{i+1} but not satisfy any predicates in $A(T_i)$ except those implied by T_{i+1} . Thus, "same_value(a, b)" must be false so that the equality of values can be tested.

Definition 7: Let T_{i+1} be a generalization of T_i . Any object which satisfies T_{i+1} but negates each element of $A(T_i) - A(T_{i+1})$ is called a crucial object. If T_{i+1} is inconsistent and T' is the target then no crucial object of T_{i+1} satisfies the target.

Let us now summarize Marvin's learning algorithm:

To learn a new clause:

The trainer gives the program an example of the concept.

- Find the list of replacement operations obtainable from the primary predicates.
- Use these operations to generalize the initial trial.
- Store the resulting concept in memory.

To find the list of replacement operations:

Let P initially be the set of all primary predicates.

- If P is a conjunction, find the set, I , of all concepts implied by P .
- Append I to P ,
- Repeat this procedure until no more implied concept can be found. The result is $A(P)$.

The list of replacement operations consists of all the clauses used to find the implied concepts in $A(P)$. (see defn 1).

To generalize a trial T_i :

- Choose the first concept in the list of replacement operations.

- Perform the replacement to obtain T_{i+1} . (see defn 3).
- Try to "qualify" T_{i+1} .
- T_{i+1} cannot be qualified, abandon this generalization.
- Repeat this procedure with the next operation in the list.

To qualify a trial T_{i+1} :

- Construct a crucial object which satisfies T_{i+1} but does not satisfy $A(T_i) - A(T_{i+1})$ (defn 7).
- Ask trainer if object satisfies target. If it does then T_{i+1} has been qualified.
- If not then restrict T_{i+1} to new trial T_{i+2} .
- Try to qualify T_{i+2} .

To restrict a trial, T_{i+1} :

- Search list of replacement operations for an $S_2 \leftarrow M_2$ such that $M_1 * M_2 \neq \emptyset$. M_1 is the replacement for T_i .
- Perform replacement on T_{i+1} to obtain the new trial T_{i+2} . (see defn 6).

To construct an example from a trial, T_{i+1} :

- To construct an example from P & Q :
Construct P , Construct Q .
- To construct an example from an atomic predicate P and there is a clause $P \leftarrow B$ in memory, and B is not in $A(T_i) - A(T_{i+1})$, if B is already true, do nothing, otherwise construct example using B .
- To construct an example from an atomic predicate P and there is no clause $P \leftarrow B$, add B to conjunction representing example.

3. CONCEPTS THAT MARVIN HAS LEARNED

In this section we describe some of the concepts which Marvin has been able to learn.

Since concepts are represented as Horn clauses in first order logic, they may be regarded as logic programs. Thus Marvin is able to perform as an automatic programming system as well as a learning system.

Marvin has learned arithmetic concepts, including representing numbers as strings of bits, comparing those numbers and adding them. Also in the domain of program synthesis, Marvin has learned what lists are, how to append lists, reverse them, and sort lists of numbers using an insertion sort. Sorting required that the knowledge of numbers and lists be combined so that the elements of the list could be compared.

All of the program concepts above are disjunctive and recursive. To illustrate how such concepts are learned, consider the definition of a list of alphabetic characters. A list is either nil or an object which has a head (a letter) and a tail (a list).

The learning algorithm in §2 showed how a single clause could be learned by presenting an instance of the concept. If the concept is disjunctive then a new example is shown for each disjunct to be learned. Learning the alphabet is merely rote learning. The clauses shown below are entered by

the trainer and stored without generalization.

```
letter(X) <-value(X, a).
. . . . .
letter(X) <- value(X, z).
```

To teach Marvin about lists, the trainer shows it nil first:

```
list(n) <- value(n, nil).
```

The object "n" is a list because its value is nil. The term "value(n, nil)" is an atomic predicate which has no interpretation. This will be stored in memory as:

```
list(X) <- value(X, nil).
```

The object name "n" has been replaced by the variable "X". A further example of a list is:

```
list(w) <-
  head(w, h) &
  value(h, a) &
  tail(w, t) &
  value(t, nil).
```

Matching predicates in this example with the right hand sides of clauses stored in memory, the program will generalize "value(h, a)" to "letter(h)" and "value(t, nil)" to "list(t)".

In a different domain, Marvin is capable of learning simple grammars, such as:

```
S → NP VP
NP → Pnoun
NP → DET NOUN
Pnoun → John
Pnoun → Mary
DET → a
DET → the
NOUN → apple
NOUN → giraffe
VP → IVERB
VP → TVERB NP
IVERB → sleeps
IVERB → dreams
TVERB → dreams
TVERB → eats
```

Although we have no space to show the learning session for this grammar, it is an interesting task for Marvin since some concepts such as VP must be partially learned (one disjunct), then NP must be learned and then VP completed. See [Sammut, 1981] for a complete description.

A more difficult language recognition problem which Marvin has learned is that posed by [Hayes-Roth, 1978]. The task was to learn the rules to transform a sentence in the active form, such as, "The little man sang a lovely song" to the passive form, "A lovely song was sung by the little man".

No work in concept learning would be complete without some reference to Winston's ARCH [1970]. Marvin learned to describe physical objects with the properties: orientation, shape, being on top

of another object, touching another object. The program was then shown a single example of an arch. It generalized the description of the arch and tested the generalizations by showing the trainer its own descriptions of would-be arches until it produced a correct description.

4. BUILDING HIERARCHICAL MEMORIES

At present, the trainer is responsible for teaching concepts in an order which will allow the program to build a hierarchical memory. If the teaching sequence is not correct then Marvin may never be able to learn some concepts. For example, if we want to teach Marvin what a royal flush is then it must know which cards are court cards. What happens if these have not been taught before an example of a royal flush is shown?

In the present system, the program only shows the trainer one example to test a generalization. If the training sequence is carefully ordered, this is sufficient. If not, Marvin may produce an incorrect description. The concept "any_value" which we introduced earlier made no distinction between court cards and other cards. When constructing an example to show the trainer, Marvin could, by chance, choose court cards to test the generalization to "any_value". This would appear correct to the trainer, even though the concept should only refer to court cards. Thus, an inconsistent generalization could be passed as correct.

This problem could be avoided if Marvin showed more examples. This works for cards because there is only a finite number of objects in the domain. If the program is learning about number or lists, how many examples should be shown? The design of a good experiment to test a hypothesis only comes with some experience in the domain being investigated. Therefore, we should expect the program to make mistakes and develop some strategy for recovering when the mistake is discovered.

The choice of learning strategy will largely depend on the confidence the learning program has in its model of the world and the training data. That is, if contradictory results occur frequently then the model or data must not be correct. What to do when a contradiction appears may be controlled by heuristics which describe the confidence in a particular concept. For example, if Marvin is playing poker with its trainer (a somewhat far-fetched possibility) and the trainer realizes that the program's definition of a royal flush is incorrect, Marvin must be able to resolve this contradiction to its world model.

A possible course of action is as follows:

- initially, the example may be rejected as due to noisy data. This may be worthwhile doing if the concept in question has been used successfully elsewhere.
- if contradictions recur, they may be kept as special cases (exception conditions).
- If the recur frequently enough, then the model must be changed.

Let us simplify the learning task to "royal pair" rather than royal flush. Given the concepts we introduced earlier, Marvin could learn:

```
royal_pair(X, Y) <-
  playing_card(X) &
  playing_card(Y) &
  same_value(X, Y).
```

Clearly, this is an overgeneralization and will be discovered when Marvin's poker opponent points out that what it thought was a royal pair was not, i.e. it discovers a negative example such as:

```
not(royal_pair(X, Y)) <-
  suit(X, SX) &
  atom(SX, hearts) &
  value(X, V) &
  atom(V, 10) &
  suit(Y, SY) &
  atom(SY, diamonds).
```

Our problem now is to identify which generalization went wrong. Shapiro uses a method called backtracing to work back through the derivation of a hypothesis. At each point where a generalization was made, his program, MIS, asks an oracle (the trainer) if the generalization was valid. To do this with Marvin, we need to know the replacement operations performed to determine that X and Y were supposed to be a royal pair.

By questioning the trainer, asking which replacement caused the inconsistent generalization, the program will find that "any_value(V)" should not occur in the description. Hence all inferences following from this premise must be false, or at least must be qualified by additional restrictions.

There are several alternatives to the way in which we may proceed from here. If a concept like "any_value" has been used frequently in the descriptions of concepts that have proved useful then we may wish to leave the definition of "any_value" unchanged and just add an exception to the definition of royal pair. If the negative example used the value, 10, then we may add the qualification that the cards must have the same value as long as it is not 10. E.g.

```
royal_pair(X, Y) <-
  playing_card(X) &
  playing_card(Y) &
  value(X, V) &
  value(Y, V) &
  not(atom(V, 10)).
```

Assuming less confidence in "any_value", we may decide that any reference to it in "royal_pair" must be removed. Therefore generalizations made with "any_value" must be undone.

Suppose that Marvin has seen two positive examples of a royal pair: pair of kings and a pair of queens. This was generalized to a pair of any value. If the generalization "any_value" is then the program may return to a concept with two disjuncts: "a pair of kings or a pair of queens". A further example involving a jack would add a

third disjunct.

Now we run into another problem. Will we allow an arbitrary number of disjuncts to be added whenever contradictions in the model occur? With some introspection, Marvin may see that disjuncts involving kings, queens and jacks are created often. Therefore, it might be worthwhile to create a new concept for them, thus eliminating some disjuncts.

For example, if there are two clauses:

```
W <- Q & R & S.
W <- Q & R & P.
```

then they may be rewritten as:

```
W <- Q & R & SP.
SP <- S.
SP <- P.
```

Throughout this discussion we have used very vague terms such as "confidence" and "it may be worthwhile". This is because we are considering learning in an environment which does not enable the program when it has found the "right" description. An example of this in real life is that a scientific theory cannot be proven correct, only incorrect. The best we can do is to accumulate evidence to support the theory. The way we find the evidence is not always systematic but is often based on experience. Thus, we propose that a system which learns by discovery must also learn "confidence" in its model.

5. CONCLUSION

The examples we have used in explaining the method of operation of Marvin may have been deceptive in their simplicity. A more complex example would have been more convincing of the powers of the program, but would probably have obscured its structure. It is important to note that the simple learning techniques devised for Marvin can lead to quite complex concepts.

The major motivation behind this work is the realization that, no matter how sophisticated a language be devised initially to describe objects, there is no assurance that the descriptions of object classes will remain conjunctive. The major difference between Marvin and many previously designed programs lies in the fact that it can handle disjunctions with almost the same ease as it can handle conjunctions. Both its strengths and weaknesses stem from this fact.

Let us now take stock of the advantages one loses once one gives up the insistence on conjunctive descriptions. The major advantage lay in the fact that once a positive example is focussed on at the beginning of the learning process, it is known that the final description will be the conjunction of a subset of all the predicates which the object satisfies. The only problem that remains is to weed out the unnecessary predicates. An easy way of doing this, especially if these predicates are independent, is to create test objects which

differ from the focus object in differing only by a single statement. If the test object lies in the concept then the corresponding statement is irrelevant.

Things get somewhat complicated if the statements are not independent. But as long as the description remains conjunctive, the problems are not insurmountable, as our above discussion of Marvin has shown. As a matter of fact, we have also shown that as long as the requirements of memory hierarchy can be met (i.e. the disjuncts of a concept are disjoint and the simpler concepts are placed in memory before the complex ones), even disjunctions can be handled.

That, then, is the main contribution of Marvin: to show that the advantages of conjunctive descriptions need not be given up entirely if one can retain a hierarchical memory structure. Of course, to retain these advantages, one has to give up the obsolete graph structures and nets of old and take advantage of the flexibilities of Horn Clauses. What causes the real problem is the question, "How does one retain a hierarchical memory structure while learning concepts?" Marvin has side-stepped this question by assuming a benevolent teacher who gives examples of simpler concepts before showing more complex concepts. We have described our initial thoughts about how one can try to keep memory hierarchical while trying to learn concepts without the help of a benevolent teacher.

6. REFERENCES

- Cohen, B.L., A Theory of Structural Concept Formation and Pattern Recognition, Ph.D. Thesis, Dept. of Comp. Sci., University of New South Wales, 1978
- Hayes-Roth, F. and McDermott, J., "Knowledge Acquisition from Structural Descriptions", Communications of the ACM, Vol. 21, pp 401-411, May 1978.
- Michalski, R.S., "A Theory and Methodology of Inductive Learning", Machine Learning, Michalski, Carbonell, Mitchell (Eds), Tioga, Palo Alto, 1983.
- Mitchell, T.M., Utgoff, P.E., Banerji, R.B., "Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics", Machine Learning, Michalski, Carbonell, Mitchell (Eds), Tioga, Palo Alto, 1983.
- Sammur, C.A., Learning Concepts by Performing Experiments, Ph.D. Thesis, Dept. of Comp. Sci., University of New South Wales, 1981.
- Shapiro, E.Y., "Inductive Inference of Theories From Facts", Research Report 192, Yale University, 1981.
- Utgoff, P.E. and Mitchell, T.M., "Acquisition of Appropriate Bias for Inductive Concept Learning", Proc. Second National Conference on Artificial Intelligence, Pittsburgh, 1982.
- Winston, P.H., Learning Structural Descriptions From Examples, Ph.D. Thesis, MIT-AI Lab, 1970.

LEARNING AS A NON-DETERMINISTIC BUT EXACT LOGICAL PROCESS

Y. Kodratoff, J.-G. Ganascia
L.R.I., Bât. 490
Université de Paris-Sud
F-91405, Orsay CEDEX-FRANCE

ABSTRACT

In this paper we intend to show that learning is closely related to problems of generalization of expressions and changes in description languages. We claim that one must use properties of description, without forgetting anything, during the generalization process. So the definition of generalization we propose here is the one used in classical logic. On the other hand the forgetting action, which is central in learning, is but a change of description. In this sense a good description keeps only the meaningful characteristics for the learning task. Our aim is to demonstrate that regardless of the point of which the change of description takes place (either during or after the generalization process), any change which leads to forgetting anything, or to completing the description has to be preceded by an analysis of the generalization success or failure.

Key terms : Learning, machine learning, pattern matching, generalization, change of description, structural matching, canonical form.

I. INTRODUCTION

The problem of learning is felt as a central one in A.I. since "a computer that doesn't learn really cannot make much of a claim on being intelligent" [Shank 82]. In spite of this strong social pressure pushing us to study learning, we realize very well that the field improves at an amazingly low speed.

A description of the main ideas relevant to the field has been given by Dietterich and Michalski [Dietterich & Al. 81] in a particularly understandable paper. We shall here consider that the reader is aware of [Dietterich & Al. 81] since our own paper criticizes and refines the generalization concept used by these authors.

The ideas more or less implicitly found in [Hayes-Roth & Al. 78] and [Vere 81] are presented in a more systematic way and their consequences are analysed. Our main point is that generalization implies that the descriptions to be generalized must be given in quite a standard way. If this is not the case then some preliminary "massaging" is necessary as shown in section 2.2. . The reader is reminded that Dijkstra used to say that some specification massaging is necessary before program

construction. In the following we hope to specify what "massaging" is.

II. KNOWLEDGE REPRESENTATION

The simple point we want to emphasize here is that generalization can be used as one of the tools necessary for building an efficient representation of knowledge.

If one admits that a good generalization algorithm must draw relevant information from a set of examples, then a failure to obtain this information must bring us to the idea that our representation of knowledge is not well adapted to the examples. If we have little knowledge about the world in which the learning is taking place, then we can only imagine some random changes in the representation. On the contrary, in a well-known and structured world, representation changes can be made in a systematic way.

Besides imagination or other meta-rules, generalization is an essential tool in the invention of the correct description language because it gives some properties the language must display so that one can learn from it.

The effect of fitting a description into a theory is also a very efficient way to find new descriptions (the meta-rules are given by the theory itself).

We shall now illustrate the above statements with two examples.

2.1. A "Bongard" example

Let us try to learn the difference between the following sets $S_1 = ((\nabla, \Delta, *), (\nabla, \Delta), (*), (\nabla, \square))$ and $S_2 = ((\nabla, \Delta, *, \square), (\nabla, *), (\Delta, \dagger), (\Delta, *))$, where S_1 is a set of examples and S_2 a set of counter examples.

The problem is : find a description of S_1 which excludes S_2 . The generalization algorithm we shall use has been given by Vere [Vere 80]. Let us suppose that we start from a description where the only information is the presence or absence of each geometrical pattern. For instance, $(\nabla, \Delta, *)$ can be described by (1,1,1,0,0) because we implicitly associate the first item of the quintuplet to ∇ , the second one to Δ , the third one to $*$, the fourth one to $\square$ and the fifth one to $\dagger$. The representation of S_1 and S_2 becomes : $S_1 = (A_1 = (1,1,1,0,0), A_2 = (1,1,0,0,0), A_3 = (0,0,1,0,0),$

$A4 = (1, 0, 0, 1, 0, 0)$,
 $S_2 = (B1 = (1, 1, 1, 1, 0, 0), B2 = (1, 0, 1, 0, 0, 0), B3 = (0, 1, 0, 0, 0, 1),$
 $B4 = (0, 1, 1, 0, 0, 0))$

Let x_i denote the name of a variable in the i th place of the quintuplets (otherwise stated the A_i 's and the B_i 's have the form $(x_1, x_2, x_3, x_4, x_5)$).

The least generalization of $A1, A2, A3, A4$ is such that the first four places are variables but such that the fifth place is always equal to 0. We shall write this as: $G_1 = (x_5 = 0)$. In Vere's generalization algorithm, the first approximation to the solution AP_1 is $AP_1 = G_1 = (x_5 = 0)$.

Following Vere, we delete the B_i 's which are not instances of G_1 , here $B3$ only and consider the least generalization of $S'_2 = (B1, B2, B4)$. It is $G_2 = (x_3 = 1) \& (x_5 = 0)$.

Following Vere, one says that the second approximation to the solution is

$$AP_2 = G_1 \& \neg(G_2 - G_1) \\ = (x_5 = 0) \& \neg(x_3 = 1).$$

We now delete from S_1 the A_i 's that are instances of AP_2 , i.e. those that are not instances of $G_2 - G_1$, namely $A2$ and $A4$. The least generalization of $S'_1 = (A1, A3)$ is

$$G_3 = (x_3 = 1) \& (x_4 = 0) \& (x_5 = 0) \text{ which gives the} \\ \text{third approximation} \\ AP_3 = G_1 \& \neg((G_2 - G_1) \& \neg(G_3 - G_2)) = \\ (x_5 = 0) \& \neg((x_3 = 1) \& \neg(x_4 = 0)).$$

One deletes from S'_2 the B_i 's that are not instances of $G_3 - G_2$, namely $B1$. The least generalization of $S'_2 = (B2, B4)$ is still G_3 . The process is therefore stopped and the reader can check that AP_3 is not a good description of $S_1 \& S_2$ since it recognizes $B2$ and $B4$ which should be rejected.

One standard reaction to this failure is to take into account the presence or absence of a pair of patterns. The description of the examples is no longer a 5-uplet but a 15-uplet $(x_1, \dots, x_{15})$ where we associate: x_1 and ∇ , x_2 and Δ , x_3 and $*$, x_4 and $\square$, x_5 and $\dagger$, x_6 and $\nabla\Delta$, x_7 and $\nabla*$, x_8 and $\nabla\square$, x_9 and $\nabla\dagger$, x_{10} and $\Delta*$, x_{11} and $\Delta\square$, x_{12} and $\Delta\dagger$, x_{13} and $* \square$, x_{14} and $* \dagger$, x_{15} and $\square \dagger$.

The examples and counterexamples are now described by:

$$S_1 = (A1 = (1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), \\ A2 = (1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), \\ A3 = (0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), \\ A4 = (1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)) \text{ and}$$

$$S_2 = (B1 = (1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0), \\ B2 = (1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0), \\ B3 = (0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0), \\ B4 = (0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0))$$

We obtain $AP_1 = G_1 = ((x_5 = 0) \& (x_9 = 0) \& (x_{11} = 0) \& (x_{12} = 0) \& (x_{13} = 0) \& (x_{14} = 0) \& (x_{15} = 0))$.

One deletes from S_2 the B_i 's that are not instances of G_1 : $B3$ and $B1$ and one obtains $S'_2 = (B2, B4)$.

The least generalization of $B2$ and $B4$ is $G_2 = ((x_3 = 1) \& (x_4 = 0) \& (x_5 = 0) \& (x_6 = 0) \& (x_8 = 0) \& (x_9 = 0) \& (x_{11} = 0) \& (x_{12} = 0) \& (x_{13} = 0) \& (x_{14} = 0) \& (x_{15} = 0))$.

It follows that $AP_2 = G_1 \& \neg(G_2 - G_1)$ with $G_2 - G_1 = (x_3 = 1) \& (x_4 = 0) \& (x_6 = 0) \& (x_8 = 0)$.

One deletes from S_1 the A_i 's that are not instances of $G_2 - G_1$, namely $A2, A4, A1$. One is left with $S'_1 = (A3)$ therefore $G_3 = A3$ and $AP_3 = G_1 \& \neg((G_2 - G_1) \& \neg(G_3 - G_2))$ with $G_3 - G_2 = (x_1 = 0) \& (x_2 = 0) \& (x_7 = 0) \& (x_{10} = 0)$.

It is easy to see that there is no instance of $G_3 - G_2$ in S_2 , so the algorithm is finished and the result is AP_3 which can be rewritten as:

$$(x_5 = 0) \& (x_9 = 0) \& (x_{11} = 0) \& (x_{12} = 0) \& \\ (x_{13} = 0) \& (x_{14} = 0) \& (x_{15} = 0) \& ((x_3 = 0) \vee \\ (x_4 = 1) \vee (x_6 = 1) \vee (x_8 = 1) \vee ((x_1 = 0) \& \\ (x_2 = 0) \& (x_7 = 0) \& (x_{10} = 0)))$$

The reader can check that AP_3 recognizes S_1 and rejects S_2 as it should be. Nevertheless, it is not the best generalization (it is easy to check its redundancy) and this indicates that the representation could still be improved if one admits that $S_1 \& \neg S_2$ is complete.

2.2. Reaction to uncomplete descriptions

It is usual to consider that the less complete description is the more general. As we shall see, we try to avoid this situation by a description completing whenever possible. For instance, let us consider the two descriptions $D1 = (\text{BLOCK } A) \& (\text{RED } A)$ and $D2 = (\text{BLOCK } B)$. The classical way to generalize $D1$ and $D2$ is $DG = (\text{BLOCK } x)$ where x is a free variable. The free variable is here to indicate that $(\text{BLOCK } x)$ takes the value TRUE when $x = A$ or $x = B$ and the value FALSE when x takes another value.

This reaction can be well adapted to the problem, but we claim that it may also happen that one has some information allowing to "improve". DG , i.e. one can often obtain another generalization DG' such that DG is more general than DG' .

In our example, let us suppose that one knows that each BLOCK has some color, i.e. the data basis contains the two pieces of information: $\forall x \exists y (\text{COLOR } y \ x)$ and RED is a COLOR: $\exists x (\text{COLOR RED } x)$.

It is therefore possible to transform $D1$ into $D1' = (\text{BLOCK } A) \& (\text{COLOR RED } A)$ and $D2$ into $D2' = (\text{BLOCK } B) \& \exists y (\text{COLOR } y \ B)$.

In that case, $DG' = DG$ since $\forall x \exists y (\text{COLOR } y \ x)$ implies $\exists y (\text{COLOR } y \ x)$. But if one considers

$D_3 = (\text{BLOCK } A) \ \& \ (\text{RED } B)$
 $D_4 = (\text{BLOCK } C)$

Then our "massaging" keeps the information that there are two objects, one of them being a block, the other one being coloured. If we assume that there exists at least one object which is coloured in red in our universe the generalisation of D_3 and D_4 should be
 $DG = (\text{BLOCK } x) \ \& \ \exists y (\text{COLOR RED } y).$

Besides, there examples being trivial no very significant information is kept by our massaging but we hope that everyone is aware that we introduce here the possibility of a deduction during generalization, less trivial examples will be given in the following.

III. PREDICATE GENERALIZATION

It is quite evident that any description is a kind of logical formula which takes the value TRUE on the examples.

Generalizing these logical formulas is recognized as an important field of A.I. research (cf [Hayes-Roth & Al. 78], [Michalski 83], [Dietterich & Al. 81], [Vere 80], ...). On the other hand, classical logics define generalization for terms (recursively defined on free variables and functions) and not for formulas (recursively constructed from logical connectives and predicates depending on free or bound variables). As far as we need in some way to be able to generalize formulas in order to achieve even an elementary learning, two reactions are possible. One is the one used in [Hayes-Roth & Al. 78] and in [Vere 80] : stick as much as possible to classical logics but define a new way of generalizing. The other reaction will be described here (and has already been presented in [Kodratoff 82] : stick as much as possible to the classical definition of generalization but change the way of computing the canonical form of a logical formula. Our main reasons are as follows.

3.1.1. Learning implies anyhow that the computation of a canonical form is not the same as in classical logics. For instance, in any rewriting system computing a canonical form (as well for the well-known disjunctive or conjunctive normal forms (that are not canonical) as for the new Hsiang form [Hsiang 82]) an expression known for being TRUE must be rewritten as TRUE. On the contrary, in a learning context, a description is nothing but a tautology known for being TRUE : you introduce $\text{SQUARE}(A)$ in your description only if you know that A is actually a square.

As far as classical logics must be violated, we want to be as precise as possible in the estimation of what is new and what remains in learning versus classical logics systems.

3.1.2. The classical definition leads to difficulties due to the necessity of introducing operators that separate variables during the generalization process [Hayes-Roth & Al. 78] [Vere 81].

It is easy to see (an example follows) that

this problem is linked to the idempotency of logical AND. As we shall also see in the following, our presentation of generalization takes this difficulty into account. This is now exemplified by the following problem, where $\&$ is the logical AND, $A, B, C, \dots$ are constants while $x, y, z, \dots$ are variables. Let

$E1 = \text{SQUARE}(A) \ \& \ \text{RED}(A) \ \& \ \text{SMALL}(A)$ (There is a small red square) and
 $E2 = \text{SQUARE}(B) \ \& \ \text{RED}(B) \ \& \ \text{SQUARE}(C) \ \& \ \text{SMALL}(C)$ (there are two squares, one is small, the other one is red).
 The problem is to find the "best" generalization Eg of $E1$ and $E2$.

A partial matching generalization will lead either to
 $Eg1 = \text{SQUARE}(x) \ \& \ \text{RED}(x) \ \& \ \text{SMALL}(y)$ or to
 $Eg2 = \text{SQUARE}(y) \ \& \ \text{RED}(x) \ \& \ \text{SMALL}(y)$
 which are, for $x \leftarrow A, y \leftarrow A$, both instances of $E1$ but, for $x \leftarrow B, y \leftarrow C$, only partial instances of $E2$.

An alternative solution is the use of $\&$ idempotency relative to $\text{SQUARE}(A)$, stating that :
 $E11 = \text{SQUARE}(A) \ \& \ \text{RED}(A) \ \& \ \text{SQUARE}(A) \ \& \ \text{SMALL}(A)$
 is totally equivalent to $E1$ (this is sure from the logic's point of view, but is it also sure from a learning point of view?).
 The generalization of $E11$ and $E2$ is :
 $Eg = \text{SQUARE}(x) \ \& \ \text{RED}(x) \ \& \ \text{SQUARE}(y) \ \& \ \text{SMALL}(y)$ which has $E1$ as instance for $x \leftarrow A, y \leftarrow A$ (and because $\text{SQUARE}(A) \ \& \ \text{SQUARE}(A)$ reduces to $\text{SQUARE}(A)$) and which has $E2$ as instance for $x \leftarrow B, y \leftarrow C$.
 A difficulty arises from the fact that $E2$ and $E3 = \text{SQUARE}(D) \ \& \ \text{RED}(D) \ \& \ \text{SQUARE}(E) \ \& \ \text{SMALL}(E)$ generalize also to Eg . Intuitively, this a bad feature since $(E1, E2)$ is a different set of examples than $(E2, E3)$ (clearly, the first one is more particular) and they lead to the same "knowledge". The solution to this problem, discussed in [Hayes-Roth & Al. 78], [Kodratoff 82] and [Vere 80] is that one should look at the instances which produce the generalization and mark two variables u, v that never have the same instances in one example by : $\text{DIFFERENT}(u, v)$, and two variables w, t that have the same instance in one example by $\text{SAME}(w, t)$. For instance, the generalization of $E1$ and $E2$ should be
 $Eg = \text{SQUARE}(x) \ \& \ \text{RED}(x) \ \& \ \text{SQUARE}(y) \ \& \ \text{SMALL}(y)$ since it is always implicitly stated that x and y can be instantiated by the same value, which happens in $E11 = E1$. On the contrary, the generalization of $E2$ and $E3$ should be :
 $Eg' = \text{SQUARE}(x) \ \& \ \text{RED}(x) \ \& \ \text{SQUARE}(y) \ \& \ \text{SMALL}(y) \ \& \ \text{DIFFERENT}(x, y)$ since the instances of x and y are different in $E2$ and $E3$ which implicitly contain the information $\text{DIFFERENT}(B, C)$ and $\text{DIFFERENT}(D, E)$ which should be used.

Marking each couple of variables when they have the same instantiation, or when they have different instantiation is equivalent to say that two variables which did not appear simultaneously in one SAME or in one DIFFERENT relationship, may be either identical, or unidentical. Another solution to this problem consists in marking two variables: either they have the same instantiation, with the SAME relationship, or they may have the same instantiation by the MBS (May Be the Same) relationship. So two variables which are not marked are implicitly DIFFERENT .

For instance the generalization of E_1 and E_2 should be with the above convention
 $E_g = \text{SQUARE}(x) \& \text{RED}(x) \& \text{SQUARE}(y) \& \text{SMALL}(y) \& \text{MBS}(x, y)$
 and the generalization of E_2 and E_3 should be
 $E'_g = \text{SQUARE}(x) \& \text{RED}(x) \& \text{SQUARE}(y) \& \text{SMALL}(y)$
 Refer to [Kodratoff 82] for more details on the non-determinacy which is introduced in that way.

3.1.3. Unification in equational theories is a well-known field of research (see for instance [Stickel 81] or [Martelli & Al. 82]) and matching in an equational theory (even a complicated one) is quite possible [Costa 81].

This kind of problem comes down to the research of a canonical form of terms. Instead of changing the classical definition of generalization, we felt it more economical to modify the definition of a canonical term and to introduce "learning canonical terms".

3.1.4. Besides, it is not at all counter-intuitive to state that the usual canonical form of a logical formula is not particularly well-suited in order to express the trivialities one should start from when learning.

3.1.5. In the following, we shall briefly recall the definitions of generalization, then describe our way to compute the canonical forms of a term (learning canonical forms) and discuss some of its consequences.

We shall continue to use the above notations as well as the following : $x \leftarrow A$ means that the variable x must be substituted by the constant A , V is the logical OR, $A < B$ means that A is more general than B , $A < > B$ means that A and B are not comparable from a generalization point of view.

3.2. Generalization

3.2.1. Generalization of terms

Let t_1 and t_2 be two terms. Saying that t_1 is more general than t_2 ($t_1 < t_2$) iff there is a substitution σ such that $\sigma t_1 = t_2$.

A substitution has the form $\sigma = (x_i \leftarrow t_i)$ where each x_i is a variable and each t_i is a term.

Example : $E_g = \text{SQUARE}(x) \& \text{RED}(x) \& \text{SQUARE}(y) \& \text{SMALL}(y)$

$E_2 = \text{SQUARE}(B) \& \text{RED}(B) \& \text{SQUARE}(C) \& \text{SMALL}(C)$

E_g is more general than E_2 since there exists a substitution σ such that $\sigma E_g = E_2$ where

$\sigma = (x \leftarrow B, y \leftarrow C).$

$E_{g1} = \text{SQUARE}(x) \& \text{RED}(x) \& \text{SMALL}(g)$

E_{g1} is not comparable with E_2 , indeed there does not exist any substitution such that $\sigma E_{g1} = E_2$.

3.2.2. Vere's definition of generalization [Vere 80]

3.2.2.1. Let us first consider conjunctions of elementary operators ; any expression A is supposed to take the form $A = A_1 \& \dots \& A_n$ where each A_i is elementary (i.e. contains no logical connectives). He defines the set associated to A by

$\{A\} = \{A_1, \dots, A_n\}.$

Then A is more general than B iff there is a substitution such that $\sigma A = B'$ where $\{B'\} \subset \{B\}$.

Otherwise stated, σA is equal to a subpart of B .

Example : with the Vere's definition $E_g < E_2$ and $E_{g1} < E_2$.

There exists in fact a substitution

$\sigma, \sigma = (x \leftarrow B, y \leftarrow C)$, such that $\sigma E_g = E_2$, which implies $E_g < E_2$. On the other hand, E_{g1} being the set associated with E_2 , and $\{\sigma E_{g1}\}$ the set associated with σE_{g1} , $\{\sigma E_{g1}\} \subset \{E_2\}$, so $E_{g1} < E_2$.

3.2.2.2. For disjunctions of conjunctions, this definition becomes :

let $G_a = g_{a1} \vee \dots \vee g_{an}$, $G_b = g_{b1} \vee \dots \vee g_{bm}$
 then G_a is more general than G_b iff $\forall j \exists i \ g_{ai} < g_{bj}$.

Example :

$F_1 = (\text{SQUARE}(x) \& \text{RED}(x)) \vee (\text{SMALL}(y) \& \text{BLOCK}(y) \& \text{GREEN}(y))$

$E_2 = \text{SQUARE}(B) \& \text{RED}(B) \& \text{SQUARE}(C) \& \text{SMALL}(C)$

$F_1 = F'_1 \vee F''_1$ and there exists σ such that

$F'_1 = E'_2$ with $\{E'_2\} \subset \{E_2\}$, $\sigma = (x \leftarrow B)$

Therefore F'_1 is more general than E_2 , and applying the above definition we can conclude that F_1 is more general than E_2 .

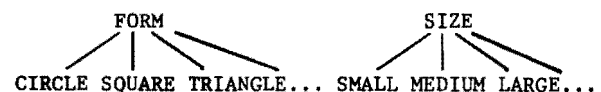
IV. OBTAINMENT OF A LEARNING CANONICAL FORM

There is an infinite number of "learning canonical forms" for a given term. On the other hand, we claim that there is a finite number of learning canonical forms of a term as compared to another. We shall always therefore speak of one of the canonical forms of t_1 and t_2 . Each of them is the least generalization of t_1 and t_2 which structurally matches with t_1 and t_2 . (Structural matching is defined in 4.2.1.). Actually if t structurally matches with t_1 and t_2 , then t is one of the generalizations of the learning canonical forms of t_1 compared with t_2 . Therefore structural matching may lead to the obtainment of learning canonical forms, and limit their number. Each of them is obtained by applying the same rules which depend either on the representation, as in the AND idempotency, or on the description, as in the use of the taxonomy of predicates. The order in which one attempts structural matching, the rules used, and the priority of rules define the strategy. For instance, let be G_1 and G_2 two terms :

$G_1 = (\text{CIRCLE } A) \& (\text{SMALL } A) \& (\text{SQUARE } B) \& (\text{LARGE } B) \& (\text{ON } A \ B)$

$G_2 = (\text{CIRCLE } C) \& (\text{SMALL } C) \& (\text{SQUARE } D) \& (\text{SMALL } D) \& (\text{LARGE } E) \& (\text{CIRCLE } E) \& (\text{ON } C \ D) \& (\text{ON } D \ E).$

We want to generalize G_1 and G_2 using idempotency of $\&$, and the two following predicates taxonomies :



If we take the same properties of descrip-

tions and representations for any given strategy of generalization, then the result depends on the strategy.

Let us suppose that, in the first strategy used, the predicates describing the shape of objects (i.e. descriptors which belong to the FORM's taxonomy) have the priority for structural matching. Using this strategy the generalization of G1 and G2 should be :

$$G' = (\text{CIRCLE } x_0) \& (\text{SIZE SMALL } x_0) \& (\text{SQUARE } y_0) \\ \& (\text{SIZE } u_0 y_0) \& (\text{CIRCLE } z_0) \& (\text{SIZE } v_0 y_0) \& \\ (\text{ON } x_0 y_0) \& (\text{ON } t_0 w_0) \& (\text{MBS } z_0 x_0) (\text{MBSt}_0 x_0) \\ \& (\text{MBS } t_0 z_0) \& (\text{MBS } w_0 y_0) \& (\text{MBS } t_0 y_0) \& \\ (\text{MBS } w_0 z_0).$$

On the other hand, if one wants to generalize G1 and G2 using another strategy giving priority to the predicates describing the size of the objects (i.e. descriptors which belong to the SIZE's taxonomy), we obtain the following generalization: $G'' = (\text{FORM CIRCLE } x_0) \& (\text{SMALL } x_0) \& (\text{FORM } u_0 y_0) \& (\text{LARGE } y_0) \& (\text{FORM } v_0 z_0) \& (\text{SMALL } z_0) \& (\text{ON } x_0 t_0) \& (\text{ON } z_0 y_0) \& (\text{MBS } x_0 z_0) \& (\text{MBS } y_0 t_0) \& (\text{MBS } t_0 z_0).$

Notice that we use a convention for two variables. Those which are not simultaneously present in one MBS or in one SAME relationship are implicitly different (see 3.1.2.). Now we may examine the two generalizations G' and G'' : they are not comparable ($G' < > G''$) from a generalization point of view. G' fits together variables which correspond to objects having the same shape, and G'', to objects having same size. Therefore the strategy indicating the order in which we have to consider the different descriptors is not artificial, it has consequences for the generalized formulae.

Notice that we do not claim that :

- our way of computing a learning canonical term is unique. Using another strategy than the one here presented corresponds to another type of learning. For instance, we distinguish a "civilian" learning (where the logical operators idempotency has priority : it amounts to using the same object several times in different situations) as opposed to a "military" learning where each object is destroyed when used (in practice, this can be taken into account also by using operators associated with numbers so that $\text{PERSHINGMISSILE}(x,n)$ is different from $\text{PERSHINGMISSILE}(x,n+1)$).

- starting from unique learning canonical forms of t1 and t2, one gets a unique generalization. On the contrary, reference [Kodratoff 82] explains how one may reduce the size of the combinatory explosion of all the possible different generalizations.

4.1. The generalization axioms

There are cases where a substitution cannot be found and where the generalization ordering is quite clear. Instead of using another definition of generalization we introduced the following axioms which order some terms relative to genera-

lization :

4.1.a. $\exists x P(x,y) < P(x,y)$

4.1.b. $P(A,y) < \forall z P(z,y)$

Axioms 4.1.a. and 4.1.b. deal with quantifiers but their intuitive meaning is not self-evident. To give some intuitive justification we have to take into account the following : let us suppose that E1 and E2 are two expressions, that $x_1, x_2, \dots, x_n$ are the free variables belonging to E1, and $y_1, y_2, \dots, y_p$ are the free variables belonging to E2.

If E1 and E2 have the following properties :

* $E1 < E2$

* $\{y_1, y_2, \dots, y_p\} \subset \{x_1, x_2, \dots, x_n\}$

Then, for all $(x_1, x_2, \dots, x_n)$ E2 is true implies that there exists same distribution of free variables which are not present in E2 such that E1 is true.

Example 1 :

E1 = $\text{SQUARE}(x) \& \text{RED}(x) \& \text{SMALL}(y) \& \text{SQUARE}(y)$

E2 = $\text{SQUARE}(x) \& \text{RED}(x) \& \text{SMALL}(x) \& \text{SQUARE}(x)$

* $E1 < E2$ because there exists σ such that $\sigma E1 = E2$: $\sigma = (x \leftarrow x, y \leftarrow x)$

* x which is the only free variable in E2 is present in E1. For each x E2(x) is true implies that there exists y ($y = x$) such that E1(x,y) is true.

Example 2 :

Another example may be more convincing : we shall demonstrate (see 4.2.4.) that (LOWER x y) is more general than (EQ x 1) & (EQ y 2). This proposition, not surprisingly, corresponds to our intuitive feeling because for all x and for all y , (EQ x 1) & (EQ y 2) implies (LOWER x y).

Now if we try to apply the above assertion to formulae which contain quantifiers, we can verify that it is, in fact, true :

* Axiom 4.1.a : $\exists x P(x,y) < P(x,y)$. Applying the last assertion to this axiom leads us to : for all x and for all y $P(x,y) \Rightarrow \exists x P(x,y)$ which is always true.

* Axiom 4.1.b : $P(A,y) < \forall z P(z,y)$

Similarly axiom 4.1.b leads to : for all y $\forall z P(z,y) \Rightarrow P(A,y)$ which is a tautology.

4.2. The elementary rewriting steps

4.2.1. Structural matching

Let us say that t1 matches structurally t2 when t1 becomes identical to t2 (modulo a variable renaming) when all the leaves of t1 and t2 are given variables names. The messaging introduced in section 2.2. indicates that learning canonical forms structurally match each other as far as possible.

4.2.2. Using idempotency

It amounts to a use of idempotency in order to favor a structural matching of t1 and t2.

For instance, suppose that $t_1 = P \& Q$ and $t_2 = Q$, one then uses idempotency when one tries to find if Q can take the form $Q' \& Q''$ such that Q' structurally matches P . The term t_2 is then rewritten as $t_2' = Q' \& Q$ which is logically equivalent to t_2 but allows us to attempt a matching of $t_2 = t_2'$ and t_1 .

4.2.3. Detecting a matching

Given t_1 and t_2 , one attempts to find substitutions such that $\sigma t_1 = t_2$ or $\sigma t_2 = t_1$.

4.2.4. Using knowledge of descriptors

When two expressions do not structurally match, we can enforce the match by using rules describing the semantic of descriptors. As we shall see in 4.2.5. if no structural match does appear, one may "lose" information by "abstracting" terms. We illustrate, in the following example an application of these rules :

It is intuitively clear that $(\text{LOWER } X \ Y)$ is more general than $(\text{EQ } X \ 1) \ \& \ (\text{EQ } Y \ 2)$, but actually, this is not easy to demonstrate. In order to do this, we need to use some semantic rules :

- (1) $\forall X (\text{EQUAL } X \ X) = \text{True}$
- (2) $(\text{LOWER } 1 \ 2) = \text{True}$
- (3) $(\text{LOWER } X \ Y) (\text{EQUAL } X \ Z) = (\text{LOWER } Z \ Y)$
- (4) $(\text{LOWER } X \ Y) (\text{EQUAL } Y \ Z) = (\text{LOWER } X \ Z)$

Now let us suppose that :

$T_1 = (\text{LOWER } X_1 \ Y_1)$
 $T_2 = (\text{EQ } X_2 \ 1) \ \& \ (\text{EQ } Y_2 \ 2)$

There is no substitution σ such that $\sigma T_1 = T_2$. In order to demonstrate that T_1 is more general than T_2 , first we generalize T_1 and T_2 .

Let T the least generalization of T_1 and T_2 . Then we try to prove that $T = T_1$. In fact $T < T_2$ and $T = T_1$ implies that $T_1 < T_2$.

Obtainment of T :

We attempted structural matching between T_1 and T_2 , but it failed : there is no descriptor EQ in T_1 and no descriptor LOWER in T_2 . We try to make apparent these descriptors by using semantic rules.

Therefore we obtain :

$T'_1 = (\text{LOWER } X_1 \ Y_1) \ \& \ \forall X (\text{EQ } X \ X)$
 $T'_2 = (\text{EQ } X_2 \ 1) \ \& \ (\text{EQ } Y_2 \ 2) \ \& \ (\text{LOWER } 1 \ 2)$

As there is no universal quantifier in T'_1 , we make it disappear by using the axiom 4.1.b' :

$T''_1 < T'_1$
 $T''_1 = (\text{LOWER } X_1 \ Y_1) \ \& \ (\text{EQ } X_1 \ X_1) \ \& \ (\text{EQ } Y_1 \ Y_1)$
 $T'_2 = (\text{LOWER } 1 \ 2) \ \& \ (\text{EQ } X_2 \ 1) \ \& \ (\text{EQ } Y_2 \ 2)$

We try to detect the match between T''_1 and T'_2 .
 $\sigma_1 = (X_1 \leftarrow X_1, 1 ; Y_1 \leftarrow Y_1, 2)$

$T''_1 = (\text{LOWER } X_0 \ Y_0) \ \& \ (\text{EQ } X_0 \ X_0) \ \& \ (\text{EQ } Y_0 \ Y_0)$
 $T'_2 = (\text{LOWER } X_0 \ Y_0) \ \& \ (\text{EQ } X_2 \ X_0) \ \& \ (\text{EQ } Y_2 \ Y_0)$

There is no match because, in T'_2 , X_2 appears in the $(\text{EQ } X_2 \ X_0)$ and the only variables in T''_1 are X_0 and Y_0 . So T''_1 is too particular, and we generalize it by creating two new variables :

$T'''_1 < T''_1$
 $T'''_1 = (\text{LOWER } X_0 \ Y_0) \ \& \ (\text{EQ } X_3 \ X_0) \ \& \ (\text{EQ } Y_3 \ Y_0) \ \& \ (\text{SAME } X_3 \ X_0) \ \& \ (\text{SAME } Y_3 \ Y_0)$

Now we may obtain T which is the least generalization of T_1 and T_2 , by using the substitution $\sigma_2 = (U_0 \leftarrow X_3, X_2 ; V_0 \leftarrow Y_3, Y_2)$

$T = (\text{LOWER } X_0 \ Y_0) \ \& \ (\text{EQ } U_0 \ X_0) \ \& \ (\text{EQ } V_0 \ Y_0) \ \& \ (\text{MBS } X_0 \ U_0) \ \& \ (\text{MBS } Y_0 \ V_0)$

The reader can check that T is more general than T_1 and T_2 . Now, using rules (3) and (4) we obtain :

$T = (\text{LOWER } U_0 \ V_0)$ which implies :
 $T(X_0, Y_0, U_0, V_0) = T_1(U_0, V_0)$.

4.2.5. Developping or reducing to TRUE or FALSE values

One can also attempt to achieve the structural match between t_1 and t_2 by using the tautologies :

$A \ \& \ \text{TRUE} = A$ and $A \ \vee \ \text{FALSE} = A$, in either direction depending on t_1 and t_2 . For instance, suppose that $t_1 = P(x) \ \& \ Q(x)$ and $t_2 = P(x)$, then we can try to prove that $Q(x) = \text{TRUE}$ so that t_1 reduces to t_2 or $Q(A) = \text{TRUE}$ so that the canonical form of t_2 is $t_2' = P(x) \ \& \ Q(A)$. In each case, the learning canonical forms of t_1 and t_2 are brought to structural matching.

4.2.6. Description transformation

One may bring t_1 and t_2 to structural matching by considering that the "missing operators" of, say, t_1 are actually variables of a more general operator which must be introduced into the description (how to invent these new operators is not considered here. As the following examples show one needs a good knowledge of the universe semantics). For instance, suppose that one wishes to compare $t_1 = P(x) \ \& \ Q(x)$ and $t_2 = P(x)$.

Instead of using idempotency, it may be more useful to consider P and Q as instances of (functional) variables and find some more general operator G such that $G(P, x)$ and $G(Q, x)$ have the same interpretation than $P(x)$ and $Q(x)$. This is useful when one has some information about $G(y, x)$. For instance, consider the assertion $\text{RED}(x)$ (there are objects that can have the red hue), it may be useful to rewrite it as $\text{COLOR}(\text{RED}, x)$ when, for instance, one knows that all the objects in our universe are colored, i.e. $\forall x \exists y \text{COLOR}(y, x) = \text{TRUE}$, especially in view of applying 4.2.5.

4.3. Abstraction

In theory, this brings us to second order logic but in practice this may well reduce to a kind of hierarchy among operators which is considered as being part of the knowledge basis describing our universe.

As in the example (see IV.), the generalization of G_1 and G_2 leads either to G' which intro-

duces general descriptor, SIZE, or to G" which introduces general descriptor FORM.

Nevertheless it happens that whatever description transformation is used, a description which appears in an example may not appear in another. In this case structural matching is impossible. In order to generalize, one may remove the term, but this lead to forgetting information. This is the classical A.I. way to generalize, which is symbolized by the use of the "Dropping condition rule" [Dietterich & Al. 81].

We could criticize this way of generalizing because it does not follow the classical logic definition of generalization, unless we assume that "True" is more general than any particular assertion and "False" less general.

For instance, let $E1 = P(y) \& Q(z)$ and $E2 = P(x)$. On one hand Vere's definition leads to $E2 < E1$ and, in the same way, the "Dropping condition rule" used by Michalski (see [Dietterich & Al. 81]) leads to $E2 < E1$. On the other hand in our theory, if $P=Q$ then $E1 < E2$ (see 4.4.8.), unless $P \neq Q$. However if P and Q belong to the same predicate taxonomy then the two formulae $E1$ and $E2$ are comparable from a generalization point of view only if $z \neq y$ (i.e. z and y are not the same parameter) (see 4.4.2.). In this case $E2 < E1$ (see 4.4.4.). Lastly if P and Q do not belong to the same predicate taxonomy than $E2 < E1$ (see 4.4.3. and 4.4.5.). If we assume as Vere and Mikalski, that whatever P and Q are, $E2$ is more general than $E1$, this implies if $P(x) = \text{True}$ that $\text{True} < Q(z) \& \text{True}$. Therefore, using the fact that $Q(z) \& \text{True} = Q(z)$, this leads to $\text{True} < Q(z)$ whatever Q may be. In the same way, it is easy to demonstrate that with Vere and Michalski's definition "False" is more particular than any other assertion.

In the case studied above, there is no way to obtain the structural matching unless we "lose" some pieces of information, but we claim that it is not permissible to omit anything during the generalization process. In order to go through the structural matching, we need to make an abstraction: This operation is equivalent to removing the term which makes the matching impossible, while keeping this term in memory. Therefore, after the generalization process, the least generalization will match with the examples, modulo terms which have been abstracted. Because we have kept these terms, we can precisely verify the matching, and analyze the failure of generalization by reading these terms. We can therefore conclude that, first, learning becomes an exact process, because nothing is lost, and secondly, the generalization process may guide the changes in description by showing the result and the terms abstracted; thereby displaying redundancies and lacks of descriptions.

4.4. Examples of the consequences of our definitions

We give examples into a universe of colored shapes so that $\text{RED}(A)$ can be rewritten as $\text{COLOR}(\text{RED}, A)$ and $\text{BLOCK}(A)$ as $\text{SHAPE}(\text{BLOCK}, A)$. Notice that we assume that it impossible to learn from partial descriptions into universes where this

kind of knowledge is not known. The starred examples are more or less in contradiction with the usual definitions of generalization and they will be further commented.

4.4-1. $\text{RED}(A) < \text{RED}(A) \& \text{BLOCK}(A)$

*4.4-2. $\text{RED}(A) < > \text{RED}(A) \& \text{BLUE}(A)$ (in general)

4.4-3. $\text{RED}(A) < \text{RED}(A) \& \text{BLOCK}(B)$

*4.4-4. $\text{RED}(A) < \text{RED}(A) \& \text{BLUE}(B)$

4.4-5. $\text{RED}(x) < \text{RED}(y) \& \text{BLOCK}(y)$

*4.4-6. $\text{RED}(x) < > \text{RED}(y) \& \text{BLUE}(y)$

4.4-7. $\text{RED}(x) < \text{RED}(y) \& \text{BLOCK}(z)$

*4.4-8. $\text{RED}(y) \& \text{RED}(z) < \text{RED}(x)$

4.4-9. $\text{RED}(A) \vee \text{BLOCK}(A) < \text{RED}(A)$

*4.4-10. $\text{RED}(A) \vee \text{BLUE}(A) < \text{RED}(A)$ (under some conditions)

4.4-11. $\text{RED}(A) \vee \text{BLOCK}(B) < \text{RED}(A)$

4.4-12. $\text{RED}(y) \vee \text{BLOCK}(y) < \text{RED}(x)$

*4.4-13. $\text{RED}(y) \vee \text{BLUE}(y) < \text{RED}(x)$ (under some conditions)

4.4-14. $\text{RED}(y) \vee \text{BLOCK}(z) < \text{RED}(x)$

4.4-15. $\text{RED}(y) \vee \text{RED}(z) < \text{RED}(x)$

Proving 4.4-2.

In general, $\text{RED}(A) \& \text{BLUE}(A)$ is equivalent to an other operator, either FALSE or REDBLUESPOTTED(A) or ROSY(A) etc... Using a distance notion which is intuitively clear in that case (and altogether not defined in this paper), replacing $\text{RED}(A) \& \text{BLUE}(A)$ by one of its equivalent form leads to an expression "nearer" from $\text{RED}(A)$ (because a failure of matching with & is avoided), and therefore those equivalent forms are canonical and $\text{RED}(A)$ is not comparable to any of them (unless it so in a very specific universe).

Proving 4.4-4.

One changes the description and $t1 = \text{RED}(A)$ becomes $t1' = \text{COLOR}(\text{RED}, A)$ and $t2$ becomes $t2' = \text{COLOR}(\text{RED}, A) \& \text{COLOR}(\text{BLUE}, B)$. In $t1$ one has the knowledge that B has a color (which is not valid in regard of $t2$ but because one supposes that all objects of the universe are colored, this may be untrue and $t1$ and $t2$ would then be uncomparable), i.e. $\exists x \text{COLOR}(x, B) = \text{TRUE}$. $t1'$ becomes $t1'' = t1' \& \text{TRUE}$

$= \text{COLOR}(\text{RED}, A) \& \exists x \text{COLOR}(x, B)$
 $= \exists x (\text{COLOR}(\text{RED}, A) \& \text{COLOR}(x, B))$

which is more general than $t2'$ by 4.1.a;

Proving 4.4-8.

One uses idempotency and

$t1 = \text{RED}(y) \& \text{RED}(z)$ is more general than
 $t2 = \text{RED}(x) \& \text{RED}(x)$ because $\sigma t1 = t2$,
 where $\sigma = (y + x, z + x)$.

Proving 4.4-10.

Let $t1 = \text{RED}(A) \vee \text{BLUE}(A)$ and $t2 = \text{RED}(A)$. We change the representation and write :
 $t1 = t1' = \text{COLOR}(\text{RED}, A) \vee \text{COLOR}(\text{BLUE}, A)$ (which, as opposed to 4.4-2. has generally no implication) and
 $t2 = t2' = \text{COLOR}(\text{RED}, A)$.
 There are two cases :
 either A is the object which has all colors, i.e.
 $\forall x \text{ COLOR}(x, A) = \text{TRUE}$, then $t1 < > t2$
 or A is not spotted by all colors, i.e.
 $\forall x \text{ COLOR}(x, A) = \text{FALSE}$ and it follows :
 $t2' = t2' \vee \text{FALSE}$
 $= \text{COLOR}(\text{RED}, A) \vee \forall x \text{ COLOR}(x, A)$
 $= \forall x (\text{COLOR}(\text{RED}, A) \vee \text{COLOR}(x, A))$
 and $t1' < t2'$ because of 4.1.b.

Proving 4.4-13.

The same proof as in 4.4-10. applies, except that one must be in a universe where there is not at all any object spotted by all colors, i.e.
 $\forall x \text{ COLOR}(x, y) = \text{FALSE}$ to be able to compare the two clauses.

V. ALGORITHM FOR GENERALIZING

5.1. Difficulties of such an algorithm

Besides the well-known difficulties due to the use of Associative-Commutative functions, same supplementary problems are introduced by the use of idempotency and semantic rules. The central problem is the obtainment of a learning canonical form : it is solved by an incremental matching process which manage it.

Our approach may lead to obtain a great number of distinct generalizations, but some of them only are usefull. The role of the strategy is to diminish the size of the search space. The strategy may depend itself on the kind of problem we have to solve. It indicates the order in which descriptors must be considered. For instance, in a block world, the central point may be the size of the blocks, or their shape or the above relationship between blocks, etc..., or it may be independent of the problem semantic, and for example, take only into account the number of variables which simultaneously appear in the generalized formula.

5.2. Description of algorithm

Step 1

Verify that each descriptor which appears at least in one example, appears in all the examples. If this is not the case, we try to make appear these descriptors by using some semantic rules. If this is impossible we need to make an "abstraction" (see 4.3.).

Step 2

One strategy being given, we choose the des-

criptors on which we have to generalize first.

Step 3

We mark each occurrence of apparition of each descriptor having been chosen in step 2, on each example, in order to compute the different possible generalizations.

Example :

Let us suppose that our strategy leads us to generalize examples E_1 , E_2 and E_3 on the descriptor SMALL :

$E_1 = (\square B) \& (\bigcirc C) \& (\text{LARGE } A) \& (\text{SMALL}_1 B) \& (\text{SMALL}_2 C) \& (\text{INSIDE } B A) \& (\text{ABOVE } A C) \& (\text{ABOVE } B C)$
 $E_2 = (\square D) \& (\bigcirc f) \& (\text{LARGE } E) \& (\text{SMALL}_3 F) \& (\text{SMALL}_4 D) \& (\text{INSIDE } F E) \& (\text{ABOVE } D E) \& (\text{ABOVE } D F)$
 $E_3 = (\square G) \& (\bigcirc H) \& (\bigcirc I) \& (\text{LARGE } H) \& (\text{SMALL}_5 G) \& (\text{SMALL}_6 I) \& (\text{INSIDE } I H) \& (\text{ABOVE } G H) \& (\text{ABOVE } G I)$

We may firstly generalize on SMALL_1 , SMALL_3 and SMALL_5 , or on SMALL_1 , SMALL_4 and SMALL_6 , or ... : we have to envisage 8 possible ways for generalizing.

Step 4

We envisage separably each possible generalization. In each one we substitute one variable to the term which appears under the scope of the descriptor having been chosen. We operate simultaneously on all the examples.

Step 5

Let be x_0 the variable that we have substituted in step 4. We try to obtain one learning canonical form in regard to x_0 . One have to verify that all the occurrences of x_0 are the same in all the examples.

If this is not the case :

- firstly, we try to make appear each occurrence of x_0 on all the examples by using either representation rules (see 4.2-2. or 4.2-5.) or knowledge about description (see 4.1., 4.2-3. or 4.2-6).
- Secondly, if we fail to make appear one occurrence of x_0 on at least one example, we make disappear this occurrence on all the examples in which it appears by creating a new variable.

Notice that we use the convention described in 3.1.2. : we mark two variables which are identical by using the SAME relationship, and two variables which may be the same by the MBS relationship. Two variables which does not appear in any SAME or MBS relationship are implicitly different.

Step 6

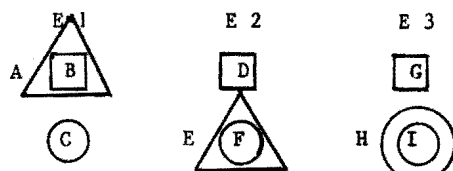
We return to step 2 until there does not appear any occurrence of variable or constant, under the scope of a descriptor, which may not be the same in all the example that we want to generalize.

Notice that the substitutions of new vari-

ables, and the "massaging" which are introduced by them, must not transform any occurrence of variables which have been previously substituted.

5.3. Example

Let be E1, E2 and E3 the three examples which have been presented in [Hayes-Roth & Al. 78] :



Each example is described by a conjunction of clauses. Each of them describes either an object, or a spatial relation between two objects. There are 5 descriptors for objects : Δ which means triangle, $\square$ square, $\bigcirc$ circle, SMALL small size and LARGE large size. There are 2 descriptors for spatial relation : INSIDE and ABOVE.

E1 = (Δ A) & ($\square$ B) & ($\bigcirc$ C) & (LARGE A) & (SMALL B) & (SMALL C) & (INSIDE B A) & (ABOVE A C) & (ABOVE B C)
 E2 = (Δ E) & ($\square$ D) & ($\bigcirc$ F) & (LARGE E) & (SMALL D) & (SMALL F) & (INSIDE F E) & (ABOVE D F) & (ABOVE D E)
 E3 = ($\bigcirc$ H) & ($\square$ G) & ($\bigcirc$ I) & (LARGE H) & (SMALL G) & (SMALL I) & (INSIDE I H) & (ABOVE G H) & (ABOVE G I).

Let us assume that there are two predicate taxonomies. The first relates the predicates describing the shape of the objects (Δ , $\square$, $\bigcirc$). We call it the FORM taxonomy. The second relates the predicates describing the size of the objects, we call it the SIZE taxonomy.



We shall omit the & in the following.

Step 1

The descriptor Δ which appears in E1 and E2 does not appear in E3. The predicate taxonomy of FORM permits us to transform the description :

E1 = (FORM Δ A) (FORM $\square$ B) (FORM $\bigcirc$ C) (LARGE A) (SMALL B) (SMALL C) (INSIDE B A) (ABOVE A C) (ABOVE B C)
 E2 = (FORM Δ E) (FORM $\square$ D) (FORM $\bigcirc$ F) (LARGE E) (SMALL D) (SMALL F) (INSIDE F E) (ABOVE D F) (ABOVE D E)
 E3 = (FORM $\bigcirc$ H) (FORM $\square$ G) (FORM $\bigcirc$ I) (LARGE H) (SMALL G) (SMALL I) (INSIDE I H) (ABOVE G H) (ABOVE G I).

Step 2

Now suppose our strategy manage us to minimize the number of generalization. This guide us to choose the descriptor LARGE or the descriptor INSIDE : there are five descriptors. Let be $N(E_i, D)$ the number of occurrences of descriptor D in the example E_i . The number of ways to generalize on D is equal to : $N(E1, D) * N(E2, D) * N(E3, D)$

Each choice of descriptor leads to a different generalization. The role of the strategy is either to minimize the number of possible generalization, or to indicate the most meaning full descriptor which has to be the focal point in the generalization process. Whatever the strategy is, it leads to some generalizations which are usually not comparable each other from a generalization point of view.

Let us assume now that our strategy contains another criteria which manage us to generalize first, with the descriptors which qualify the objects, as FORM, LARGE and SMALL, and secondly, with the descriptors which describe relationship between objects, as INSIDE and ABOVE.

In our case, the strategy leads us to generalize on LARGE.

Step 3

There is only one occurrence of LARGE in each of the example, therefore, there is only one possible way of generalizing.

Step 4

We substitute x_0 to A, E and H in all the examples :

E1 = (FORM Δ x_0) (FORM $\square$ B) (FORM $\bigcirc$ C) (LARGE x_0) (SMALL B) (SMALL C) (INSIDE B x_0) (ABOVE x_0 C) (ABOVE B C)
 E2 = (FORM Δ x_0) (FORM $\square$ D) (FORM $\bigcirc$ F) (LARGE x_0) (SMALL D) (SMALL F) (INSIDE F x_0) (ABOVE D F) (ABOVE D x_0)
 E3 = (FORM $\bigcirc$ x_0) (FORM $\square$ G) (FORM $\bigcirc$ I) (LARGE x_0) (SMALL G) (SMALL I) (INSIDE I x_0) (ABOVE G x_0) (ABOVE G I)

Step 5

Occurrences of x_0 for each example

E1	(LARGE x_0) (FORM Δ x_0)	(INSIDE B x_0) (ABOVE x_0 C)
E2	(LARGE x_0) (FORM Δ x_0)	(INSIDE F x_0) (ABOVE D x_0)
E3	(LARGE x_0) (FORM $\bigcirc$ x_0)	(INSIDE I x_0) (ABOVE G x_0)
	(LARGE x_0)	(INSIDE x_0)

But (ABOVE x_0 b) appears only in E1 and no rule permits to make appear it in E2 and E3, in the same way (ABOVE b x_0) appears in E2 and E3 and does not appear in E1.

We have to make disappear (ABOVE x_0 C) from E1, (ABOVE D x_0) from E2 and (ABOVE G x_0) from E3

E1 = (FORM Δ x_0) (FORM $\square$ B) (FORM $\bigcirc$ C) (LARGE x_0) (SMALL B) (SMALL C) (INSIDE B x_0) (ABOVE u₀ C) (ABOVE B C) (SAME u₀ x_0)
 E2 = (FORM Δ x_0) (FORM $\square$ D) (FORM $\bigcirc$ F) (LARGE x_0) (SMALL F) (INSIDE F x_0) (ABOVE D F) (ABOVE D v₀) (SAME v₀ x_0) (SMALL D)
 E3 = (FORM $\bigcirc$ x_0) (FORM $\square$ G) (FORM $\bigcirc$ I) (LARGE x_0) (SMALL G) (SMALL I) (INSIDE I x_0) (ABOVE G w₀) (ABOVE G I) (SAME x_0 w₀)

Now step 6 manages us to go to step 2. This leads us to generalize either on (FORM Q b) or on (FORM □ b).

1 - Generalization on (FORM Q b)

$y_o \leftarrow C, F, I$

E1 = (FORM Δ x_o)(FORM □ B)(FORM Q y_o)(LARGE x_o)
(SMALL B)(SMALL y_o)(INSIDE B x_o)(ABOVE u_o y_o)
(ABOVE B y_o)(SAME u_o x_o)
E2 = (FORM Δ x_o)(FORM □ D)(FORM Q y_o)(LARGE x_o)
(SMALL D)(SMALL y_o)(INSIDE y_o x_o)(ABOVE D y_o)
(ABOVE D v_o)(SAME v_o x_o)
E3 = (FORM Q x_o)(FORM □ G)(FORM Q y_o)(LARGE x_o)
(SMALL G)(SMALL y_o)(INSIDE y_o x_o)(ABOVE G w_o)
(ABOVE G y_o)(SAME x_o w_o)

Step 5

(ABOVE u_o y_o) which appears in E1 does not appear in E2 and E3. In the same way (INSIDE y_o x_o) which appears in E2 and E3 does not appear in E1. On one hand, we can make appear (ABOVE u_o y_o) in E2 and E3, by the use of & idempotency. On the other hand, there is no way to make appear (INSIDE x_o y_o) in E1. Therefore we are leading to the following generalization :

E1 = (FORM Δ x_o)(FORM □ B)(FORM Q y_o)(LARGE x_o)
(SMALL B)(SMALL y_o)(INSIDE B x_o)(ABOVE u_o y_o)
(ABOVE B y_o)(SAME u_o x_o)
E2 = (FORM Δ x_o)(FORM □ D)(FORM Q y_o)(LARGE x_o)
(SMALL D)(SMALL y_o)(INSIDE y_o x_o)(ABOVE D y_o)
(ABOVE D v_o)(SAME v_o x_o)(SAME z_o y_o)
(ABOVE t_o y_o)(SAME t_o D)
E3 = (FORM Q x_o)(FORM □ G)(FORM Q y_o)(LARGE x_o)
(SMALL G)(SMALL y_o)(INSIDE z_o x_o)(ABOVE G w_o)
(ABOVE G y_o)(SAME x_o w_o)(SAME z_o y_o)
(ABOVE t_o y_o)(SAME t_o G)

2 - Generalization on (FORM □ b)

$y_o \leftarrow B, D, G$

Using the same rewriting steps leads us to the following generalization :

E1 = (FORM Δ x_o)(FORM □ y_o)(FORM Q C)(LARGE x_o)
(SMALL y_o)(SMALL C)(INSIDE z_o x_o)(ABOVE u_o C)
(ABOVE y_o C)(SAME u_o x_o)(SAME z_o y_o)
(ABOVE y_o t_o)(SAME t_o C)
E2 = (FORM Δ x_o)(FORM □ y_o)(FORM Q F)(LARGE x_o)
(SMALL y_o)(SMALL F)(INSIDE F x_o)(ABOVE y_o F)
(ABOVE y_o v_o)(SAME v_o x_o)
E3 = (FORM Q x_o)(FORM □ y_o)(FORM Q I)(LARGE x_o)
(SMALL y_o)(SMALL I)(INSIDE I x_o)(ABOVE y_o w_o)
(ABOVE y_o I)(SAME x_o w_o)

If we pursue the generalization process we obtain two possible generalizations.

The first one corresponds to the first generalization on (FORM Q b) :

E1 = (FORM Δ x_o)(FORM □ k_o)(FORM Q y_o)(LARGE x_o)

(SMALL k_o)(SMALL y_o)(INSIDE z_o x_o)
(ABOVE u_o y_o)(ABOVE k_o y_o)(SAME u_o x_o)
(SAME k_o z_o)(ABOVE k_o p_o)(SAME p_o y_o)

E2 = (FORM Δ x_o)(FORM □ k_o)(FORM Q y_o)(LARGE x_o)
(SMALL k_o)(SMALL y_o)(INSIDE z_o x_o)(ABOVE k_o y_o)
(ABOVE k_o v_o)(SAME v_o x_o)(SAME z_o y_o)
(ABOVE t_o y_o)(SAME t_o k_o)

E3 = (FORM Q x_o)(FORM □ k_o)(FORM Q y_o)(LARGE x_o)
(SMALL k_o)(SMALL y_o)(INSIDE z_o x_o)
(ABOVE k_o w_o)(ABOVE k_o y_o)(SAME x_o w_o)
(SAME z_o y_o)(ABOVE t_o y_o)(SAME t_o k_o)

This leads to the following generalization formula :

EG1 = (FORM u_o x_o)(FORM □ k_o)(FORM Q y_o)
(LARGE x_o)(SMALL k_o)(SMALL y_o)(INSIDE z_o x_o)
(ABOVE t_o y_o)(ABOVE k_o y_o)(ABOVE k_o v_o)
(MBS t_o x_o)(MBS v_o x_o)(MBS v_o y_o)(MBS y_o z_o)
(MBS k_o z_o)(MBS k_o t_o)

We can verify that EG1 is more general than E1, E2 and E3 :

$\sigma'_1 = (u_o \leftarrow \Delta, x_o \leftarrow A, y_o \leftarrow C, k_o \leftarrow B, z_o \leftarrow B,$
 $v_o \leftarrow C, t_o \leftarrow A)$
 $\sigma'_2 = (u_o \leftarrow \Delta, x_o \leftarrow E, y_o \leftarrow F, k_o \leftarrow D, z_o \leftarrow F,$
 $v_o \leftarrow E, t_o \leftarrow D)$
 $\sigma'_3 = (u_o \leftarrow Q, x_o \leftarrow H, y_o \leftarrow I, k_o \leftarrow G, z_o \leftarrow I,$
 $v_o \leftarrow H, t_o \leftarrow G)$

The reader can easily check that
 $\sigma'_1 EG1 = E1, \sigma'_2 EG1 = E2$ and $\sigma'_3 EG1 = E3$

The second possible generalization which corresponds to the first generalization on (FORM □ b) is equal to EG1, modulo a variable renaming.

Now, let us assume that, the focal point in learning is not the descriptions of objects, but their spatial relationship. In this case we first generalize on the descriptor INSIDE.

Therefore we obtain one generalization :

EG2 = (FORM P_o x_o)(FORM q_o k_o)(FORM r_o y_o)
(LARGE x_o)(SMALL k_o)(SMALL y_o)(INSIDE y_o x_o)
(ABOVE w_o t_o)(ABOVE u_o v_o)(MBS u_o x_o)
(MBS w_o y_o)(MBS k_o v_o)(MBS t_o k_o)
(MBS t_o v_o)(MBS t_o x_o)
(MBS v_o y_o)(MBS u_o k_o)(MBS w_o k_o)
(MBS u_o w_o)

The reader can easily verify that EG2 is more general than E1, E2 and E3, but that it is not comparable from a generalization point of view, with EG1.

BIBLIOGRAPHY

- [Bongard 70] Bongard, N., "Pattern Recognition", Spartan Books, New York, 1970.
[Clavieras & Al. 83] Clavieras, B. and Ganascia, J.-G., Lemerle-Loisel, R., "Généralisation de différences entre descriptions", Actes Journées sur les outils de l'apprentissage à partir

d'exemples, Orsay, 1983.

- [Costa 81] Costa, E., "Dérécursivation automatique en utilisant des systèmes de réécriture de termes, Thèse de 3ème cycle, Paris, 1981, Pub. by L.R.I., Bât. 490, F 91405 ORSAY CEDEX, Pub. Int. n° 118, 1982.
- [Dietterich & Al. 81] Dietterich, G.T. and Michalski, S. R., "Inductive learning of Structural Descriptions : Evaluation Criteria and Comparative Review of Selected Methods", Artificial Intelligence 16, pp. 257-294, 1981.
- [Hayes-Roth & Al. 78] Hayes-Roth, F. and Mc Dermott, J. "An interference matching technique for inducing abstractions", Com. ACM, 21, pp. 401-411, 1978.
- [Hsiang 82] Hsiang, J., "Topics in automated theorem proving and program generation", Ph. D. thesis, Dept. Comp. Sci. Univ. Illinois, Urbana, novembre 1982.
- [Kodratoff 82] Kodratoff, Y., "Generalizing and particularizing as the techniques of learning", Proc. Internatl. Conf. on A. I. and Information-Control of Robots, Bratislava, pp. 131-134, 1982 ; see also the same, submitted to Computers and Artificial Intelligence.
- [Martelli & Al. 82] Martelli, A. and Montanari, U., "An efficient Unification", Algorithm. A.C.M. Vol. 4, n° 2, 1982.
- [Michalski 83] Michalski, R.S., "Inductive learning as rule-guided transformation of symbolic descriptions : a theory and implementation", Proc. Internatl. Workshop on Program Construction, Bonas, 1980, to appear in "Automatic program construction techniques", A. Biermann, G. Guiho, Y. Kodratoff eds., Mc Millan, 1983.
- [Plotkin 70] Plotkin, G. D., "A note on inductive generalization", in Machine Intelligence B. Meltzer & D. Michie eds., American Elsevier, New York, 1970.
- [Sallantin 83] Sallantin, J., "Méthodologie pour la construction de règles à partir de données binaires", Actes Journées sur les outils de l'apprentissage à partir d'exemples, Orsay, 1983.
- [Shank 82] Shank, R. C., "Looking at Learning", Proc. ECAI-82, pp. 11-18, Orsay, 1982.
- [Stickel 81] Stickel, M. E., "A unification algorithm for associative-commutative functions", J. ACM 28, PP. 423-434, 1981.
- [Vere 80] Vere, S.A., "Multilevel counterfactuals for generalizations of relational concepts and productions", Artificial Intelligence, J. 14, pp. 139-164, 1980.
- [Vere 81] Vere, S.A., "Constrained N-to-1 generalizations", unpublished draft, pp. 23, feb. 1981.

ESCAPING BRITTLINESS

John H. Holland
 Dept. of Computer and Communication Sciences
 The University of Michigan
 Ann Arbor, Michigan 48109

ABSTRACT

Expert systems are powerful when working within the domain-specific boundaries of the initial design. It is widely agreed that the usefulness of such systems would be greatly enhanced if they could be made more versatile or less brittle -- tolerant of changes in the domain and underlying model. This paper suggests seven criteria for escaping brittleness [combination, parallelism, declarative and procedural information, categorization, synchronic and diachronic pointing, gracefulness, and confirmation] and gives an example of a class of general-purpose systems, *classifier systems*, that attempt to meet these criteria.

Key terms: Apportionment of credit, Brittleness, Bucket-brigade algorithm, Classifier system, Expert system, Genetic algorithm, Knowledge structure, Learning, Parallelism, Production system.

Expert systems are one of Artificial Intelligence's real successes in the exploration of intelligence. In the longer view, the most significant part of this success may be the bright illumination it throws upon questions of system versatility. Expert Systems, as currently structured, are narrowly domain-specific and brittle in the sense that they cannot respond to slight changes in domain without substantial human intervention (see Duda and Shortliffe [1983]). A close comparison of the structures of Expert Systems with what we know of the structures of natural intelligence provides some interesting criteria for escaping brittleness:

1. Combination: The knowledge structures and models used should be constructed from a well-defined set of elements that can serve as "building blocks". The set of elements may be rich and varied, but it should be clear how the elements can be combined to yield the structures needed. This requirement, if properly implemented, forestalls ambiguous definitions and serves as a powerful deterrent to ad hoc constructions. More importantly, it makes possible comparison of different structures in terms of the elements they hold in common. Without this possibility, discovery of similarities and analogies, transfer of information from one situation to another, and a wide range of other activities vital to flexible problem-solving, become quite difficult. Additionally, structures defined from "building blocks" possess rich, com-

binatorial possibilities for combating the complexity of the environment.

2. Parallelism: In order to avoid a distinct element (or disjoint sets of elements) for each individual object encountered (a "grandmother" node, a "yellow volkswagon with a flat tire" node, etc., etc.) it must be natural to treat the simultaneous activity of an array of elements. This is part and parcel of the discussion of combinatorics. Stated another way, it is the simultaneous invocation of a set of hypotheses at various levels of generality that enables the system to model new situations by transfer of information and conjectures from previous experience.

3. Declarative and procedural information: The "building blocks", when combined, must readily incorporate *both* the declarative ("propositional") and the procedural ("state-transition") information inherent in models.

4. Categorization: It must be simple for inductive procedures to generate and test elements, and combinations of elements, that represent the categories and other generalizations underpinning the models used. Categorization is the system's *sine qua non* for combating the environment's perpetual novelty. Accordingly, it must be easy to describe processes for generating and testing categories. Once again, unless the elements themselves directly implement categorizations, it becomes difficult to provide simple processes for transferring knowledge from one situation, or structure, to another.

5. Synchronic and diachronic pointing: Synchronic pointers -- association of categories to form analogies, hierarchies of generalization, default hierarchies, etc. -- and diachronic pointers -- temporal association yielding transitions and expectation -- are the threads from which models are constructed. As such they must be easily generated and modified by the inductive processes.

6. Gracefulness: It should be simple to insert new, tentative inductions into the overall structure without disturbing well-established procedures and models (except where the new information or induction suggests a critical modification). To this end, the elements constituting the cognitive structure should act as a network of interacting, competing hypotheses. The competition replaces requirements for global consistency with one of progressive confirmation relative to the situations in which the hypotheses are used. This

makes it much easier for the system to incorporate and use new information. Elements, and structures constructed from them, should be easily modified by combinations of bottom-up (input-driven) and top-down (category-driven) procedures. Bottom-up procedures, properly implemented in this context, generalize new information and place it in the cognitive structure at appropriate places for access and use; top-down procedures yield plausible defaults and support categories.

7. Confirmation: As the system accumulates experience the relative reliability (and usefulness) of various hypotheses (elements and combinations of elements) must be readily adjustable and must reflect itself directly in the competition of hypotheses. There is also a second kind of confirmation that must emerge from the combination of elements. A hypothesis based on a very general category, and tentatively invoked in some situation, should "support" and activate more specific conjectures that are to be tested and used to confirm, or disconfirm, the more general hypothesis.

CLASSIFIER SYSTEMS

Classifier systems are general-purpose programming systems designed as an attempt to meet the criteria set forth in the preceding section. Classifier systems have many affinities to the rule-based (production) systems underpinning the usual approach to Expert Systems (see Waterman and Hayes-Roth [1978] for many examples), but with the following major differences:

1. Any number of rules, called *classifiers*, can be active at the same time. There can be no direct conflict between classifiers because the only action of a classifier is to post a message to a global message list -- more active classifiers simply mean more messages on the list. The resulting conflict-free concurrency sidesteps the difficult conflict-resolution problems of one-rule-at-a-time systems (see McDermott and Forgy [1978]), allowing the system to use many rules concurrently to summarize and act upon a situation. The rules become "building blocks" with extensive possibilities for combination and recombination. Moreover, the parallelism makes it easier to specify and control the parallel processes that pervade the real world.

2. Messages are binary strings of fixed length k . Classifiers, following the usual convention for productions, consist of a *condition* part and an *action* part. Because the system only processes messages of fixed length, each condition has a uniform format, specifying the set of messages that can satisfy it. Specifically, conditions are all specified by strings of length k over the alphabet $\{1, 0, \#\}$ where the $\#$ plays the role of a "don't care" symbol. If $M = m_1 m_2 \dots m_k$ is a message and $C = c_1 c_2 \dots c_k$, $c \in \{1, 0, \#\}$, is a condition, then the message M satisfies condition C just in case $m_j = c_j$ wherever $c_j = 1$ or 0 , and m can be either 1 or 0 wherever $c_j = \#$. This definition makes it possible to use a simple matching operator to test whether or not some message satisfies a condition. Despite these specializations, classifier systems are probably general-purpose -- able to execute any algorithm that can be defined in a language

such as LISP or C.

3. When the condition part of a classifier (one or more conditions) is satisfied by some message on the global message list, the action part uses the message to form a new message which is posted to the list. Thus the basic procedure of the system is a simple loop in which each classifier accesses the current message list, determines if its conditions are satisfied and, if so, posts a message to the message list for the next time-step. Since messages are simply uninterpreted strings (taking their only meaning from the rules they activate) there can be no conflict in the posting of messages -- any number of distinct messages can be posted on the list for $t+1$.

4. All external communication (input and output) is via messages to the message list. As a result, all internal control information and external communications reside in the same data structure.

5. Because of the global nature of the message list, tagging (the use of particular bit combinations as prefixes, suffixes, etc.) and related techniques become efficient ways of coupling classifiers, providing predetermined sequences of execution, subroutines, hierarchical organization, diachronic and synchronic pointing, etc. The combination of concurrency and a global list avoids the difficulties discussed by Davis and King [1977] in their review of production systems.

6. Because the order in which classifiers are executed is independent of their order in storage, and because satisfaction of conditions is determined by simple matching, there is no need for an interpreter. This makes it possible to design local, syntactic operators that modify systems in useful ways. For example, tags can be modified in ways that gracefully change the behavior of the system. In particular, a system can gracefully accept new rules, and various kinds of learning procedures (see LEARNING PROCEDURES below) can be imposed, without disturbing well-established (useful) procedures.

This system can be extended in a number of ways that make it easier to use (e.g., adding not-conditions, message processing, etc. -- for details the reader should see Holland [1980]). These additions are important, but the essence of the system is the parallelism that permits both a "building block" approach to the processing of information and a competition between rules. Successful full-scale tests of classifier systems have been run at several institutions (see, for example, Booker [1982], Goldberg [1983], Smith [1980], or Wilson [1982]).

LEARNING

Two kinds of learning algorithm are required if a classifier system is to adapt to changes in the domain and goals presented it. The first is an algorithm that reinforces or apports credit to rules already available to the system. (See Samuel's [1959] cogent discussion of this problem.) The second is an algorithm for generating new, plausible rules when the ones available prove inadequate. (Samuel calls this "the parameter prob-

lem"). This section describes one example of the first kind, called the *bucket-brigade algorithm* and one example of the second kind, called the *genetic algorithm*.

The bucket brigade algorithm is designed to assign a value to each classifier, called its *strength*, that is adjusted to reflect the classifier's net usefulness as a participant in goal-seeking sequences. It is important that the algorithm do this correctly for classifiers that, early in the sequence, set the stage for later successful actions. (For example, in checkers, this is the problem of crediting an early move that makes possible a triple jump some moves later.) The algorithm works by introducing a competition between classifiers through a modification of the basic match cycle. Satisfied classifiers must compete to get their messages on the message list; each satisfied classifier makes a *bid* based on its strength and only the highest bidding classifiers place their messages on the list. The size of the bid depends not only on the strength, but also on the specificity of the classifier's conditions. (For classifier conditions, specificity is easily measured: the more don't cares (#'s) in the condition, the less specific it is.) Thus, a high bid is indicative of "relevance" (high specificity) and "past usefulness" (high strength).

Formally, when the conditions of a classifier C are satisfied, it makes a bid

$$B(C,t) = cR(C)S(C,t)$$

where $R(C)$ is the specificity, equal to the number of non-#'s in the condition part divided by the length thereof, $S(C,t)$ is the strength of C at time t, and c is a constant $\ll 1$ (say 1/8 or 1/16).

The *winning* (high) bidders place their messages on the message list and have their strength *reduced* by the amount of the bid.

$$S(C,t+1) = S(C,t) - B(C,t)$$

(Each winner pays for the privilege of posting a message.) The classifiers {C'} that sent the messages *matched* by this winner have their strengths *increased* by the amount of the bid (sharing among them).

$$S(C',t+1) = S(C,t) + aB(C,t),$$

where $a = 1/(\text{no. of members of } \{C'\})$. (The senders are rewarded for setting up a situation usable by C.)

The bucket-brigade algorithm treats each classifier as a kind of "middleman" in a complex economy, the strength of a classifier measuring its ability to turn a "profit". As a middleman, the classifier only deals with its suppliers -- the classifiers that send messages satisfying its conditions -- and its consumers -- the classifiers that are in turn satisfied by the messages it sends. Whenever a classifier wins a bidding competition, it initiates a transaction in which it pays out part of its strength to its suppliers, and then receives similar payments from its con-

sumers. Its strength is a kind of capital. If a classifier receives more from its consumers than it paid out, its strength is increased. This is likely to occur only if the consumer in turn is profitable. This chain leads to the ultimate consumers, the classifiers that directly attain goals and receive payoff directly from the environment. That is, certain actions are immediately rewarded or reinforced by the environment; this *payoff* for goal-attainment is added to the strengths of all classifiers active at that time. The profitability of other classifiers depends upon their being coupled into sequences leading to payoff. Thus, the bucket-brigade assures that early-acting, "stage-setting" classifiers receive credit if they (on average) make possible later, overtly rewarding acts.

(It is worth noting that some of the fixed-point theorems of mathematical economics provide a way of proving the above for environments that have "stable" statistics.)

Once we can assign strengths to classifiers we have a basis for generating *new* classifiers to enter the competition. In broadest terms, the *genetic algorithm* uses high-valued classifiers as progenitors for new classifiers to be tested under the bucket brigade. Because there are many winners of the competition on each time-step, i.e., the system is still highly parallel, it is possible to test newly generated classifiers without seriously disrupting the system's repertoire in well-practiced situations (see below).

Genetic algorithms have been studied intensively through analysis (Holland [1975] and Bethke [1980] and simulation (De Jong [1975], Smith [1980], Booker [1982], and others). It is vital to the understanding of genetic algorithms to know that even the simplest versions act much more subtly than "random search with preservation of the best", contrary to the conclusions one might reach from a common misreading of genetics as a process primarily driven by mutation.

The central operator of a genetic algorithm is called *crossover*. It acts by exchanging a (randomly chosen) segment in a pair of strings. For example, it might exchange some initial portion of the condition parts of two classifiers; or it might exchange the middle portion of a condition with the middle portion of a message specification. This simple, seemingly non-mathematical, operator produces quite subtle effects under the competition induced by the bucket brigade. In effect, the genetic algorithm favors the construction of *new* rules from "building blocks" (called *schemata*) that appear as components of classifiers already proved useful. The "parent" rules simply supply copies of their parts for use by the crossover operator; they remain in the system in their original form.

The "building blocks" or schemata, so manipulated by the genetic algorithm, may involve tags or other portions of messages that determine classifier coupling and control sequencing. Thus, the genetic algorithm can encourage variants of useful subroutines and it can generate hierarchical substructures for testing. Theoretical work

and simulations (Holland [unpublished]) make it clear that newly generated classifiers can be introduced to the system without seriously disturbing capabilities already well-established. Classifiers with high strength will always score high in any competition in which they are relevant. In such a competition a newly generated classifier (starting, say, with average strength) will be quite unlikely to be a winner. On the other hand, new rules get their chance when no high-strength rule is very relevant to a situation; then the new rule may be tried and rewarded, or not, by the bucket brigade. This quality of the competition, combined with the classifier system's ability to act coherently without maintaining global consistency, permits the system to maintain a wide range of competing, plausible inductions. The system continually modifies its knowledge structures on the basis of subsequent (dis-) confirmations, while exploiting well-established structures in familiar situations.

[Classifier systems are currently being tested with the help of a program, CS1, that is both a "compiler", allowing one to design and simulate classifier systems on a standard computer, and a "test-bed", providing the means of simulating a wide range of environments for testing the learning algorithms. The current version provides the following facilities: (1) Simulation of a task environment consisting of up to 256 objects, each with up to 32 distinct features, emplaced on a 65,000 by 65,000 grid. Any, or all, of the objects may be mobile. (2) An input interface that consists of an arbitrarily shaped "vision cone" that views a local part of the surface (typically less than 1000 grid points) and uses feature detectors to generate an input message for each object in the vision cone. (3) An emulator for the classifier system that can retain, in random access memory, the description of over 2000 classifiers and a message list of up to 32 messages. This part of the system is written in machine language and can execute a basic time-step (all classifiers matched against all messages) in about 0.1 second. The overall system runs in close to real-time, making it convenient to run long learning sequences in the simulated environment. (4) An output interface that permits manipulation of objects on the grid, movement over the grid, rotation of the visual cone, and in fact any other effector action conveniently specifiable by a subroutine. (5) Parametrised versions of both the bucket-brigade and genetic algorithms, including provisions for contingent activation of these algorithms (such as activation of the genetic algorithm when no classifier responds, to an input message, with a high bid).]

REFERENCES

- Bethke, A. D., *Genetic Algorithms as Function Optimizers*, Ph.D. Dissertation (Computer and Communication Sciences), The University of Michigan, 1980.
- Booker, L., *Intelligent Behavior as an Adaptation to the Task Environment*, Ph.D. Dissertation (Computer and Communication Sciences), The University of Michigan, 1982.
- Davis, R. and King, J., "An Overview of Production Systems", *Machine Intelligence 8*, Elcock, E. W. and Michie, D. (Eds.), American Elsevier, New York, pp. 300-331, 1977.
- DeJong, K. A., "Adaptive System Design -- A Genetic Approach", *IEEE Transactions: Systems, Man, and Cybernetics*, Vol. 10, No. 9, 1980.
- Duda, R. O. and Shortliffe, E. H., "Expert Systems Research", *Science*, Vol. 220, pp. 261-268, 1983.
- Goldberg, D., [Dissertation in progress - Civil Engineering], The University of Michigan, 1983.
- Holland, J. H., "Adaptive Algorithms for Discovering and Using General Patterns in Growing Knowledge Bases", *International Journal of Policy Analysis and Information Sciences*, Vol. 2, pp. 217-240, 1980.
- McDermott, J. and Forgy, C., "Production System Conflict Resolution Strategies", *Pattern-Directed Inference Systems*, Waterman, D. A. and Hayes-Roth, F. (Eds.), Academic Press, New York, pp. 177-199, 1978.
- Samuel, A. L., "Some Studies in Machine Learning Using the Game of Checkers", *IBM Journal of Research and Development*, Vol. 3, pp. 211-232, 1959.
- Smith, S., *A Learning System Based on Genetic Algorithms*, Ph.D. Dissertation (Computer Science), The University of Pittsburgh, 1980.
- Waterman, D. A. and Hayes-Roth, F. (Eds.), *Pattern-Directed Inference Systems*, Academic Press, New York, 1978.
- Wilson, S., "Adaptive 'Cortical' Pattern Recognition", [Internal Report, Research Laboratories, Polaroid Corporation], 1982.

TWO PROGRAMS FOR TESTING HYPOTHESES OF ANY LOGICAL FORM

Clark Glymour

Kevin Kelly

Richard Scheines

Department of History and Philosophy of Science
 1017 Cathedral of Learning
 University of Pittsburgh
 Pittsburgh, PA 15260

ABSTRACT

Programs for machine learning from instances or "structural descriptions" have been limited by restrictions on the logical form of the conclusions inferred. We describe two programs for testing hypotheses of any logical form. The first implements a qualitative confirmation relation due to Hempel. The second, and more useful, implements an approximation to any of several quantitative confirmation relations derivative from Hempel's.

I. INTRODUCTION

Many machine learning programs attempt to infer generalizations from data represented or straightforwardly representable by singular sentences in first-order logic. Such learning programs typically generate conclusions that can also be represented as first-order sentences, usually with some quantifier structure. The conclusions that can be drawn using such programs are inevitably limited by whatever restrictions the program imposes on the logical form of hypotheses, and in most cases these limits are severe. Hunt's (1966) "Concept Learning Systems", for example, were in effect restricted to hypotheses with monadic predicates, a single universal quantifier, and a biconditional form with a single predicate on one flank of the biconditional. A program described by Vere (1975) is restricted to conjunctions with universal quantifiers. An impressive program due to Shapiro (1981) is restricted to Horn Sentences, that is, to universal conditions with conjunctive antecedents and atomic consequences. These restrictions essentially limit the expressive power of the conclusions that can be inferred by the respective systems. In our judgement, an important aspect of the progress of machine learning programs in the last twenty years has been the liberalization of restrictions on the logical form of conclusions, but existing programs cannot yet generate from instances hypotheses that are not subject to prior restrictions on logical form, and even hypotheses with so simple a form as $VxEy(Pxy \vee Pyx)$ seem excluded at present.

A computer program for efficiently testing (or confirming) hypotheses of arbitrary logical form would be a significant step towards a

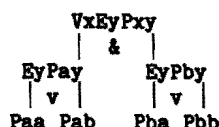
machine induction program with the same generality. We describe below two programs which, given an hypothesis H of any first-order logical form and given an instance or evidence sentence E which is a conjunction of atomic sentences or their negations, determine whether or not E confirms H . The formal ideas behind these programs derive from the philosophical literature on inductive logic, and especially from the work of Carl Hempel. Coupled with an appropriate generator, on which we are now working, the second of these programs offers the prospect of significantly liberalizing the restrictions on logical form in machine learning. The programs described below are written in MACLISP and are implemented on the computer facilities of Carnegie-Mellon University.

II. THE HEMPEL PROGRAM

The first account of confirmation that we have implemented is due to C.G. Hempel (1965a), who developed it for first-order languages without identity, although it has subsequently been extended by Glymour (1980) to languages with identity and definite descriptions. Hempel's confirmation relation is purely qualitative, that is, a singular sentence either confirms an hypothesis or it doesn't, and confirmation has no other degrees. Hempel's account of confirmation satisfies several formal conditions, the most important of which for our purposes is that if an evidence sentence confirms an hypothesis then it confirms all hypotheses logically equivalent to that hypothesis, and analogously for disconfirmation. Let E be a singular sentence in which the individual constants $a_1, \dots, a_n$ occur essentially, and let H be any first-order sentence without identity. According to Hempel, E confirms H if and only if the sentence H is valid in every model of E in which all members of the domain are named by one of the constants $a_1, \dots, a_n$. Thus Paa confirms each of the following: $VxPxx$, $VxVyPxy$, $VxEyPxy$, etc., but does not confirm $ExEyPxy$. (The evidence sentence need not be conjunctive in Hempel's theory, but it seems almost always to be assumed to be conjunctive in machine learning problems, and we will keep that restriction throughout.) Evidence E fails to confirm H if E does not confirm H ; E disconfirms A if E confirms the denial of H .

Hempel shows that E confirms H if and only if E logically entails a certain variable-free sentence determined by H and the individual

constants occurring essentially in E. This sentence, the development of H for $a_1, \dots, a_n$, can be obtained by iteration starting with the first quantifier occurring in the formula, deleting that quantifier and, if the quantifier is universal, conjoining all formulas obtained by substituting one by one the constants $a_1, \dots, a_n$ for all occurrences of the freed variable, or if the quantifier is existential disjoining the substitution instances. Equivalently, the development can be represented as a tree with the hypothesis as root, each vertex other than the root and leaves corresponding either to a conjunction or to a disjunction, and atomic sentences or their negations (basic sentences) as leaves. So represented, the development of $\forall x \exists y Pxy$ for the constants a, b is



Our program estimates Hempelian confirmation provided three conditions are met: the evidence sentence is a singular conjunction; the hypothesis to be tested is expressed so that no quantifier occurs within the scope of a negation; and the hypothesis to be tested contains no predicates not occurring in the evidence sentence. The program executes a depth first search of the development tree of the hypothesis, returning "T" for a leaf if the corresponding formula is found among the conjuncts of the evidence sentence, and "nil" otherwise. Vertices whose daughters are obtained by instantiating an existentially quantified variable are counted as disjunctive vertices, and analogously for universal quantifiers and conjunctive vertices. A disjunctive vertex is evaluated as "T" if and only if at least one of its daughters is evaluated as "T", and a conjunctive vertex is evaluated as "T" if and only if all of its daughter vertices are evaluated "T". Negations are distributed so as to have the smallest possible scope. The depth first search strategy is such that whenever the fragment of the tree already evaluated suffices logically to determine a truth value for the hypothesis to be tested (under the valuation determined by the evidence sentence), the generation and evaluation of the development tree cease and the appropriate value, T or nil, is returned. "nil" signifies that the hypothesis is not confirmed; it does not necessarily mean that the negation of the hypothesis is confirmed.

The program contains a parameter m that can be adjusted by the user when large bodies of evidence or hypotheses with many quantifiers are to be considered. The parameter can be set to any value less than or equal to the number of individual constants occurring in the evidence sentence. With the value of m less than the number of individual constants, the program behaves as described above, except that at each quantifier vertex, there are only m randomly selected daughter vertices. Instead of

instantiating each quantified variable with all of the individual constants, only m randomly selected constants are used. The selection of the m constants is performed independently at each quantifier vertex. When m has a value less than the number of individual constants the program output is no longer certain to be the result that Hempel's confirmation theory would give, but is only an estimate of that result. Coincidentally, of course, the program can return the "correct" result even though some of the data are in error.

When the program is given as evidence all conjuncts $P a_i a_{j+1} \& \neg P a_i a_j$ with $i, j = 1, 2, 3, 4, 5$ and $j \neq i+1$, the hypothesis characteristic of discrete order, $\forall x \forall y (Pxy \rightarrow \neg \exists z (Pxz \& Pzy))$ is confirmed in 0.4 seconds with $m = 5$, and in 0.27 seconds with $m = 2$. The same instance fails to confirm the hypothesis characteristic of dense order and likewise fails to confirm transitivity. With $m = 5$ these determinations require 0.24 and 0.44 seconds respectively.

Canfield (1962) pointed out, and Hempel (1965a) himself agrees, that Hempel's confirmation theory is excessively strict. It fails to allow confirmation by partial instances. Thus $Paa \& Pab \& Pba$ fails to confirm $\forall x \forall y Pxy$ because the conjunct Pbb does not occur in the evidence. The computer program shows the same feature. The evidence sentence consisting of all conjuncts $P a_i a_j$ with $i, j = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$ and $i < j$ seems as though it ought to lend some support to the hypothesis of transitivity -- that is, to $\forall x \forall y \forall z ((Pxy \& Pyz) \rightarrow Pxz)$ -- but in fact the hypothesis is not confirmed with $m = 10$. (The decision takes about 0.3 seconds). With $m = 2$ twelve trials produced no confirmation of the hypothesis and with $m = 1$ seven of thirteen trials produced no confirmation. Hempel's confirmation theory is too conservative to be useful.

III. THE PARTIAL INSTANCES PROGRAM

There are many formal ways in which to develop a confirmation measure that will extend Hempel's ideas by giving some weight to partial instances. One can let the degree of confirmation of H on E equal the proportion of the models of E (in which each element of the domain is named by an individual constant occurring essentially in E) that are also models of H. The result is a conditional probability measure, but it has the unattractive consequence that the degree of confirmation of a universal hypothesis cannot be greater than 1/2 unless it is 1. We develop a different idea, deriving from the intuition that since the evidence $Paa \& Pab \& Pba$ provides three of the four conjuncts necessary for Hempelian confirmation of the hypothesis $\forall x \forall y Pxy$, the degree of confirmation of that hypothesis on that evidence should be 3/4. One formal measure that captures this intuition is obtained by putting the development of H in full conjunctive normal form, call it H^* . In H^* , give a sentential component that is either atomic or the negation of an atomic formula the score 1 if that component is among the conjuncts of E, give it the score 0 if its negation is among the conjuncts of E, and give it

the score 0 otherwise; give a disjunction the score 1 if at least one of its disjuncts has the score 1, give it the score * if all of its disjuncts have the score *, and give it the score 0 otherwise; give a conjunction (that is, H^*) a degree of confirmation equal to the proportion of its conjuncts that have the score 1, provided none of its conjuncts have the score *, and otherwise give it degree of confirmation 0. Another measure of the same sort can be obtained by considering the minimal conjunctive normal forms of H and, because the minimal conjunctive normal form is not always unique (Hohn, 1966), applying some choice function. Still other measures can be obtained from the disjunctive normal forms. All of these measures give about the same result in the simple and intuitive cases, and they all have the same essential feature, namely they are not probability measures. The sum of the confirmation of H on E and the confirmation of $\neg H$ on E , need not be unity by any of these measures. They are closer to Shafer's (1976) belief functions than to probabilities. We find no powerful theoretical reason for preferring any one of these normal form measures to the others, and any of them will, unlike Hempel's theory of confirmation, permit partial instances to give some weight to hypotheses.

It is not practical actually to determine conjunctive normal forms of developments of hypotheses, and the Partial Instances Program we have implemented is only a crude approximation to any of the measures described above. The program distributes negations so as to give them the smallest scope or cancel them, and it distributes conjunction over disjunction, but it does not convert a formula such as $(p \vee q) \wedge (\neg p \vee r)$ into the equivalent but shorter $(q \vee r)$. Because of the scoring procedure, the Partial Instances Program, unlike the Hempel Program in all but the worst case, must typically search the entire development. Computation times are therefore significantly longer. The same sampling strategy is used in the Partial Instances program as is used in Hempel. The scoring procedure approximates the normal form measures by requiring that the confirmation of a disjunction is the confirmation of the best confirmed disjunct, by requiring that the confirmation of a conjunction is in the simplest case the ratio of confirmed conjuncts to unconfirmed conjuncts, and by working recursively on the logical structure of the development of the hypothesis to be tested. More exactly, the scoring procedure of the Partial Instances Program is as follows: a leaf (an atomic sentence or its negation) is scored (1,1) if it is among the conjuncts of the evidence sentence, * if its negation is among the conjuncts of the evidence sentence, and (0,1) otherwise; a conjunctive node is given the vector sum of the score of its daughter nodes provided none of the latter are scored *, and is given the score * otherwise; a disjunctive node is given the score x/y if x/y is the maximum ratio of the components of the scores of daughter nodes, provided some daughter node has a value other than *, and is given the score * otherwise. A

score * at the root indicates that the hypothesis has been disconfirmed in Hempel's sense, i.e., that its negation has been confirmed to degree 1.

When the Partial Instances Program is given the evidence sentence consisting of all conjuncts P_{aj} with $i, j = 1, 2, 3, 4, 5, 6, 7$, and $i < j$, the hypothesis $VxVyVz (P_{xy} \wedge P_{yz}) \rightarrow P_{xz}$ is given degree of confirmation $5/12$ with $m = 7$ in 6.4 seconds. The price of avoiding the computation of normal forms is that the program gives different degrees of confirmation to logically equivalent hypotheses, on the same evidence. We believe that this incoherence will prove harmless so long as hypotheses are generated for testing in a standard form. No loss in the possible expressive power of conclusions generated by a machine learning system is entailed by restrictions on logical form, if every hypothesis is logically equivalent to an hypothesis of the form permitted. We are at work on an hypothesis generator which will consider in order of simplicity open formulas that are instantiated at least once by any model of the evidence sentence. For each open sentence, the sentences in prenex form that can be obtained from it by prefixing quantifiers will be tested, one by one. Control of acceptance of hypotheses and cessation of generation and testing will be governed by a combined index of degree of confirmation and explanatory power, where the latter will be based on a computational approximation of a measure developed by Hempel and Oppenheim (1965b).

REFERENCES

- Canfield, J. "On the Paradox of Confirmation", *Metrika*, 5, pp. 105-18, 1962.
- Glymour, C. *Theory and Evidence*. Princeton: Princeton University Press, 1980.
- Hempel, C. "Studies in the Logic of Confirmation", *Aspects of Scientific Explanation*, Glencoe: The Free Press, 1965a.
- Hempel, C. and Oppenheim, P. "Studies in the Logic of Explanation", *Ibid*, 1965b.
- Hohn, F. *Applied Boolean Algebra*. New York: Macmillan, 1966.
- Hunt, E. *Experiments in Induction*. New York: Academic Press, 1966.
- Shafer, G. *A Mathematical Theory of Evidence*. Princeton: Princeton University Press, 1978.
- Shapiro, E. *Inference of Theories from Fact*. Research Report 192, Yale University, 1981.
- Vere, S. "Induction of Concepts in the Predicate Calculus", 4th IJCAI Proceeding, 1975.

Research for this report was supported by the National Science Foundation and the University of Pittsburgh.

LEARNING EQUATION SOLVING METHODS FROM WORKED EXAMPLES

Bernard Silver
Dept. of Artificial Intelligence
University of Edinburgh
Hope Park Square
Meadow Lane
Edinburgh EH8 9NW, U.K.

ABSTRACT

This paper describes LP, a program that learns new techniques for solving equations by examining worked examples. Unlike most of the work in this field, where the equations used have been very simple, LP uses equations of A level standard. (A levels are exams taken at 18, and are used for university selection.)

In order to be able to successfully use a new technique, LP learns many different types of information. At the lowest level, LP compares consecutive lines in the worked example, finding differences between them. This allows the program to learn new rewrite rules.

LP also tries to discover the strategic purpose of each step, expressed in terms of satisfying preconditions of following steps. From this viewpoint, the worked example can also be considered as a type of plan for solving the equation. LP extracts the necessary information, and builds a plan which is stored for future use. LP executes the plan in a flexible way to solve new equations.

Key terms: Learning, learning by example, strategy learning, equation solving, planning, meta-level inference, PROLOG.

I. INTRODUCTION

This paper describes LP, a program which learns new techniques for solving symbolic equations. LP learns the new techniques by examining worked examples. These techniques can then be tested on some new problems. LP is built around the equation solving program PRESS, [Bundy and Welham 81, Sterling et al 82], and is intended to be part of a self-improving algebra system, [Bundy and Sterling 81].

The equations used by LP are symbolic, transcendental, non-differential equations, e.g.

$$\log_2 x + 4 \cdot \log_x 2 = 5.$$

1.1. Motivation

The motivation for LP arose from our equation solving program PRESS, [Bundy and Welham 81, Sterling et al 82]. This is a non-learning program, the only way to increase the ability of PRESS is to write new code. We have a test set of equations, and each version of PRESS is primarily judged by how many of these equations it can solve. Over several years, new equation solving methods have been added to PRESS, and its performance has markedly increased. However, coding and debugging the new methods can be tedious, and we set ourselves the goal of building a program which could learn new methods automatically.

Other workers have dealt with learning equation solving methods, but our interests are somewhat different.

Most of the other work in this field, e.g. [Brazdil 78, Neves 78, Cotton et al 81, Langley 82], have concentrated on very simple equations, such as

$$2 \cdot x + 3 = 5.$$

The machine learning community have undoubtedly found such work interesting. It has been demonstrated that computer programs can learn to solve equations, building up from very little knowledge. Also, some of this work may prove interesting to educationalists, particularly the programs that attempt to model childrens' errors.

However, not only beginners learn, so do experts and people in between these two extremes. We required a learning program with abilities nearer the expert end of the spectrum. While it would be nice for the program to build up to expert performance from a beginners knowledge, we felt that such an approach would be impractical. A more realistic goal is to build a program that starts with abilities around the "fairly good" mark, and learns to the standard of "very good".

LP is able to learn a new method by examining a worked example which uses the new technique. We feel that a single example should be sufficient. As Neves points out in [Neves 78], many textbooks present just one worked example before proceeding to the exercises.

This consideration rules out many concept-learning methods, e.g. [Winston 75, Mitchell et al 81, Young et al 77], as these programs generally need to process several instances before a concept is learned.

1.2. Worked Examples

A worked example should contain enough detail so that the reader (either a human or a program) can understand the technique being demonstrated, but there should not so much detail that the important points are swamped.

We now give a typical instance of the type of worked example used by LP:

$$\cos(x) + 2.\cos(2.x) + \cos(3.x) = 0 \quad (i)$$

$$2.\cos(2.x).\cos(x) + 2.\cos(2.x) = 0 \quad (ii)$$

$$2.\cos(2.x).(\cos(x) + 1) = 0 \quad (iii)$$

$$\cos(2.x) = 0 \text{ or } \cos(x) + 1 = 0$$

$$\cos(2.x) = 0$$

$$x = 180.n1 + 90$$

$$\cos(x) + 1 = 0$$

$$\cos(x) = -1$$

$$x = 180.(2.n2 + 1)$$

The key step in this worked example is the transformation from equation (i) to equation (ii).

II. THE METHODS OF LP

In this section we describe the basic operators of LP, called methods. These are modified PRESS methods.

Equations are solved by applying methods, which in turn may apply rewrite rules. An example of a method is Collection, which applies to equations containing more than one occurrence of the unknown. Collection reduces the number of occurrences of the unknown.* Collection is used in the worked example above to obtain line (iii) from line (ii). This step uses the Collection rule

$$U.V + U.W \Rightarrow U.(V + W).$$

Two other methods that will be referred to later are Factorization and Change of Unknown. Factorization transforms equations of the form

$$E_1.E_2 \dots E_n = 0$$

into

$$E_1 = 0 \vee E_2 = 0 \vee \dots \vee E_n = 0,$$

and solves each factor separately.

The Change of Unknown method transforms equations of the form

$$f(g(x)) = 0,$$

where g occurs at least twice in f , into

$$f(y) = 0, \quad (iv)$$

where

$$y = g(x).$$

The equation (iv) is called the changed variable equation. If it is solved, producing the answer $y = a$, say, then solving the equation

$$g(x) = a \quad (v)$$

solves the original equation. Equation (v) is called the substitution equation.

A method consists of a set of rules applicable to that method, and some information telling when and how the rules should be applied. For the case of rule-learning, in [Bundy & Silver 82], we call this control information. The control information includes preconditions, facts which must be true before the method can be applied, and postconditions which must be true after the application of the method. The postconditions are used both to ensure that the method has achieved the desired effects, and to allow planning.

Many learning programs work in domains where the basic operators are similar to those of STRIPS. The operators of STRIPS, [Fikes et al 72], have preconditions, an add list which contains the facts that the operator makes true, and the delete list which contains the facts that are no longer true after the application of the operator. If the preconditions are satisfied the operator can be applied.

In contrast, LP methods do not have this desirable property. The preconditions of a method represent necessary, but not sufficient conditions. In general, a method is not certain to succeed, even if the preconditions are applicable. This is because the preconditions are too general, but we can not give stronger preconditions that do not involve actually applying the method to test if it is applicable! It seems that this might be a problem in many domains.

*If the number can be reduced to one, another method, Isolation, can be applied to solve the equation.

Similarly, the effects of a method are hard to classify. The postconditions are used to specify what should be true after a method has been applied in the desired way, but there is no guarantee that the method will produce these effects. It is also difficult to say which facts will no longer be true.

2.1. What LP needs to Learn

In order to learn a "new technique", LP may need to learn at several levels. At the lowest level, it may need to learn new algebraic identities. These will be used as rewrite rules. We are not particularly concerned with this low level task.

At the next level up, LP needs to learn new operators, i.e. methods. This will involve learning the control information for the method, and associating with it some rewrite rules. These rewrite rules may be new or old rules. The control information does not depend on which equation is being used.

However, LP needs also to learn the meta-control information, which controls the order in which methods are used. This information is recorded in a plan, called a schema. The schema records how the equation is solved, and can be used to solve new equations. The plans may be equation dependent. For example, the plan may record that the solution involved applying method M followed by method N. In the general case however, even if method M can be applied, it may not be the case that method N will then be applicable.

III. LEARNING FROM EXAMPLES

LP learns by examining worked examples. LP is given a worked example that shows how a particular equation is solved. We will call this equation the generating equation.

3.1. Step Justification

The first task for LP is to discover how each line in the worked example is transformed into the next. We call this step justification. To do this, it examines consecutive pairs of lines, trying to find the method that transforms each line to the next.

Suppose that LP is working on the step from a line p to the next line, called q. LP first tries to see if an existing method can account for the step. To do this, LP computes the characteristic tuple, henceforth CT, for each of the lines p and q.

The CT is a tuple of meta-level characteristics of the equation, consisting of facts such as the number of occurrences of the unknown, the type of function symbols occurring in the equations, e.g. trigonometric, whether the equation is a single equation or a

disjunction etc. For example, the CT of

$$\cos(x) + 2.\cos(2.x) + \cos(3.x) = 0$$

is

[occ(multiple,3), trig, eq, functor(+,2)],

meaning that there are multiple occurrences of the unknown, 3 in all. The expression is trigonometric, it is an equation (rather than a disjunction), and the dominating functor on the Lhs is "+" of arity 2.

LP then looks for a method that can transform the CT of p into the CT of q.* Use of the CT to constrain search is an example of the technique of meta-level inference that is used extensively in the PRESS, IMPRESS ([Bundy and Sterling 81]), and LP projects.

If it finds a method, it attempts to use it to transform p into q (a method may not apply even if its CT and other control information indicate that it is suitable). If the method is successful, LP records that the step from p to q was performed by that method, and proceeds to the step from q to the next line. Otherwise, it tries to find another possible method. If no more possible methods can be found, LP conjectures a rewrite rule that would explain the step.

The rewrite rule is obtained by removing common terms** from p and q and equating the remaining terms. Neves, [Neves 78], uses a similar technique. For example, consider the lines (i) and (ii) in the worked example shown in section 1.2. Both lines contain the additive term $2.\cos(2.x)$ and the right hand side of both is 0. Deleting these and equating the remainder produces the conjecture

$$\cos(x) + \cos(3.x) = 2.\cos(2.x).\cos(x).$$

LP asks the user if the conjecture is correct, and if it is, the user is asked to provide the general rule. In this case, the user gives LP the rule

$$\cos(A) + \cos(B) \Rightarrow 2.\cos((A+B)/2).\cos((A-B)/2).$$

*This information is part of the control information of the methods.

**LP uses its knowledge of properties of functions to decide whether terms are common or not. For example, A can be deleted from the equations $A + B = C$ and $D + A = C$, to produce the conjecture $B = D$. However, A can not be deleted from the pair $\log_A B = C$ and $\log_A A = C$, as the two arguments of the logarithmic function do not commute.

If the rewrite rule has the same characteristics as an existing rule set*, the new rule is added to that set, otherwise LP puts it in a special set of miscellaneous rules.

Given that LP is learning from one trial, there seems to be no easy way for the program to correctly generalize rules of this sophistication. Unlike Neves, [Neves 78], it is no longer sufficient just to replace numbers with variables. However, we do not worry about this problem. As mentioned above, we are not particularly interested in the learning of rules, we are more concerned with the learning of methods and schemas.

3.2. Creating New Methods

Once every step has been processed, LP examines its analysis to see if any new methods need to be created. New methods are created to explain the application of new rules. Suppose a new rule R is applied at line i , to produce line $i + 1$. LP first finds the preconditions, P , of the method M applied at line $i + 1$, to give line $i + 2$. It then finds which of the members of P are satisfied at line i . Call these preconditions S . The set difference $P \setminus S$ is the set of preconditions not satisfied at line i , but satisfied at $i + 1$, call this set U .**

If U is non-empty, LP assumes that the purpose of applying R is to satisfy U so that M can be applied. The set U is called the major effects of the rule.

If LP finds that all the preconditions of M were satisfied, i.e. U is empty, it looks for another explanation. One possibility is that the rule is used to manipulate the equation so that M can be applied, although no new preconditions are satisfied. This kind of behaviour occurs because the methods are not STRIPS type operators (see section II).

The above analysis is performed for each application of a new rule in the worked example, causing LP to create a new method. This method applies the rule and then method M . The preconditions of the method are S , plus any preconditions of rule. The postconditions of the method are those of M .

3.3. The Division of the Worked Example

The next step involves dividing the worked examples into sections. This allows LP to

*the rules for each method have certain characteristics, e.g. Collection rules reduce the number of occurrences of the unknown.

** U must be satisfied at $i + 1$ or M could not be applied.

construct new macros and schemas. The first step is to give the worked example a type. At present, LP has three types.

If the worked example contains an application of Change of Unknown (see section II) type is chunk, otherwise, if it contains a Factorization step its type is factor. If neither of these steps occur, the type is normal.

The worked example can now be divided into sections. If the example has type chunk, it is divided into three sections. The first section contains the steps leading up to the Change of Unknown step, the second section contains the steps that solve the changed variable equation, the final section contains the steps involved in the solution of the substitution equation.

If the example has type factor, and the Factorization step produces n factors, the example will contain $n + 1$ sections. The first section contains the steps leading up to the application of Factorization. The remaining sections each contain the steps involved in solving one factor.

If the example has type normal, all the steps go into a single section.

3.3.1. Macro Formation

Each section consists of a sequence of steps. LP now creates a macro operator from the sequence. The process is similar to that for creating new methods, except that the purpose of the sequence can be described differently. Thus macros produced from the first section of a factor type example are marked as being useful for transforming an equation to a state where Factorization can be applied. The preconditions of the macro are the preconditions of the first method in the sequence, and the postconditions are the postconditions of the last method in the sequence, plus preconditions derived from the macros purpose. For example, a macro useful for Factorization would have the extra postcondition that the resulting equation is of the form

$$E_1 \cdot E_2 \cdot \dots \cdot E_n = 0,$$

so that Factorization can be applied.

The new macros are like MACROPS in STRIPS, [Fikes et al 72]. They have a similar status to normal methods, and LP can detect the use of a new method in a worked example. Macros are executed in the obvious way, each method in the sequence is executed in turn. If one of the methods fails, the macro fails.

3.3.2. Schema Formation

Now most of the work has been done, and the schema can be created easily. The schema contains four main parts. These are

- the generating equation,
- the unknown,
- the type of the worked example,
- and the body of the schema.

The body is a list consisting of all the methods used in the worked example. Each step is tagged with the conditions that it is used to satisfy, i.e. the major effects, plus any conditions that must also be maintained. The steps are divided into the divisions that were found above.

The schema is used to solve new equations.

IV. SOLVING NEW EQUATIONS

When LP is given an equation it first tries to find a schema that seems to be relevant. A schema is relevant if the equation that produced it has the same CT as the current equation. If one is found, LP tries to apply the steps listed in the schema. We first discuss the case where the schema is unsectioned, i.e. the worked example was of type normal.

Suppose that the current line in the schema suggests the application of method M. LP uses the following procedure:

1. Try to apply method M. If this succeeds, continue with the next step in the schema.
2. Otherwise, try to find another method that has the same major effects as M, and apply it. If this succeeds, continue with the next line in the schema.
3. If the two steps above both fail, try to find a method that does not undo already satisfied conditions, and apply it. If this succeeds, go to 1.
4. If none of the above steps have succeeded, and if M has no major effects, omit the step entirely, and proceed with the next schema step.

If none of these attempts are successful, LP tries to solve the equation without the schema, in a similar way to PRESS.

If the schema is divided, i.e. the worked example was not of type normal, there is one addition to the above account. Each division in the schema has a purpose. For example, the first section of a factor type schema consists of the steps that transform the equation so that Factorization can be applied. At each step LP checks for fortuitous circumstances. For example, if LP is executing the first section of a factor type schema, it checks to see if the Factorization can be applied to the current equation. If it can, LP assumes that the purpose of the section has been achieved, and ignores the remaining steps in the section. It then proceeds

to the next section.

The schema acts as a simple plan that can be executed in a flexible way.

4.1. Learning by Doing

Neves, [Neves 78], has written a program ALEX that also learns by examining worked examples. He calls this approach learning by example. ALEX can also learn during the equation solving stage, using the learning by doing technique. We have found it preferable to centralize the learning process. New schemas, macros, methods and rules are formed only when LP is analyzing a worked example. However, when LP solves an equation it produces a trace of the solution path. This trace is just a list of the equations produced during the solution process and can be used as a worked example. In this way LP can produce new macros and schemas to help it solve future problems.

4.2. Generating New Problems

Like LEX, [Mitchell et al 81, Mitchell et al 82], LP is able to create new problems for it to solve. After LP has completed its examination of the worked example, it produces a set of equations that differ in only one feature from the generating equation. The simplest change is the replacing of one numerical coefficient by another. Another consists of replacing a function by a sibling function, e.g. replacing $\sin$ by $\cos$. The problem generator allows LP to test the applicability of the schema. At present, LP simply notes if the schema allowed it to solve the test problem or not. A future version could incorporate a concept-learning mechanism that would enable modification of the schema.

V. RESULTS

LP has learned several new techniques from worked examples. One result is that LP has been able to solve difficult equations after examining much easier examples. For instance, after LP has been given worked examples for the equations

$$\sin(x) + \sin(2.x) = 0,$$

$$\sin(4.x) = \cos(2.x), \text{ and}$$

$$\cos(x) + \cos(3.x) = 0 \quad (vi)$$

it is able to solve

$$\sin(2.x) + \sin(3.x) + \sin(5.x) = 0. \quad (vii)$$

In the second equation the coefficient of x in the $\sin$ term* must be twice that in the $\cos$ term,

*The number 4 in our example.

but is otherwise arbitrary. In the first and third equations there are no restrictions on the choice of coefficients. The worked examples for the first three equations contain 7 or 8 lines. The solution for equation (vii) contains 14 major steps, and includes a variant of equation (vi) as a subproblem.

VI. CONCLUSIONS

In this paper we have described LP, a program that learns equation-solving techniques from worked examples. LP uses concepts from planning, analyzing strategic information in terms of satisfying preconditions.

This approach seems to be applicable to other domains. Symbolic integration seems to be an obvious choice. LEX, [Mitchell et al 81, Mitchell et al 82] learns heuristics for symbolic integration. LEX solves integrals by applying a sequence of operators. Each operator transforms the integral into another (hopefully) simpler one. This process is very similar to equation solving. It appears that LP could easily be adapted to learn how to perform symbolic integration, although at present this idea has not been tested. LEX refines heuristics, using several examples, whereas LP learns strategies from one example. Thus the two approaches are somewhat complementary.

Using concepts from planning, the technique of learning from examples seems to work well in the algebra domain, even when the equations are much harder than those considered by earlier workers, [Cotton et al 81, Neves 78].

ACKNOWLEDGEMENTS

This work is supported by S.E.R.C. grant GR/C/20826, and a S.E.R.C. studentship to the author. I would like to thank Alan Bundy, Lawrence Byrd, Jane Hesketh, Richard O'Keefe, Mary Angela Papalaskaris, Dave Plummer, Leon Sterling, Maarten van Someren, and Lincoln Wallen for their help with this work.

REFERENCES

- Brazdil, P. Experimental Learning Model, pages 46-50. Society for the Study of Artificial Intelligence and Simulation of Behaviour, 1978.
- Bundy, A. and Silver, B. A critical survey of rule learning programs, pages 150-157. European Conference on Artificial Intelligence, 1982.
- Bundy, A. and Sterling L.S. Meta-level Inference in Algebra. Research Paper 164, Dept. of Artificial Intelligence, Edinburgh, September 1981.
- Bundy, A. and Welham, B. Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation. Artificial Intelligence, 1981, 16(2), pp 189-212.
- Cotton, J., Byrd, L. and Bundy, A. How can Algebra steps be learned by students with only arithmetic skills. Working Paper 84, Dept. of Artificial Intelligence, Edinburgh, 1981.
- Fikes, R.E., Hart, P.E. and Nilsson, N.J. Learning and executing generalized robot plans. Artificial Intelligence, 1972, 3, 251-288.
- Langley, P. Strategy Acquisition Governed by Experimentation, pages 171-176. European Conference on Artificial Intelligence, 1982.
- Mitchell, T.M., Utgoff, P. E., Nudel, B. and Banerji, R. Learning problem-solving heuristics through practice, pages 127-134. International Joint Conference on Artificial Intelligence, 1981.
- Mitchell, T.M., Utgoff, P. E. and Banerji, R. Learning by Experimentation: Acquiring and modifying problem-solving heuristics. Tioga Press, 1982.
- Neves, D.M. A computer program that learns algebraic procedures by examining examples and working problems in a textbook, pages 191-195. Canadian Society for Computational studies of Intelligence, 1978.
- Sterling, L., Bundy, A., Byrd, L., O'Keefe, R., and Silver, B. Solving Symbolic Equations with PRESS, pages 109-116. Springer Verlag, 1982.
- Winston, P. Learning structural descriptions from examples. McGraw Hill, 1975.
- Young, R.M., Plotkin, G.D. and Linz, R.F. Analysis of an extended concept-learning task, pages 285. International Joint Conference on Artificial Intelligence, 1977.

ADJUSTING BIAS IN CONCEPT LEARNING*

Paul E. Utgoff**

Department of Computer Science
Rutgers University
New Brunswick, New Jersey 08903

ABSTRACT

We approach concept learning as a heuristic search through a space of concepts for a concept that satisfies the learning task at hand. The heuristics represent bias that the concept learning program employs when forming an inductive generalization. We present a model of bias adjustment and report our experience with an implementation of the model.

Key Terms: Inductive concept learning, heuristic, bias, candidate elimination, description language, assimilation, constraint back-propagation.

I. ROLE OF BIAS

A major objective in research on inductive concept learning is creation of a program that can accept training data, apply knowledge, and form inductive hypotheses of the concept, all without human intervention. The learning program searches a space of hypotheses for those that are consistent with the observed examples, and which classify the unobserved instances as indicated by heuristics. The heuristics, which we call *bias*, determine the inductive generalizations that the program will form, given some set of training examples. The initial bias may be appropriate for one learning task, yet inappropriate for another.

We see two basic approaches to using heuristics for inductive concept learning by example. The first approach is to use a relatively large space of hypotheses, and to employ bias to influence how the space is searched. An example of this approach is Michalski's STAR methodology [Michalski 83]. Michalski defines a large hypothesis space via a comprehensive description language. The program uses bias provided by hand to determine whether one consistent hypothesis is to be

preferred to another. Often, the bias is specified as a meta-statement about descriptions, e.g. preference for a simpler description. The second approach is to use a relatively small space of hypotheses, and to employ no bias in searching the space of hypotheses for consistent descriptions. An example of this approach is Mitchell's Candidate Elimination Algorithm [Mitchell 77]. His algorithm uses a restricted space of hypotheses to limit those that the program can consider. The program prunes every hypothesis that has been refuted by a training instance.

Whichever approach is adopted, the common fundamental problem is to obtain and use bias so that the learning program guesses the classification of unobserved instances on the basis of appropriate heuristics rather than simple chance. In most research to date, the search for bias is done by hand. A particular bias is encoded, the program is tested, performance noted, new bias formulated, and the cycle repeated until acceptable performance is achieved.

Automation of the search for appropriate bias is a major problem to be solved. Vere's counterfactual technique [Vere 80] uses a biased hypothesis space, and an unbiased search. When no consistent description can be found, the set difference operator is then employed to construct a new description using those that are already in the space. For example, a new description C can be defined as equal to A-B, where A and B are existing descriptions. The new description C is then added to the space of hypotheses that can be searched by the program.

Lenat's EURISKO [Lenat 83] addresses the same problem. The goal of EURISKO is not only to discover interesting concepts, but also to learn heuristics, or bias, that will lead a program to discover interesting concepts. Unlike Lenat's AM, EURISKO alters the bias that drives the discovery procedure.

II. AN APPROACH TO ADJUSTING BIAS

We represent bias as a restricted search space of concepts that we call the *concept description language*. We use "language" in a formal sense; a set of legal sentences in some given formalism.

* This work was supported by National Science Foundation Grant GMCS80-08889, Rutgers University Laboratory for Computer Science Research, and Siemens Research and Technology Laboratories.

** Current address is Siemens Research and Technology Laboratories, 105 College Road East, Princeton, New Jersey 08540.

As shown in figure 1, we use Mitchell's Candidate Elimination Algorithm [Mitchell 77] to maintain the set of all concept descriptions in the concept description language that are consistent with the training instances. That set is called a *version space*. When the trainer supplies a positive instance, all concept descriptions that exclude the instance are refuted and removed from the version space. Similarly, when the trainer supplies a negative instance, all concept descriptions that include the instance are refuted and removed from the version space.

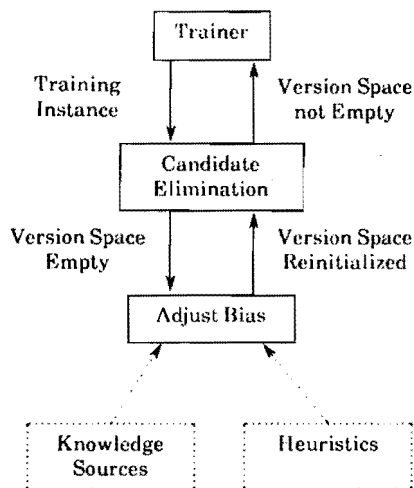


Figure 1 - Learning Element

Figure 2 shows the relationship of the version space, the concept description language, and the formalism for the concept description language. As the version space shrinks, the complementary set of refuted hypotheses grows. If the version space becomes empty, then all descriptions in the concept description language have been refuted. A refuted concept description language indicates a refuted bias, and is sufficient justification for adjusting the bias. We do not yet know stronger justifications. The role of bias adjustment is to enlarge the concept description language so that the version space becomes non-empty and search can continue. It may be that certain concepts are not describable within a particular formalism. If so, shift to a more appropriate formalism may be necessary.

We approach bias adjustment as a three step process:

1. Determine a membership specification for the new concept.
2. Translate the membership specification into a description that uses the formalism of the concept description language.
3. Assimilate the new concept description into the hierarchy of the concept description language.

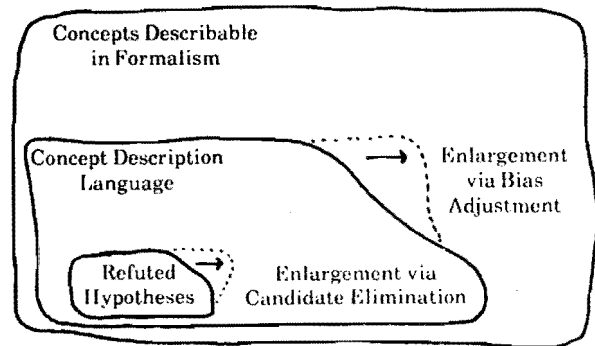


Figure 2 - Model of Bias

A *membership specification* is itself a concept description, but is expressed in an unbiased language so that the heuristics for determining the specification are not compromised by what is expressible. If the membership specification is expressed in an alternate formalism then we require the ability to translate the membership specification into a concept description in the formalism of the concept description language. The translation step may require creation of new descriptive terms that correspond to membership specifications that are not describable in the current concept description language. A new concept must then be assimilated into the concept description language. *Assimilation* is the process of inserting a new concept description into the hierarchy of descriptions in the concept description language. Assimilation is necessary so that we can use the *match* predicate to evaluate whether one concept is more specific or equal to another.

III. AN EXAMPLE

We have implemented a program that models bias adjustment as part of the LEX program [Mitchell 83]. LEX is a concept learning program that learns heuristics that suggest whether a given operator should be applied to a given problem state. For each operator there is a heuristic that is represented as the concept "set of states to which this operator should be applied."

LEX currently learns heuristics in the domain of integral calculus. The program is initially given a set of problem-solving operators. Each operator has a domain of applicability, a rewrite rule, and a range of producible states. For each operator, the legal domain of applicability describes states to which the operator can be applied. For each operator, the heuristic domain of applicability describes states to which the operator should be applied.

LEX solves a problem by producing a state that contains no integral. The solution tree is then criticized. Each operator application along the minimum solution path is labelled as a positive instance showing a state to which the given operator should be applied. Each operator application that leads away from the minimum

solution path is labelled as a negative instance showing a state to which the given operator should not be applied. The training instances are then used to update the heuristics being learned for the operators. For each operator, Mitchell's Candidate Elimination Algorithm is used to maintain a version space of all candidate versions of the heuristic that have not yet been refuted. If a version space becomes empty, then the bias adjustment module is invoked.

3.1. A Heuristic for Membership

Deciding the membership of a new concept is the essence of induction. Which instances should be included, and which should be excluded? We employ a heuristic:

The domain of an operator sequence that leads to a solution should be describable in the concept description language.

We choose this heuristic because of our a priori knowledge that the operator heuristic we hope to learn will be the disjunction of all useful sequences that start with that operator. To describe the disjunction, we can start by being able to describe the disjuncts. To compute a specification of the domain of an operator sequence that produces a state in a given range, in this case the set of solved problems, we apply a deduction procedure that we call *constraint back-propagation*.

Constraint back-propagation is a procedure for deducing the domain of a macrooperator that produces some constrained range of states. Consider an operator sequence of length 1 that uses operator Op_i , as shown in Figure 3. If we constrain the range of the Op_i to some set of states $R \subseteq \text{Range}(Op_i)$, and then apply the inverse operator Op_i^{-1} to R , then we deduce a constrained set $D \subseteq \text{Domain}(Op_i)$ such that $Op_i(D) = R$. For example, an operator $r \rightarrow [r/2]$ has the set of real numbers as its domain and also as its range. If the domain is constrained to the set of integers, then the range is constrained to the set of even integers.

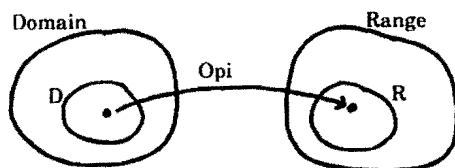


Figure 3 - Constraint Propagation

If we know that Op_i can produce a state that is a solved problem, we can compute $Op_i^{-1}(\text{Intersect}(\text{SolvedProblems}, \text{Range}(Op_i)))$ to deduce a constrained set D . D is then the set of states to which application of Op_i will produce a solved problem. For operator sequences of length greater than 1, the deduction step is simply applied sequentially to the result of the previous deduction.

3.2. A Translation Method

A nested expression of intersections and inverse-operator applications, as shown above, is a specification of concept membership, but is not useful as a concept description because operator applications are required to evaluate whether an instance matches the description. The nested expression must be translated from a membership specification into a concept description that uses the formalism of the concept description language. Our translation method is to evaluate the membership specification. To let this work, we have defined a description of the set of solved problems, an Intersection function, and inverse operator definitions. New concepts are created and assimilated during evaluation of the membership specification.

When an operator only restructures a state via a simple rewrite rule, e.g. $a \cdot b \rightarrow b \cdot a$, creation of new concepts is not required because the range of the operator is expressible by a description in the concept description language. When an operator requires a computation, e.g. multiplication by 2, the range may be describable in the operator language, but not the concept description language. For example, consider an operator $r \rightarrow [r/2]$. The $[r/2]$ indicates that, as part of applying the operator, the bound value of r should be multiplied by 2, and the result used in the rewritten state. If an operator is to be applied to a set of states, as with constraint back-propagation, then correct specification of the operator's domain and range becomes very important. Application of $r \rightarrow [r/2]$ to the set of integers k produces a description $[k/2]$. This is a description in the operator language, but not the concept description language.

Our method of translating a description implied by a computation $y = f(x)$ to a term in the concept description language is to first create a definition $\{y | (\lambda(y) (\text{match } f^{-1}(y) \ x))\}$ and then search for a term with the same definition. If a term with that definition is not found, then a new term is created and defined as per the constructed definition. For example, the definition that corresponds to $[k/2]$ is $\{y | (\lambda(y) (\text{match } y/2 \ 'k))\}$, which we could associate with a new term "even integer."

3.3. An Assimilation Method

Our present method is imprecise. In general, if we are to assimilate $z = \{y | (\lambda(y) (\text{match } f^{-1}(y) \ x))\}$, then we make z a specialization of y . This method is correct only in the sense that no false match relation is asserted. For example, our method may assert that "even integers" is a specialization of "real numbers", but fail to assert that "even integers" is also a specialization of "integers". At the same time, the method does indicate y as a most general bound on where z should be assimilated. If the set of possible assimilations is small, we would expect that a mechanical procedure could prune these by empirical refutation, but we have not pursued this. Below we show an example of deducing the correct assimilation.

3.4. Experiments

We present three experiments with the implementation. Several problems came to light.

3.4.1. Experiment #1

As in [Utgoff 82], LEX found a solution for $\int \cos^7(x) dx$ similar to that shown in Figure 4. LEX was then given $\int \cos^5(x) dx$, for which the same solution sequence, and $\int \cos^6(x) dx$ for which the same solution sequence did not work because the polynomial in the integrand was not an integer. Because the concept description language did not contain a description that included $\int \cos^7(x) dx$ and $\int \cos^5(x) dx$ and excluded $\int \cos^6(x) dx$, the heuristic for Op51 could not be described. Bias adjustment was invoked, and constraint back-propagation was expected to proceed as shown in Figure 4. The implementation was successful for the last three propagation steps through Op-152, Op-150, and Op-151. Back-propagation through Op-113 failed because our formalism, a modified context-free grammar, does not permit specifying that whatever matches f' be the derivative of whatever matches f .^{*} This example shows that the formalism of the concept description language also implies bias. The ability to identify the bias that is associated with a given formalism is an important open problem.

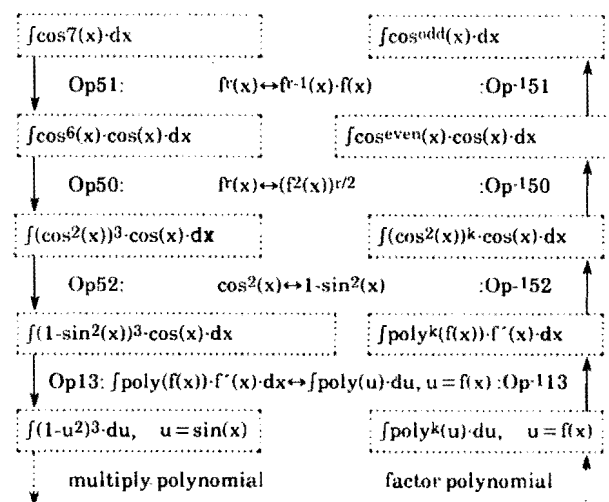


Figure 4 - Solution and propagation for $\int \cos^7(x) dx$

Back-propagation of the set of integers k through the exponent in Op-150 caused a new description $\{x | (\lambda(x)) (\text{match } x/2 \text{ 'k'})\}$ to be constructed; the set of even integers. Because the operator could divide any real exponent by 2, the constructed set description was assimilated as a specialization of the set of real numbers. It would have been better to

^{*} The constraint that f' be the derivative of f is encoded as a test within the operator definition, not within the domain description itself.

assimilate the set of even integers as a specialization of the set of integers. Precise assimilation could have been determined with the following reasoning: An alternate equivalent definition of even integers is $\{y | (\exists x \in \text{Integers}) y=2x\}$. Because x and 2 are integers, and because we know that integers are closed under multiplication, we prove that every y is also an integer. The critical piece of knowledge for the proof is that integers are closed under multiplication. We have not implemented a mechanism for executing such proofs. Proposing and then proving subset relationships is an open problem of great interest. The backward constraint propagation method is useful for deducing which conjectures are worth proving.

3.4.2. Experiment #2

In this experiment, the set of real numbers "r" was to be back-propagated through Op22 as shown in Figure 5. The result of the back-propagation step was $r_4 \cdot (r_3/r_4)$. Because this set was describable by an existing set description, it was not necessary to add any new descriptions to the search space.

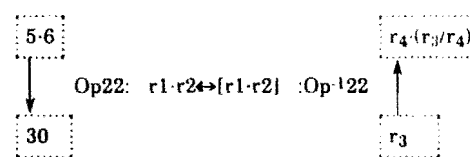
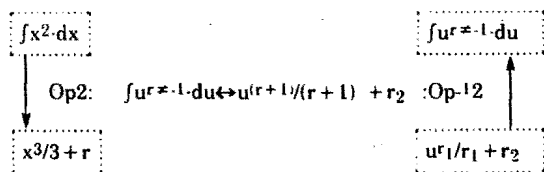


Figure 5 - Evaluating Product with Op22

Nevertheless, the back-propagation failed because the deduced set $r_4 \cdot (r_3/r_4)$ did not satisfy a safety check requiring that a deduced set, in this case $r_4 \cdot (r_3/r_4)$, match the state that occurred in the original solution sequence, in this case 5-6. Although r_4 matched 5, r_3/r_4 did not match 6. The concept description language did not include knowledge that r_3/r_4 is equivalent to the set of rational numbers, and that 6 is an instance of a rational number. The inverse operator had inserted an implicit computation; division.

3.4.3. Experiment #3

A problem $\int x^2 dx$ was given to LEX, and was solved by Op2 as shown in Figure 6. Other instances were also provided that caused the bias adjustment module to be invoked. The program needed to back-propagate the set of real numbers "r" through the exponent in Op-12. The code created a description $\{x | (\lambda(x)) (\text{match } x+1 \text{ 'r'})\}$, and assimilated it as a specialization of r . This produced a description with infinite recursion of the form "x is a real number if x+1 is a real number". If the program could have applied knowledge that the real numbers are closed under addition, in this case by 1, then it would have proven that adding 1 to an x will not alter whether that x is a real number.

Figure 6 - Solution and propagation for $f x^2 dx$

It is important to consider not only where, but whether to assimilate a new description. To circumvent superfluous descriptions, we added an ad hoc procedure for computing closures; if the description D_{new} is intended to be a specialization of an existing description D_{old} , but D_{new} is more general or equal to D_{old} , then D_{new} is superfluous.

IV. SUMMARY

We suggested that heuristics used for induction should be found by the learning program itself, not by hand. We presented a model of a concept learning program that adjusts its own bias, thereby altering the inductive generalizations it will make. We implemented an example of the model by adding a bias adjustment module to the existing LEX concept learning program. Experience with the implementation uncovered examples where the formalism of the concept description language was weak, or additional knowledge was needed to determine whether or where to assimilate a newly constructed description. Where these problems were not a factor, the program was able to automatically adjust its own bias.

ACKNOWLEDGEMENTS

I thank Tom Mitchell, Rich Keller, and Donna Nagel for discussions of ideas presented here. I thank Jack Mostow and Bruce Ladendorf for helpful comments on a draft.

REFERENCES

- Lenat, D. B., "The Role of Heuristics in Learning by Discovery," *Machine Learning*, Michalski, Carbonell, Mitchell (eds.), Tioga, Palo Alto, 1983, pp. 243-306.
- Michalski, R. S., "A Theory and Methodology of Inductive Learning," *Machine Learning*, Michalski, Carbonell, Mitchell (eds.), Tioga, Palo Alto, 1983, pp. 83-134.
- Mitchell, T. M., "Version Spaces: A Candidate Elimination Approach to Rule Learning," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Mass., 1977, pp. 305-310.
- Mitchell, T. M., Utgoff, P. E., and Banerji, R. B. "Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics," *Machine Learning*, Michalski, Carbonell, Mitchell (eds.), Palo Alto, Tioga, 1983, pp. 163-190.
- Utgoff, P. E. and Mitchell, T. M., "Acquisition of Appropriate Bias for Inductive Concept Learning," *Proceedings of the Second National Conference on Artificial Intelligence*, Pittsburgh, August 1982, pp. 414-417.
- Vere, S. A. "Multilevel Counterfactuals for Generalizations of Relational Concepts and Productions," *Artificial Intelligence*, Vol. 14, No. 2, September 1980, pp. 138-164.

Operationalizing Advice: A Problem-Solving Model

Jack Mostow
USC Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA. 90291¹

Abstract

A problem with learning by taking advice arises when the advice is *non-operational*, i.e., expressed in terms of data or actions unavailable to the learner. For example, the advice "avoid taking points" in the card game Hearts is non-operational because the game does not give players an explicit choice about whether to take points. *Operationalization* is the process of converting such advice into an executable (perhaps heuristic) procedure. A model of this process has been implemented in a problem-solver that takes a piece of advice for an agent performing a task and reformulates it in terms of the agent's capabilities by applying transformations from its knowledge base.

The program uses means-ends analysis and planning in an abstraction space to identify non-operational aspects of advice and find transformation sequences that eliminate them. Rather than using a hand-coded difference table, the program analyzes the transformations in order to identify transformation sequences likely to be useful in solving a given problem. Thus new transformations can be added without modifying the problem-solver itself.

Key terms: learning by taking advice, operationalization, program transformation, means-ends analysis, planning in a hierarchy of abstraction spaces

1. Introduction

The idea of learning by taking advice has been around for some time [McCarthy 68], but the ability to accept and follow advice poses several difficult problems. This paper focusses on the problem of handling advice that is *non-operational*, i.e., expressed in terms of data or actions unavailable to the learner.

My dissertation [Mostow 81] [Dietterich et al 82] showed how a transformational approach could capture the process of translating heuristic advice into an effective procedural form. It modelled operationalization as a series of transformations leading from the initial advice statement to an executable procedure. For example, the Hearts advice "avoid taking points in a trick" was transformed into the procedure "play a low card."

¹This research was supported by DARPA Contract MDA-903-81-C-0335. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government. I am grateful to Jeff Barnett, Martin Feather, Matthew Morgenstern, Bill Swartout, and especially Don Cohen for their comments, to the Gist project at ISI for its intellectual influence, and to my dissertation committee (Rick Hayes-Roth, Allen Newell, Jaime Carbonell, and Bob Batzer) for manifold contributions to this research.

This model was incorporated in a program called FOO,² which was used to operationalize a dozen pieces of advice (for Hearts and a music composition task) by applying sequences of transformation rules, some equivalence-preserving, others heuristic. The sequences averaged several dozen steps, but typically each sequence was based on one high-level operationalization strategy; the rest of the steps "jittered" the advice to match the form required to apply the strategy [Fickas 82].

FOO lacked a problem-solver; the user selected transformations by hand and stopped when an operational procedure was produced. This paper describes FOO's successor, a problem-solver called BAR.³ BAR accepts an expression and a pattern; it must rewrite the expression to match the pattern by applying a sequence of transformations from its knowledge base. Automating what was done by hand in FOO raises several issues:

1. *Model of operability:* how can operability be decided automatically?
2. *Goal structure:* given a problem to solve, what subgoals need to be achieved along the way?
3. *Contact:* given a subgoal to achieve, which transformation sequences are likely to help?
4. *Selection:* given several plans for achieving a goal, which is best?

To provide a basis for discussing these issues, Section 2 describes the problem space in which BAR operates; the model of operability is part of this space. Section 3 shows how BAR uses means-ends analysis to identify key subgoals, and Section 4 shows how BAR's abstract plan generator addresses the contact problem. Section 5 presents an extended example to illustrate BAR's strategies for reducing various kinds of differences between an expression and a pattern. Section 6 presents BAR's rather rudimentary techniques for selecting among alternative plans. Section 7 describes some remaining problems. Section 8 evaluates BAR's approaches to the issues, and concludes by putting this work in perspective.

2. The Problem Space

This section describes BAR's problem space [Newell 79] for operationalization, consisting of problem states (initial and transformed advice), goals (patterns), and operators (transformation rules). The problem space represents BAR's knowledge about the task domain, operability, and transformations. BAR's control component is described later.

²For "First Order Operationalizer."

³For "Basically Automatic Reformulator."

2.1. The States

Like FOO, BAR represents advice in a language similar to applicative LISP: there are no assignment statements, the only variables are quantifier variables, and the only functions with side effects are actions that change the state of the environment. A complete definition of this language is outside the scope of this paper: the constructs relevant to this paper include:

- constants
- function applications of the form $(f e_1 \dots e_n)$
- quantification in the form $(Q x S e)$, where Q is a quantifier and e is an expression in which the variable x is quantified over the set S

For example, the initial advice "avoid taking points in a trick" is encoded as $(\text{Avoid} (\text{Take-points ME}) (\text{Trick}))$, where the constant ME denotes the advice recipient and the other symbols are functions.

2.2. The Goals

BAR considers a procedure to be operational for an agent if it is expressed entirely in terms of the data and actions available to the agent. In this model, the agent interacts with its environment (which may contain other, autonomous agents) via a two-way channel: it can sense observable data, and it can act in certain ways to change the state of the environment. The model assumes that the agent can remember any condition or event it could observe in the past, and derive any computable value of observed or remembered data.

This model of operability leads immediately to two types of goals:

1. Find a procedure to evaluate a given expression using available data.
2. Find a procedure to achieve a given condition using available actions.

In the course of solving such problems, BAR generates subgoals of the form

3. Transform an expression to match the left side of a specified rule.

These goals could be represented formally as the goal of proving (constructively) that

$$\exists e' \exists P(e') \wedge R(e, e')$$

where e is the initial expression, P is a predicate that describes the desired syntactic form of the transformed expression e' , and R specifies the desired semantic relation between e and e' . For example, for type (1) e should be semantically equivalent to e' , whereas for type (2) e' should be a procedure for achieving e . At present, BAR pretends that all its transformations are equivalence-preserving; since they are not, it sometimes produces heuristic solutions. All three types of goals are represented in BAR as transforming an expression to match a pattern expressed in a pattern language.

BAR's pattern language includes the following constructs:

- *Literals*: a literal expression matches itself.

- *Variables*: an expression E matches a pattern variable $?v$, binding $?v$ to E , if $?v$ is unbound; otherwise E matches $?v$ only if E matches $?v$'s binding.

- *Sequences*: a sequence $(f e_1 \dots e_n)$ (normally a function application) matches the pattern $(\text{patt}_0 \dots \text{patt}_n)$ if f matches patt_0 and e_i matches patt_i for $1 \leq i \leq n$.

- *Ellipsis*: the segment pattern $"\dots"$ matches zero or more consecutive elements of a sequence.

- *Kleene star*: the segment pattern patt^* matches zero or more consecutive sequence elements all of which match patt .

- *Tests*: E matches the pattern $@P$ if $P(E)$ is true. Notice that P is a predicate on syntactic expressions, not on their referents. The tests in the language include *IsAchievable*, *IsComputable*, *IsPredicate*, *IsConstant*, *IsQuantifier*, etc. Tests act like variables, i.e., matching E to $@P$ binds $@P$ to E , so multiple instances of a test in a pattern must be matched by the same expression. Tests can be bound to different expressions by giving them different suffixes. Thus the expression $(= 3 4)$ matches the pattern $(= @\text{IsConstant1} @\text{IsConstant2})$ but not the pattern $(= @\text{IsConstant} @\text{IsConstant})$. BAR automatically generates such suffixes to distinguish among different instances of a test inside a Kleene star: this allows $(= 3 4)$ to match $(@\text{IsComputable} @\text{IsConstant}^*)$.

- *Containment*: an expression matches the pattern $[\dots \text{patt} \dots]$ if it matches patt or contains a subexpression that matches patt . This is a weak version of what it means to be expressed in terms of something; for example, the expression $(f x y)$ matches $[\dots y \dots]$ even if f ignores its second argument. However, BAR's current set of transformations and the way it uses them have so far prevented this deficiency from causing any vacuous solutions.

- *Independence*: an expression matches the pattern $[\sim \dots v \dots]$ if it contains no references to the variable v .

- *Conjunction*: an expression matches the pattern $[\text{patt}_1 \dots \text{patt}_2 \dots]$ if it matches both patt_1 and $[\dots \text{patt}_2 \dots]$, and similarly for $[\text{patt} \sim \dots v \dots]$. More general conjunctive patterns have not been needed yet.

2.3. Model of Operability

BAR's knowledge about operability includes a description of the advice recipient's capabilities in a given environment. For example, BAR knows that the predicate *Has(player, card)* is evaluable when $\text{player} = \text{ME}$, but not in general, since player ME (the learner) can see its own cards but not its opponents'. This is implemented by defining *Has-ME(card) = Has(ME, card)* and attaching the property *IsComputable* to the function *Has-ME* but not to *Has*. Similarly, the fact that player ME chooses its own card but not its opponents' cards is represented by attaching the property *IsControllable* to *Card-played-by-ME*, defined as *Card-played-by(ME)*, but not to *Card-played-by*.

General knowledge about what is operational is encoded in BAR as a set of facts defining the class of operational procedures. For example, the fact that the application of a computable function to evaluable arguments is evaluable is represented as

(@IsComputable @IsEvaluable*)
—matches→ @IsEvaluable (1)

Similarly, the fact that an evaluable constraint on a controllable variable is considered operational is represented as

(@IsConstraint (@IsEvaluable...@IsControllable...))
—matches→ @IsAchievable (2)

Rule 2 embodies the heuristic assumption that the value of a controllable variable can be chosen to satisfy a given constraint. For example, it says that the constraint (Require (Low (Card-played-by:ME))) is achievable, but this is false if player ME has no low cards.

2.4. The Operators

BAR's operators are transformations that rewrite expressions in this language. A transformation rule $lhs \rightarrow rhs$ can be used to rewrite the advice or to replace a sub-expression of it.

Some of the transformations encode knowledge about domain concepts, which BAR represents as functions. For example, the concept "avoid" is represented by the function

Avoid(event,scenario) =
(Prohibit (During scenario event)) (3)

That is, "don't let event occur during scenario." Each definition corresponds to two possible transformations: *unfolding* an instance $(f e_1 \dots e_n)$ of a function $f(x_1 \dots x_n)$ by substituting $e_1 \dots e_n$ for $x_1 \dots x_n$ throughout f 's definition, and *folding* an expression that matches this definition into an instance of f [Darlington 76].

Other transformations can represent knowledge about a whole class of functions. For example, BAR lacks a definition for the relation During but knows that it is reflexive, and has rules for transforming reflexive relations. One such rule is

(R (f $x_1 \dots x_n$) (f $y_1 \dots y_n$)) → (And (= $x_1 y_1$) ... (= $x_n y_n$)) (4)

This transformation rule rewrites a reflexive relation R between two expressions $(f x_1 \dots x_n)$ and $(f y_1 \dots y_n)$ by equating their corresponding arguments, which is a useful simplification in the absence of a definition for R. Rule 4 is shown here in an informal notation for readability; rules are actually implemented as procedures using the Interlisp Pattern Match Compiler [Teitelman 78]. Rule 4's right side logically implies its left side, but the converse need not be true. Thus depending on where it is used, Rule 4 can make heuristic (non-equivalence-preserving) transformations.

3. Overview of Problem-Solver

BAR uses means-ends analysis [Newell 69] to find a sequence of transformations that rewrites a given expression to match a given pattern.

BAR's top-level procedure, called Transform, takes an expression and a pattern as input. For example, the problem of operationalizing "avoid taking points" is given to BAR by calling Transform with (Avoid (Take-points ME) (Trick)) as the expression and @IsAchievable as the pattern.

Transform calls Reduce to reduce differences between the expression and the pattern. Reduce identifies the grossest difference between them, looks for plans to eliminate it, selects a plan, and tries to apply it. If this generates subproblems (rewriting the expression to match the left side of a proposed rule), Reduce recursively invokes Transform to solve them. After each transformation, BAR tries to simplify the new expression. Simplifying the expression typically renders the current plan obsolete and causes BAR to replan. Transform calls Reduce repeatedly until either the rewritten expression matches the pattern (success) or Reduce is unable to make further reductions (failure).

Reduce knows about several kinds of differences, listed below in the order it tries to reduce them:

1. Semantic difference between expression (f ...) and pattern (P ...): f doesn't match P
2. Extraction: f doesn't match P, but (f ...) contains a sub-expression that might
3. Recursion: sub-expression doesn't match corresponding subpattern
4. Deletion: sub-expression must be deleted to match pattern [...?v...]
5. Expansion: expression must mention a specified function to match pattern [...f...]
6. Restructuring: arguments must be rearranged to match pattern

A plan is simply a series of proposed transformations to apply: it is abstract in that getting the expression into the form required by a step in the plan may require applying additional transformations along the way. The next section shows how BAR finds abstract plans for reducing semantic differences. The treatment of other kinds of differences is illustrated in the Section 5, and BAR's mechanisms for selecting among alternative plans are described in Section 6.

4. Planning with Operator Abstractions

To find plans for reducing semantic differences, BAR solves an abstracted version of the problem. The purpose of the abstract search is to find a high-level plan for rewriting an expression (f ...) to match a pattern (P ...), that is, to find a sequence of rules that reformulates an instance of f as an instance of some g that matches P. This is accomplished by abstracting the original problem space essentially as follows:

- Abstract a state (f ...) as f
- Abstract a goal (P ...) as P
- Abstract a rule (L ...) → (R ...) as L → R

BAR then searches for a sequence of abstracted rules leading from f to P; this sequence constitutes an abstract plan for rewriting (f ...) to match (P ...), i.e., carrying out the plan generally requires additional transformations to fully match the left side of each step.

Somewhat more precisely, BAR searches for a path from f to P (without repeated symbols) consisting of the following kinds of steps:

- $f \text{---unfold-->} g$, where f 's definition has the form $(g \dots)$
- $f \text{---fold-->} g$, where g 's definition has the form $(f \dots)$
- $L \text{---rule-->} R$, where rule has the form $(L \dots) \rightarrow (R \dots)$
- $f \text{---matches-->} @P$, where f satisfies the test P^4
- $f \text{---fact-->} @P$, where fact has the form $(f \dots)$
- $(f \dots) \text{---matches-->} @P$

To find such plans, BAR uses straightforward depth-first search up to a preset depth limit, currently 10. The recursive search procedure `FindRephrasePlan(expression, pattern, depth)` can be described as follows:

```

If expression matches pattern
then return the one-step plan
    expression ---matches--> pattern
else if depth = 0 then fail
else if there is an allowable step
    expression---next such that
        FindRephrasePlan(next, pattern, depth-1) returns
        a plan
    then return CONS(step, plan)
else fail.
```

To prune the search, `FindRephrasePlan` uses memo functions [Marsh 70] in several ways. First, it remembers successful calls; if it has earlier found a path from g to P , and encounters g while searching for a path from f to P , it will use it to form a path to P . Second, it remembers unsuccessful calls and their associated depth limit. If `FindRephrasePlan(g.P,n)` fails, subsequent calls of the form `FindRephrasePlan(g.P,k)` with $k \leq n$ will fail immediately. BAR further prunes the search by testing whether a proposed step is allowable; it rejects any step that repeats an earlier state in the current path, or has proved impossible to carry out in the past.

BAR does not stop after finding one plan to reduce a given difference; it tries to cover the set of possible plans by finding all first steps that can be extended into a plan. BAR does not bother to generate more than one plan with the same first step, since there is no point in finding all such plans if the first step fails. If BAR gets stuck partway through a plan, it tries replanning from the current expression. If this fails, BAR backs up to the original expression and tries a different first step. This arrangement reduces the search for plans at the cost of replanning whenever the chosen plan strays from the correct path after the first step. To generate only one path with a given first step, `FindRephrasePlan` rejects the first step if a plan that starts with it has already been found for reducing the current difference.

In the case at hand, `FindRephrasePlan` finds one plan for eliminating the semantic difference between `(Avoid (Take-points ME) (Trick))` and the pattern `@IsAchievable`:

```

Avoid ---unfold-->
Prohibit ---matches-->
@IsConstraint ---Fact 2-->
@IsAchievable
```

(5)

⁴To make this abstraction work, BAR's tests are usually defined so that if the expression $(f \dots)$ satisfies a test, e.g., `IsPredicate`, so does the function f .

Actually, this description is over-simplified -- a plan can contain more information than a series of atomic symbols connected by rules. `FindRephrasePlan` tries to preserve as much information as possible about the expression resulting from each step. If a rule in a plan can be applied without subgoaling, `FindRephrasePlan` does so. If not, it tries to force a match between the expression and the left side of the rule, and then substitute the resulting bindings in the right side of the rule to produce a partial instantiation. These bindings represent subproblems to be solved. Only as a last resort does BAR abstract an expression as its topmost symbol.

Although BAR applies its transformation rules procedurally, a procedural rule representation makes it difficult to identify the subgoals required to match the left side of a rule in a plan, or to predict what the eventual result of applying it will look like. For this kind of reasoning, BAR uses rule knowledge expressed in the pattern language. For example, Rule 4 is described as

$(@IsReflexive (?f \dots) (?f \dots)) \rightarrow (And (= ?x ?y))$

This is an abstraction; it lacks information contained in the procedural representation, but captures several facts useful in deciding when to try Rule 4:

- Rule 4 rewrites a reflexive predicate as a conjunction.
- The two arguments of the predicate must be instances of the same function.
- Rule 4 eliminates the reflexive predicate and the function.

Thus plan 5 actually looks like this:

```

(Avoid (Take-points ME) (Trick))
---unfold-->
(Prohibit (During (Trick) (Take-points ME)))
---matches-->
(@IsConstraint (@IsEvaluable...@IsControllable...))
---Fact 2-->
@IsAchievable
```

(6)

Executing this plan will require solving the subproblem of rewriting `(During (Trick) (Take-points ME))` to match `[@IsEvaluable...@IsControllable...]`. This subproblem takes several dozen steps to solve, and illustrates the benefit of planning in the abstracted problem space [Sacerdoti 74]: it prunes the search by trying only those transformations that lead to a solution in the abstraction space. Otherwise BAR could waste lots of time solving such subproblems along the way to a dead end.

5. Reduction Strategies Illustrated

BAR's strategies for reducing each kind of difference are now illustrated by means of an extended example: the operationalization of "avoid taking points." This takes about 60 transformation steps, but the reasoning can be loosely paraphrased as follows:

1. I want to avoid taking points in the trick.
2. To take points, I would have to take a trick with points.
3. To take the trick, I would have to play the highest card in the suit led.

4. So I should play a card lower than somebody else's.
5. But I can't always tell whether my card will be lower than somebody else's.
6. So I should simply play a low card, and hope for the best.

The problem is given to BAR by calling Transform with (Avoid (Take-points ME) (Trick)) as the expression and @IsAchievable as the pattern to match. Reduce identifies the semantic difference between Avoid and @IsAchievable as the grossest difference between the two, and FindRephrasePlan finds plan 6 above. BAR looks for all possible plans before choosing which one to apply, but in this case it can only find one, so it proceeds to execute it.

The first step of the plan unfolds the definition (3) of Avoid, rewriting the initial advice as

(Prohibit (During (Trick) (Take-points ME))) (7)

Although Prohibit satisfies the test IsConstraint, this expression does not match the left side of Fact 2, because

(During (Trick) (Take-points ME)) (8)

is not an evaluable constraint on a variable controlled by ME. When the current expression does not match the next step in a plan, Reduce invokes Transform to rewrite the expression to match the left side of the rule. This idea is not new -- GPS invoked itself to satisfy the preconditions of an operator it wanted to apply [Newell 60]. Thus Reduce now calls Transform to rewrite expression 7 to match (@IsConstraint [@IsEvaluable ...@IsControllable...]), and Transform calls Reduce to reduce the differences between them.

Given an expression ($f(e_1 \dots e_n)$) and a pattern ($P \text{ patt}_1 \dots \text{patt}_n$) where f matches P , Reduce uses recursion to reduce the differences between them by Transforming each e_i to match the corresponding subpattern patt_i . Thus Reduce now calls Transform with expression 8 and the pattern [@IsEvaluable...@IsControllable...].

By unfolding the definitions of Trick and Take-points and applying Rule 4, BAR eventually reduces expression 8 to

(And (Trick-has-points)
(= (Card-played-by ME) (Winning-card))) (9)

That is, for player ME to take points, the trick must have points and player ME must play the highest card in the suit led.

At this point, BAR is partway through a plan to rewrite the expression as an instance of the computable function And, and then apply Fact 1; to match the left side of the rule, each conjunct must match @IsEvaluable. To simplify this example, Trick-has-points is marked IsComputable to represent the existence of (heuristic) methods for predicting whether a trick will have points. (BAR has not been used to derive such methods but this could be done.) However,

(= (Card-played-by ME) (Winning-card)) (10)

is unevaluable because Winning-card cannot be predicted. To make this expression evaluable, BAR decides to apply

(= (The x S Px) c) $\rightarrow$ (And (In c S) Pc) (11)

It turns out that (Card-played-by ME) cannot be transformed to match (The x S Px) given BAR's current set of concept definitions.

When Reduce cannot transform the arguments of an expression to match the arguments of a pattern, it looks for rules that might restructure the expression by moving information between arguments. It then tries to transform the restructured expression to match the pattern. A rule is considered a restructuring rule if it has the form $P \dots \rightarrow P \dots$, i.e., leaves the function of an expression unchanged. One such rule transposes the arguments of a symmetric function:

(@IsSymmetric ?x ?y) $\rightarrow$ (@IsSymmetric ?y ?x) (12)

Applying Rule 12 to expression 10 produces

(= (Winning-card) (Card-played-by ME)) (13)

To apply Rule 11, BAR unfolds (Winning-card), producing

(= (The ?c (Cards-played-in-suit-led)
(Forall ?x (Cards-played-in-suit-led)
(Not (Higher ?x ?c))))
(Card-played-by ME)) (14)

The result of applying Rule 11 reduces to

(And (In-suit-led (Card-played-by-ME))
(Forall ?x (Cards-played-in-suit-led)
(Not (Higher ?x (Card-played-by-ME))))) (15)

That is, in order for ME to take the trick, no other player can play a higher card in the suit led.

BAR considers the first conjunct of expression 15 to be evaluable, but the second conjunct is quantified over (Cards-played-in-suit-led), which player ME cannot generally predict. When an expression cannot be transformed to match a pattern but contains an argument that can, BAR looks for rules that might extract the argument. A rule is considered if its right side is embedded in its left side, like

(@IsQuantifierPredicate ?x ?S [@IsPredicate -...?x...])
 $\rightarrow$ @IsPredicate (16)

Rule 16 extracts a predicate from the scope of a quantifier provided the predicate is independent of the quantifier variable. It suggests a plan for evaluating expression 15: eliminate the quantifier variable ?x from the predicate

(Not (Higher ?x (Card-played-by-ME))) (17)

Then extract the predicate and make it evaluable.

To delete a variable from an expression, BAR looks for rules that have a pattern variable in an argument position on the left side but not on the right side. One such rule is abstracted as

(@IsComparative ?e ?v) $\rightarrow$ (@IsPredicate ?e) (18)

Rule 18 approximates a binary ordering relation as the corresponding unary predicate, thereby deleting ?v. BAR decides to use Rule 18 to delete ?x from

(Higher ?x (Card-played-by-ME)) (19)

However, ?x must be the second argument of the binary relation for this plan to work, so BAR first rewrites expression 19 as (Lower (Card-played-by-ME) ?x) and then applies Rule 18, reducing expression 19 to (Low (Card-played-by-ME)). That is, to make my card lower than somebody else's, I should play a low card and hope for the best.

The final result of Transforming expression 8 to match @IsEvaluable is the expression

(And (Trick-has-points)
(In-suit-led (Card-played-by-ME))
(Not (Low (Card-played-by-ME)))) (20)

Since Card-played-by-ME is a choice variable controlled by player ME, expression 20 also happens to match the pattern [...@IsControllable...] in Fact 2. Otherwise, Reduce would try to expand expression 20 in terms of some controllable function. To transform an expression to match [...P...], BAR uses a search that resembles FindRephrasePlan but allows steps that go from an expression to one of its subexpressions.

At this point the overall expression is

```
(Prohibit (And (Trick-has-points)
  (In-suit-led (Card-played-by-ME))
  (Not (Low (Card-played-by-ME))))) (21)
```

This expression matches the left side of Fact 2.

```
(@IsConstraint (@IsEvaluable...@IsControllable...))
--matches-- @IsAchievable
```

@IsAchievable was the original goal, so BAR halts. Thus the result of operationalizing "avoid taking points" can be freely paraphrased as "play a low card if the trick has points."

6. Selection

Since BAR deals with problems requiring dozens of steps to solve, and often finds several plans for reducing a given difference, it faces a combinatorial search problem that will worsen as more rules and concepts are added. It uses several techniques to deal with this problem.

BAR rates the plans it finds according to their estimated chance of success, and tries them in order until it finds one that succeeds. The rating depends on the number of steps in the plan that require additional problem-solving and a numerical strength for each rule. Heuristic rules like Rule 18 are penalized so as to try correctness-preserving plans first.

Alternatively, BAR provides a mode called UserSelectPlan that allows the user to guide the search. In this mode, whenever Reduce finds multiple plans it asks the user to select among them, but whenever it finds only one plan it tries it without asking. This reduces the question-answering burden on the user but leads to extra search in cases where a single plan is found and fails.

BAR caches the results of all calls to Transform and Reduce in the course of solving a given problem [Lenat 79] [Marsh 70]. This allows a subproblem solution found in one branch of the search to be used in another, and prevents a failed subproblem from being re-attempted. Caching is also used to avoid the kind of looping that occurs when a problem generates itself as its own subproblem. When a problem is first attempted, a failure result is cached; if in the course of solving the problem, a subproblem identical to it is encountered, the subproblem will fail without preventing the solution of the original problem by other means.

Finally, BAR keeps track of the number of successful and unsuccessful attempts to apply each rule. This information is used both to order the path-finding search and to rate plans once they are found, allowing BAR to tune its feasibility estimates even during the course of a single problem. This simple form of learning generalizes experience gained in the course of problem-solving, and allows BAR to tune its own performance within and between problems. It is compatible with user guidance; in UserSelectPlan mode, the plan ratings have no effect, but tuning increases the chances that the first plan found with a given initial

step will be the most feasible plan starting with that step. This cuts down on the replanning required when the adopted plan gets stuck after the first step. Of course such statistical learning is quite weak; learning to discriminate between successful and unsuccessful attempts to apply a rule would be much stronger. However, predicting whether a rule can be applied to a given expression requires rather deep analysis, and it may be less work to try it and see.

As of this writing the "avoid taking points" example has been operationalized only in UserSelectPlan mode. Even though the right plan is frequently the only one found or is scored higher than the alternatives, and wrong paths often reach dead ends quickly, BAR's rating scheme is not strong enough to prevent an explosive search, and is fragile in the face of added rules.

7. Remaining Problems

BAR can be evaluated by how well it decides operability, models the goal structure of operationalization, makes contact between problems and methods, and selects among competing methods.

As implemented, BAR's model of operability ignores preconditions and state dependencies -- it does not handle procedures that can be executed only at certain times or in certain situations. For example, the procedure to avoid taking points implicitly assumes that player ME can legally play a low card. Also, the circumstances under which Trick-has-points is computable are not explicitly represented. The model assumes that the environment is external to the agent operating in it: it does not address the problem of representing and reasoning about states of knowledge. Although BAR uses heuristic methods like Rule 18 that lead to fallible procedures, it does not consider their relative reliability. To analyze the logical status of a transformation, BAR would have to keep track of the type of goal being pursued and the context of the sub-expression to be transformed. Also, the model assumes that all computation is free.

A more realistic model would acknowledge that operability is relative: some methods are more reliable than others, some computations are more expensive than others, and the operability of a procedure depends on the feasibility of executing it and the probability that doing so will produce the desired result. These improvements are needed to handle operationalization goals of the form "make this procedure cheaper to execute," "make this estimate more accurate," or "make this plan succeed more often." Some such problems were solved in FOO, for example the problem of refining a heuristic search procedure [Mostow 82], but the goals were implicit in the hand-selected sequence of transformations.

BAR uses a rather syntactic goal structure: given an expression and a pattern, its top-level goal is to change the top-most symbol of the expression to match the top-most symbol of the pattern. It addresses the contact problem by searching for abstract plans to get from one to the other. This planning narrows the set of attempted transformations to those for which some abstract path to the solution exists. However, this scheme generates vacuous plans when the top-most symbol contains little semantic content, e.g., "And" or "Exists." Moreover, this goal structure appears unintuitive when the important reasoning is buried in one of the subproblems. The planner might be improved by explicitly identifying the obstacles to the evaluation of an expression. For example, the fact that BAR lacks a definition for During would motivate its deletion from expression 8.

To constrain the search, BAR assumes that an expression ($f \dots$) can be transformed to match a pattern ($g \dots$) without rewriting the expression in the form ($g \dots$) along the way, if it can be done at all. It is easy to construct examples that violate this assumption. For example, to transform $(\text{Union } A \ B)$ into $(\text{Union } B \ A)$ if one didn't know Union was commutative, one might have to unfold $(\text{Union } A \ B)$ as $(\text{SetOf } x \ \text{Universe } (\text{Or } (\text{In } x \ A) \ (\text{In } x \ B)))$, transpose the arguments of Or, and refold into $(\text{Union } B \ A)$. BAR cannot find proofs of this simple "bridge" form so familiar to algebra students, although none have been needed in the few examples attempted thus far. Allowing BAR to consider such plans would explode its search space unless the conditions for considering them were strongly restricted.

BAR's selection mechanism is rather weak, since it relies on crude numerical estimates of rule feasibility. Tuning these estimates based on experience provides limited improvement at best. Also, BAR searches depth-first, although it tries plans in order of estimated feasibility. A breadth-first strategy could find short transformation sequences faster, but would require more memory. Also, implementing it would be complicated by the recursive nature of the problem-solver. A more promising approach is to constrain the output of the plan generator based on the identified non-operational aspects of the current expression.

8. Conclusion

BAR is still in a relatively early stage of development; so far it contains only 60 of FOO's 230 rules and 38 of FOO's 112 concept definitions, and FOO itself came nowhere close to covering the space of operationalization problems. In the absence of additional knowledge or improved control strategies, BAR's syntactic approach can be expected to doom it to combinatorial explosion in the face of added rules and concepts. Thus BAR should not be viewed as a practical tool for operationalization, but as a vehicle for investigating the knowledge, inference methods, and goal structure needed by such a tool. Already it has inspired the several improvements proposed in the previous section.

BAR addresses only one of many problems involved in taking advice. Other key problems include interpreting informally expressed advice, acquiring domain concepts, developing a catalog of transformation rules, and integrating multiple pieces of advice [Mostow 79b]. Practical application of the work described here will be easiest where these other problems are minimized, i.e., where the advice, domain knowledge, and a small but adequate set of transformations have already been formalized.

A promising candidate for such an application arises in the course of transforming a formal closed system specification [Feather 83] into a compilable program. Such a specification defines acceptable behavior for a system of interacting components, and describes the data and actions available to each component; the problem is to implement each component subject to these restrictions. Many of the required transformations have already been formalized [London & Feather 82]. An especially attractive aspect of this application is that a useful solution need not solve the control problem completely; automating even some of the implementation decisions could significantly amplify human software productivity [Fickas 82].

References

- [Darlington 76] J. Darlington and R. M. Burstall, "A system which automatically improves programs," *Acta Informatica* 6, 1976, 41-60.
- [Dietterich et al 82] T. G. Dietterich, Bob London, K. Clarkson, and G. Dromey, "Mostow's Operationalizer," in P. R. Cohen and E. A. Feigenbaum, Volume 3 (eds.), *Handbook of Artificial Intelligence*, Stanford Computer Science Department, Stanford, CA, 1982. In section on Learning and Inductive Inference, available as STAN-CS-82-913/HPP-82-10.
- [Feather 83] M. S. Feather, Closed System Specifications, 1983. In preparation.
- [Fickas 82] S. Fickas, *Automating the Transformational Development of Software*, Ph.D. thesis, University of California at Irvine, 1982.
- [Lenat 79] D. B. Lenat, F. Hayes-Roth, and P. Klahr, "Cognitive economy in artificial intelligence systems," in *IJCAI-6*, pp. 531-536, Tokyo, Japan, 1979.
- [London & Feather 82] London, P.E. & Feather, M.S., "Implementing specification freedoms," *Science of Computer Programming*, (2), 1982, 91-131.
- [Marsh 70] D. Marsh, "Memo functions, the Graph Traverser, and a simple control situation," in B. Meltzer and D. Michie (eds.), *Machine Intelligence* 5, pp. 281-300, American Elsevier, New York, 1970.
- [McCarthy 68] J. McCarthy, "The advice taker," in M. Minsky (ed.), *Semantic Information Processing*, pp. 403-410, MIT Press, Cambridge, MA, 1968.
- [Mostow 79a] D. J. Mostow and F. Hayes-Roth, "Operationalizing heuristics: some AI methods for assisting AI programming," in *IJCAI-5*, pp. 601-609, Tokyo, Japan, 1979.
- [Mostow 79b] D. J. Mostow and F. Hayes-Roth, *Machine-aided heuristic programming: A paradigm for knowledge engineering*, The Rand Corporation, Santa Monica, CA, Technical Report Rand-N-1007-NSF, 1979.
- [Mostow 81] D. J. Mostow, *Mechanical Transformation of Task Heuristics into Operational Procedures*, Ph.D. thesis, Carnegie-Mellon University, 1981. Technical Report CMU-CS-81-113.
- [Mostow 82] D. J. Mostow, "Learning by being told: Machine transformation of advice into a heuristic search procedure," in J. G. Carbonell, R. S. Michalski, and T. M. Mitchell (eds.), *Machine Learning*, Palo Alto, CA: Tioga Publishing Company, 1982.
- [Newell 60] A. Newell, J. Shaw, H. Simon, "Report on a general problem-solving program for a computer," in *Proceedings of the International Conference on Information Processing*, pp. 256-264, UNESCO, Paris, 1960.
- [Newell 69] A. Newell, "Heuristic programming: Ill-structured problems," in J. Aronofsky (ed.), *Progress in Operations Research*, pp. 363-414, Wiley, New York, 1969.
- [Newell 79] A. Newell, *Reasoning, problem solving and decision processes: the problem space as a fundamental category*, Carnegie-Mellon University Computer Science Department, Pittsburgh, PA, Technical Report, June 1979.
- [Sacerdoti 74] E. D. Sacerdoti, "Planning in a hierarchy of abstraction spaces," *Artificial Intelligence* 5, 1974, 115-135.
- [Teitelman 78] Teitelman, W., *Interlisp Reference Manual*, Xerox Palo Alto Research Center, 1978.

GOAL DIRECTED LEARNING*

Tom M. Mitchell and Richard M. Keller

Computer Science Department
Rutgers University
New Brunswick, NJ 08903

1 Introduction

Much research on learning has focussed on methods for empirical generalization: for learning descriptions of general concepts from training examples. This generalization problem is usually formulated so that the only information available to the learner is the set of input training instances, and some a priori biases for preferring certain generalizations over others. We argue here that this kind of formulation of the generalization problem is unrealistic and overly difficult--that when humans face such generalization problems, they typically have some goal, or purpose for attempting to learn, a particular concept. Our main point is that explicit knowledge of this learning goal, and of the context in which it was formulated, can provide very strong guidance for learning.

The longer version of this paper will compare two recent learning systems that use explicit knowledge of their learning goal as the basis for driving learning. One of these is the LEX2 system [Mitchell 82, Keller 83, Mitchell 83], which learns problem-solving heuristics in integral calculus. The learning goal of LEX2 is to learn "heuristics that recommend problem solving steps which lead to solutions". LEX2 interprets each of its positive training instances in terms of this learning goal, in order to determine which features of the instance are essential, and which are irrelevant to the heuristic to be learned. In LEX2, we have found that introducing explicit knowledge of the learning goal leads to two important new capabilities: (1) the ability to form a justifiable generalization by analyzing a single training instance [Mitchell 82, Keller 83, Mitchell 83], and (2) the ability to discover new terms, such as "odd integer", which are essential for describing heuristics in the program's task domain of integral calculus [Utgoth 82].

In comparison to LEX2, its predecessor, LEX [Mitchell 81], had no explicit knowledge of its learning goal. In giving LEX2 knowledge of its learning goal, we have changed in a fundamental way the nature of the generalization problem. In particular, we have told the system the concept (e.g., "problem solving steps that lead to solutions") that it is to learn! Without explicit knowledge of its learning goal, LEX was forced to make inductive leaps to guess the classification of unobserved training instances. With knowledge of this learning goal, LEX2 begins by knowing *in principle* the classification of every possible training instance. But while the learning goal

constitutes a *valid* definition of the concept to be learned, it is not a very *useful* definition. In particular, to determine whether any particular problem solving step is a "problem solving step which leads to a solution", one must first attempt to find a solution! Since learned heuristics are meant to guide the problem solver in finding solutions, this method for evaluating whether a potential search step should be attempted is prohibitively expensive. Thus, the learning problem in LEX2 can be viewed as a problem of reformulating a valid definition of the concept to be learned, into a form in which it is more useful for recognizing instances of the concept.

A second system which uses explicit knowledge of its learning goal is the BAR system [Mostow 83] (a successor to the FOO system [Mostow 82]) which learns effective strategies for playing the card game Hearts. BAR starts with knowledge of its learning goal (for example to learn "which cards to play to avoid taking points"). As with LEX2, this initial piece of advice is not particularly useful, since it gives no information as to how the recommendation should be implemented. Similarly, a piece of advice which requires complete knowledge of another player's hand (e.g. "lead with a card having higher point value than any card of the same suit in another player's hand") is not directly useful since this kind of information is not typically available to the card player. The system must reexpress such advice into a form which the player can use more directly. For example, there are many possible ways in which the advice to "avoid taking points" could be successfully realized by a Hearts-player. Consider the following operational forms:

- Always play your lowest card in the suit led.
- Underplay another card in the current hand.
- Use probabilistic methods to determine the best card to play.
- Search through a space of possible scenarios in which one might envision taking points. Avoid any such scenarios.

Mostow's program can derive each of these forms from the initial, non-usable advice about avoiding tricks with points. In contrast to LEX2, BAR reasons directly from its learning goal to the more operational form of its knowledge, without considering training examples to guide this process.

*This is an extended abstract of a forthcoming paper to be presented at the Second International Machine Learning Workshop. This work is supported by the National Science Foundation under grant MCS80-08889, and by the National Institutes of Health under grant RR-64309.

Goal-directed learning is a form of learning by being told: the learning goal defines the concept to be learned, and the learning problem is one of reexpressing this concept in a more useful form. While the learner's goal could come from outside the learning system, we argue that for many kinds of learning, the learning goal could instead be formulated by the system itself. Given the capability to monitor its own performance, and to formulate learning tasks, a problem-solving system should be able to compose learning goals such as "learn heuristics for task X, so that the heuristics suggest problem-solving steps leading to solutions." In such cases, goal-directed learning corresponds to learning by being told *by oneself*. The Eurisko program [Lenat 82] provides one example of a system that monitors aspects of its own performance in order to formulate certain types of useful learning tasks. We suggest that such learning tasks are naturally suited to goal-directed learning.

[Keller 83]

[Lenat 82]

[Mitchell 81]

[Mitchell 82]

[Mitchell 83]

[Mostow 82]

[Mostow 83]

[Utgoff 82]

References

- Keller, R.
Learning by Re-Expressing Concepts for Efficient Recognition.
In *Third National Conference on Artificial Intelligence*, to appear. Washington, D.C., August, 1983.
- Lenat, D. B.
The role of heuristics in learning by discovery: three case studies.
In Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (editors), *Machine Learning*, Tioga Press, Palo Alto, C.A., 1982.
- Mitchell, T. M., Utgoff, P. E., Nudel, B. and Banerji, R.
Learning problem-solving heuristics through practice.
In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 127-134. Vancouver, August, 1981.
- Mitchell, T. M.
Toward Combining Empirical and Analytic Methods for Learning Heuristics.
In Elithorn, A. and Banerji, R. (editors), *Human and Artificial Intelligence*, Erlbaum, 1982.
- Mitchell, T.
Learning and Problem Solving.
In *Proceedings of IJCAI-8*, to appear. Karlsruhe, Germany, August, 1983.
- Mostow, D. J.
Transforming declarative advice into effective procedures: a heuristic search example.
In Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (editors), *Machine Learning*, Tioga Press, Palo Alto, C.A., 1982.
- Mostow, D.J.
Operationalizing Advice: A Problem-Solving Model.
In *Proceedings of the Second International Workshop on Machine Learning*, pages . Urbana, Illinois, June, 1983.
- Utgoff, P. E. and Mitchell, T. M.
Acquisition of Appropriate Bias for Inductive Concept Learning.
In *Proceedings of the 1982 National Conference on Artificial Intelligence*. Pittsburgh. August 1982.

PROGRAM SYNTHESIS AS A THEORY FORMATION TASK: PROBLEM REPRESENTATIONS AND SOLUTION METHODS*

Saul Amarel**
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

EXTENDED ABSTRACT

The long-term objective of this research is to understand theory formation processes and to develop effective methods for mechanizing them. It has been clear for some time that we need to make progress in this area of AI in order to develop applications that will have serious impact on the "doing of science." Recent work shows that advances in theory formation are of a more fundamental significance for AI, especially in developmental issues related to problem solving. Theory formation capabilities are necessary for problem solving systems that can improve their performance by learning from experience; in particular, systems that can shift the representation of the problem on which they are working via the effective use of new knowledge discovered during their problem solving activity [See Amarel, 1982].

The specific focus of our work has been on the interactions between problem representations and methods for searching/building solutions in theory formation problems, and also on processes for moving between alternative problem representations. We are exploring these issues in the context of a specific program formation (synthesis) task in the mathematical domain of finite partially ordered structures. This is an extremely rich domain with many applications to computing. In particular, we are considering the following problem: given a language of programs, and the specification of a function (such as Infimum) in the form of a table of associations between data elements in the domain; build a program that defines the function.

The problem that we face here is how to search in the given language of programs for a program that satisfies the constraints imposed by the given set of associations. This can be regarded as a constraint satisfaction problem, where the constraints are presented in a data space, and the possible solutions are presented in a second space -- the program space; and furthermore, the initial body of available knowledge is not in a form that permits useful links to be readily established between the two spaces. An effective solution method requires the development of strong links between the two spaces, and the

establishment of a strategy for good coordination between the search/reasoning activities in these spaces. Problems of this type have been studied by Simon and Lea [1974].

In previous work, we developed two approaches for solving our program formation problem. The first approach [See Amarel, 1962] started by specifying a grammar for the language of programs, which provided rich facilities for a hierarchical description of the structure of programs. The choice of the grammar was essentially arbitrary. We then assumed that certain types of regularities exist between structured descriptions of programs obtained in the grammar and the functional behavior of these programs; and we formulated an evaluation function which ranked candidate programs by their degree of partial success in satisfying the given functional constraint of the problem, i.e. the table of data associations. Using the grammar, the evaluation function and the assumptions of regularity, we formulated a "Generate-and-Test" procedure, which was guided by a heuristic hill-climbing search strategy in program space. No reasoning in data space took place here. The problem data was used mainly for testing a candidate program/hypothesis, and only very weakly for a priori control of program generation. Even though a moderate level of performance was obtained by this approach, it became clear very soon that performance was highly sensitive to the choice of program grammar. Small changes in the way of specifying the grammar had strong effects on overall system performance. This experience convinced us of the centrality of the problem of representation in the formation process; and it led us to the development of a second approach for handling the problem.

The second approach [See Amarel, 1971] is based on a model of program space which enables us to reason about programs in terms of their functional characteristics. We constructed such a model in the form of a modified algebra of relations whose terms correspond to structural descriptions of programs. Given the model, it was possible to formulate a strategy of search in problem space which is more goal-directed than in our first approach. The search involves obtaining estimates of "distance" in program space between candidate programs and the goal program, and applying "distance reducing" moves in program space that are chosen on the basis of algebraic properties of the model. In this approach, a considerable amount of specific knowledge about program space had to be used -- to constrain the choice of program

*This research was supported in part by the Division of Research Resources, NIH, Grant RR00643 to the Rutgers Research Resource on Computer in Biomedicine.

**Visiting Research Fellow at the AI Center, SRI International, Menlo Park, California, during the Spring of 1983.

grammar, and to provide a more solid basis for the assumptions of structure-function regularity that are needed in the problem solving process. This approach resulted in a more robust (and an improved level of) performance in solving our problem by improving the representation of program space relative to the given task. Again, no direct reasoning in data space was involved.

In recent work we are concentrating on a third approach to program formation, where the process is strongly guided by the detailed analysis of individual data associations. In contrast to the previous two approaches where the main thrust of reasoning is from candidate programs to data associations, the present approach is mainly data driven; but it is strongly guided by the program grammar and by the algebraic properties of the model of program space. Given a data association, a derivation problem is solved to obtain a "computational path" between the two elements of the association. The program grammar provides the reduction schemes and the moves for solving the derivation problem. The key problem that we face here is how to combine the "computational paths" of several data associations in order to infer a single program that can be seen as generating them; and how to merge programs that provide partial solutions to our program (by satisfying subdomains of the function) into a program that represents a global solution for the entire domain. Algebraic properties of program space can help in this merging process.

By shifting the reasoning towards the data space, several interesting domain-dependent problems of concept specialization and generalization have been encountered. In general, the present approach appears more flexible and more general than our previous two approaches.

Our recent work in this area has resulted in the identification of several basic issues in AI and in theory formation. Clearly, the degree of *dependence* between the conditions (or the goals) of a problem is of fundamental importance for the choice of a problem solving method. More research is needed on methods for handling conjunctive goal situations, not only for planning or design problems but also for theory formation problems. Also, it became evident that the notion of problem formulation in theory formation is somehow fluid and open-ended. While our present formulation of the theory formation task is oriented to the finding of a definition for a *given* concept in terms of a given language, this represents a special case of theory formation. In our case, the concept is well-specified from outside the system in terms of a concrete domain which we are asked to express in the language of the theory. More generally, theory formation involves both the choice of a (sub)domain to express within a language of theory, *and* the finding of an expression for defining/explaining the (sub)domain in the theoretical terms of the language. The interplay between domain choice and the formation of a theory for the domain is an interesting and difficult problem, which we have started to explore.

REFERENCES

- Amarel, S., "On the Automatic Formation of A Computer Program which Represents a Theory," in Self Organizing Systems - 1962, Yovits, Jacobi, Goldstein. (Editors), Spartan Books, 1962.
- Amarel, S., "Representations and Modeling in Problems of Program Formation," in Machine Intelligence 6, Meltzer and Michie (Editors), University of Edinburgh Press, 1971.
- Amarel, S., "Expert Behavior and Problem Representation," in Artificial and Human Intelligence, Elithorn and Banerji (Editors), North-Holland Publishing Co. (to appear); also available as Technical Report CBM-TR-126, LCSR, Rutgers University, 1982.
- Simon, H.A. and Lea G., "Problem Solving and Rule Induction: A Unified View," in Knowledge and Cognition, Gregg (Editor), Lawrence Erlbaum, 1974.

Mechanisms for Qualitative and Quantitative Discovery¹

Pat Langley
Jan Zytkow
Herbert A. Simon
Gary L. Bradshaw
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213 USA

ABSTRACT

Scientific discovery is a complex activity involving many different components. Our interest in discovery has led us to construct four AI systems that address different facets of the scientific process. BACON.6 focuses on discovering empirical laws that summarize numerical data. This program includes heuristics for finding numerical relations in noisy data, postulating intrinsic properties such as mass and specific heat, and noting common divisors. GLAUBER is concerned with discovering laws of qualitative structure, such as the hypothesis that acids react with alkalis to form salts. It includes heuristics for noting recurring patterns, for defining classes based on those patterns, and for combining classes that have many elements in common. STAHL focuses on determining the components of substances involved in reactions, and has been used to model the reasoning that led to the phlogiston theory. DALTON is concerned with formulating molecular models that explain reactions such as those occurring in chemistry and particle physics. It carries out a search through the space of possible models, using information it has acquired in other contexts to direct its search down promising paths. While each of these discovery systems is interesting in its own right, we are also exploring ways in which the systems can interact to help direct each other's search process.

Key Terms: scientific discovery, quantitative laws, qualitative laws, chemical reactions, classification, theory of acids and bases, phlogiston theory, atomic theory, molecular models

Introduction

Science is a complex endeavor, involving such diverse activities as designing experiments, inventing new measuring devices, discovering empirical laws, formulating explanations, and testing hypotheses. Until recently, philosophers and historians of science have focused on the criteria for verification and testing of hypotheses, which were amenable to logical analysis. The discovery process was largely ignored, since there seemed no logically valid methods for approaching this elusive activity. However, in recent years, researchers in computer science and cognitive psychology have developed powerful new approaches to modeling thought. The most central ideas of this new methodology are the notion of *search* through a problem space, and the notion of *heuristics* for directing that search. Although these concepts were originally developed with respect to problem solving, they are equally applicable to explaining the mechanisms that lie behind scientific discovery.

In this paper we examine scientific discovery in the light of such notions as search and heuristics. Of course, there are many different types of discovery, ranging from finding numerical laws, to classifying similar objects and events, to formulating structural models to explain regularities. In the following pages, we address each of these forms of discovery. Our approach has been to consider some period from the history of science, and then to construct an AI system capable of rediscovering laws that were first found during that period. Thus, while we have not attempted to replicate the historical discovery process in detail, our ideas have been influenced by the historical data. We have organized the paper around four discovery programs that we have developed in this manner. In each case, we discuss the historical context, describe the system in terms of the heuristics it employs, and consider the manner in which these heuristics let the system discover the historical laws. Most of our examples are drawn from the history of chemistry, but none of the systems are limited to this domain. In closing, we discuss the relations between the various programs, and examine the potential for constructing an integrated discovery system that includes these programs as components.

Discovering Quantitative Empirical Laws

At the very end of the 18th century and the beginning of the 19th, three fundamental discoveries were made that shaped the directions of chemical research for several generations thereafter. The first of these was Proust's (1799) statement of the law of constant proportions. Proust conducted a painstaking analysis of chemical compounds, finding that the ratio of the combining weights of the constituent elements was always constant for a particular compound. The second fundamental advance was Dalton's (1804) introduction of the law of multiple proportions. This law asserts that when two elements combine to form several different compounds, the ratios of their combining weights are always small integer multiples of one another. The third advance was Gay-Lussac's (1809) discovery of the law of combining volumes for gaseous reactions, which states that gases combine with each other in very simple ratios by volume.

These three discoveries provided the foundation for a quantitative theory of chemical reactions, and ultimately led to the determination of the atomic weights of the elements. To the extent that the work of Dalton and Gay-Lussac was motivated by the atomic hypothesis, these discoveries were not products of pure data-driven induction but were partly guided by theory. However, we shall see that data-driven induction is sufficiently powerful to attain the laws without making use of a full-blown atomic hypothesis.

¹ Reprint requests should be sent to Pat Langley, The Robotics Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213. This work was supported by Contract N00014-82-0168 from the Office of Naval Research.

Finding Numerical Relations in Noisy Data

We have explored the process of quantitative discovery through BACON.6, the sixth in a line of programs named after Sir Francis Bacon (1561-1626). Given a set of independent variables, BACON.6 varies one of the terms, looking for relations between that term and some dependent variable. Once a functional relation has been found, the parameters in that function are given the status of dependent terms at a higher level of description. When the system varies the next independent term, it looks for a relation between that variable and the new higher level terms. This process continues, with BACON.6 recursing to higher levels of description, until all the independent terms have been incorporated into a complex quantitative relationship. We will not discuss this process in any great detail, since it has been described for earlier versions of BACON elsewhere [Langley, 1981, Langley, 1982].

Unlike previous versions the current system is capable of dealing with significant amounts of noise in its data. The program uses a differencing technique to find the best polynomial function for relating two terms. However, it also considers polynomial relations between *functions* of these terms, so that relations such as $y^3 = ax^2 + bx + c$, and $\sin(y) = a \log(x) + b$ can be found. The differencing method accepts any relation that accounts for more than a user-specified percentage of the variance in the data. When this requirement for explained variance is high, BACON.6 behaves much like its predecessors: if the data are noise-free, it generates only a single hypothesis; however, if the data are noisy, it fails to find any relation at all. In contrast, when the setting is low, the system tends to generate a number of alternate hypotheses, whether the data are noisy or not.

In cases where a number of competing hypotheses have been generated, BACON.6 must have some way to order these hypotheses. In addition to the explained variance, the system takes into account the *complexity* of each law, measuring this by the number of terms that make up the polynomial expression. The user can specify the exact role played by the two criteria, but in our experiments with the system, we have found that the *ratio* of explained-variance to complexity gives good results. Once the score for each hypothesis has been calculated, a threshold score is computed, and only those laws exceeding the threshold are retained. BACON.6 also takes the *generality* of each law into account. For example, if two laws are found to describe the relation between y and z when $x = 1.0$, but one of these laws does not fit well when $x = 2.0$, then that law will be rejected. In this way, the system ensures that only hypotheses holding across a broad range of data are retained.

Postulating Intrinsic Properties

While the above heuristics are useful for discovering relations between numerical terms, they cannot be used to relate *nominal* or symbolic independent terms to numeric dependent variables, and this is precisely the situation in which the early chemists found themselves. For instance, the independent terms in Proust's, Dalton's, and Gay-Lussac's chemical experiments were the elements or compounds involved, while the dependent terms were numerical measures such as weight or volume. In such cases, BACON.6 defines *intrinsic properties* that take on numeric values, and then associates these properties with the nominal terms.

ELEMENT ₁	ELEMENT ₂	COMP.	W ₁	W ₂	W ₂ /W ₁
NITROGEN	OXYGEN	NO	1.0	1.14	1.14
NITROGEN	OXYGEN	NO	2.0	2.28	1.14
NITROGEN	OXYGEN	NO	3.0	3.42	1.14
NITROGEN	OXYGEN	N ₂ O	1.0	0.57	0.57
NITROGEN	OXYGEN	N ₂ O	2.0	1.14	0.57
NITROGEN	OXYGEN	N ₂ O	3.0	1.71	0.57
NITROGEN	OXYGEN	NO ₂	1.0	2.28	2.28
NITROGEN	OXYGEN	NO ₂	2.0	4.56	2.28
NITROGEN	OXYGEN	NO ₂	3.0	6.84	2.28

Table 1. Determining the combining weights for reactions.

Let us consider the role of intrinsic properties in BACON.6's rediscovery of the early chemical laws. Given control over the substances entering and resulting from a reaction, as well as the weight of the first substance that is used, the system gathers data like those shown in Table 1. Upon varying the amount of oxygen used to form nitric oxide (NO), the program discovers that the two weights w_1 and w_2 are linearly related with a slope of 1.14 and an intercept of zero. Upon varying the output of the reaction, BACON.6 then examines the weight relations for the compound nitrous oxide (N₂O). In this case, the law is also linear, but the slope has changed to 0.57. A similar result is obtained when the system examines the values for nitrogen dioxide, and in this case the slope is 2.29.

The slopes that BACON.6 finds in these experiments are closely related to the weight ratios found by Proust. Having found these ratios, the program summarizes its results at a higher level of description, as shown in Table 2, and treats these summaries as data. The table also includes the results obtained for two reactions of oxygen and carbon. In this case, the system finds three nominal independent terms and a single numeric dependent variable, so it defines an intrinsic property (say p) whose values are associated with the three nominal values under which they occur. Thus, the value of p for the triple nitrogen/oxygen/nitric oxide would be set to 1.14, the value for nitrogen/oxygen/nitrous oxide would be 0.57, and the value for nitrogen/oxygen/nitrogen dioxide would be 2.29. As stated, these intrinsic values simply store an already known fact, and in this sense they are tautological. However, they can be retrieved in future experiments involving the same chemicals, and used to make predictions or to discover new empirical laws.

Noting Common Divisors

As we have seen, Proust's insight about combining weights laid the groundwork for Dalton's law of multiple proportions. This law stated that in cases where two elements combine to form *different* compounds, the ratios of their combining weights were always small integer multiples of one another. BACON.6 includes a heuristic which lets it discover just such a relation in the data from Table 2. This heuristic operates whenever the system defines a new intrinsic property, and examines the values of the new property to see if they (or their inverses) have a common divisor. This technique is especially useful when intrinsic values are associated with multiple nominal values, as often occurs in chemistry. We have described both the intrinsic property and common divisor methods at length in earlier papers [Bradshaw, 1980, Langley, 1983].

In this case, BACON notes that 1.14, 0.57, and 2.28 have the common divisor 0.57, and would replace these intrinsic values with their corresponding integers 2, 1, and 4. In addition, the program defines a higher level intrinsic property based on the divisors it finds in different situations, and associates the divisors with those cases. Thus, the common divisor 0.57 would be associated with the nitrogen/oxygen pair, while the divisor 1.33 would be associated with carbon and oxygen. These relations are formally equivalent to Dalton's law of multiple proportions. BACON takes a similar path in discovering Gay-Lussac's common divisors for combining volumes, and has even arrived at the correct relative atomic weights of hydrogen, oxygen, and nitrogen from data similar to those in Table 1. Thus, BACON's discovery mechanisms account for many of the quantitative laws found by chemists in the early 19th Century.

ELEMENT ₁	ELEMENT ₂	COMP.	W_2/W_1	P	W_2/W_1P
NITROGEN	OXYGEN	NO	1.14	2	0.57
NITROGEN	OXYGEN	N ₂ O	0.57	1	0.57
NITROGEN	OXYGEN	NO ₂	2.28	4	0.57
CARBON	OXYGEN	CO	1.33	1	1.33
CARBON	OXYGEN	CO ₂	2.66	2	1.33

Table 2. Noting common divisors for chemical reactions.

Expectation-Driven Discovery

The discovery heuristics we examined above can all be characterized as *data-driven* techniques. In other words, although they search a predefined space of hypotheses, the directions followed during that search are largely controlled by the data that are observed. However, the current version of BACON (along with its immediate predecessor, BACON.5) also includes *expectation-driven* discovery heuristics. These range from very simple rules for expecting the same *form* of law to hold across different contexts, to much more sophisticated assumptions about symmetric laws. The second of these is employed in cases involving two objects, each having the same set of associated variables. For instance, in a collision between two balls, one can measure the initial and final velocities for each object. In such situations, BACON.6 first finds a partial law relating the terms associated with one of the objects, and then assumes that the final law will have a symmetric form. Using this technique, the system has rediscovered Snell's law of refraction, Black's specific heat law, and conservation of momentum. BACON's expectation-driven methods have been described in greater detail in an earlier article, so we will not elaborate on them here [Langley, 1982].

Such heuristics simplify BACON's task in two important respects. First, they allow the system to consider a few alternate hypotheses in a top-down manner, rather than forcing it to systematically search the entire hypothesis space in a bottom-up fashion. Second, these heuristics greatly reduce the data the system must collect in order to determine laws. Since the program expects laws of only a few alternate forms to occur, it need gather only enough observations to determine parameter values and to see which of these fares best. In addition, BACON's expectation-driven methods simplify the discovery process with very little loss in generality. This results from the fact that they obtain their expectations not from domain-specific knowledge, but from relations that have been found by the data-driven methods. This interaction between data-driven and expectation-driven techniques lets BACON take advantage of its early discoveries to direct its search at later stages.

Comments

In the preceding pages, we have described BACON's inputs, outputs, and the heuristics it employs to generate one from the other. By itself, BACON provides an interesting and useful account of the discovery of quantitative empirical laws. However, the system leaves some important questions unanswered. For example, how does BACON decide which variables to employ in the experiments it runs? Similarly, how are the laws generated by the system put to use? In the current version, the representation of data is provided by the programmer, and the laws are simply printed on a terminal screen. One can imagine a version of BACON with an improved user interface, serving as a scientist's aide in analyzing data, and fulfilling a useful function while still requiring its user to design its input and interpret its output. This is one direction in which the system might be extended, and it should probably be done.

However, if one's goal is to understand the nature of scientific discovery, then a more serious answer to the above questions is required. The introduction of expectation-driven heuristics provides a partial answer to the second question, stating that laws found in one context are used to reduce the search for laws in other contexts. However, we know from the history of science that laws also lead eventually to theories and explanations, and BACON has little to say about such aspects of discovery. This suggests that we will find answers to both questions only by studying other facets of the discovery process. Although constructing AI models of these components would undoubtedly be interesting even in isolation, the true advantage will come from exploring the interrelations between different forms of discovery. Our long-range goal, then, should be to understand components of the discovery process whose outputs can be used as BACON's inputs, and to uncover other components which can employ BACON's outputs as their inputs. In the remainder of the paper, we focus on three different models of discovery that we have constructed to this end, and we close with some speculations on possible interactions between the various systems.

Discovering Qualitative Empirical Laws

Although BACON's heuristics are quite useful for discovering quantitative relations, they cannot be applied when only qualitative information is available.² For example, let us consider the state of chemistry before scientists discovered theories involving quantitative measures such as weight and volume. By the 17th and 18th Centuries, chemists had made considerable progress in classifying substances on the basis of qualitative properties. During this period, researchers focused on features such as the taste and texture of substances, as well as their interactions with other substances. Thus, they knew that the substance we now call hydrochloric acid had a sour taste, and that it combined with ammonia to form NH₄Cl (though the structure of this compound was not known). Moreover, they knew that sulfuric acid also tasted sour, and that it also combined with ammonia to form (NH₄)₂SO₄. From such facts as these, the early chemists defined classes such as *acids*, *alkalis*, and *salts*, and formulated laws involving these terms, such as "acids taste sour" and "acids react with alkalis to form salts". Eventually, they came to view both alkalis and metals as special cases of the more abstract concept of a *base*, and arrived at the more general law that "acids react with bases to form salts".

²The intrinsic property heuristic does let one associate numbers with symbols, but some numbers must be present at the outset.

A Qualitative Discovery System

In our efforts to understand the process by which chemists arrived at the notions of acids and bases, and by which they developed theories involving these terms, we constructed a second discovery system that accepts qualitative facts as input. We have named the program GLAUBER, after the 17th Century German chemist who played an important role in developing the concepts of acids and bases. Before considering the program's heuristics, we should examine its representation of data, since this representation is closely linked to its discovery methods.

GLAUBER represents facts using a simple frame-like structure. Each fact contains a *predicate* that specifies the type of fact it is, along with one or more attribute-value pairs. For example, the fact that HCl reacts with NH_3 to form NH_4Cl would be stored as (reacts inputs (HCl NH_3) outputs (NH_4Cl)). Here the predicate is *reacts*, the attributes are *inputs* and *outputs*, and the lists (HCl NH_3) and (NH_4Cl) are the values for these attributes. GLAUBER knows that the order of symbols in a value list does not matter, so that (reacts inputs (NH_3 HCl) outputs (NH_4Cl)) would be considered identical to the above fact. The knowledge that HCl tastes sour would be stored as (has-quality object (HCl) taste (sour)). In this case the values are enclosed in parentheses for consistency with other predicates (such as *reacts*), which may have multiple symbols as values. The reason for not representing this fact in the simpler form (sour object (HCl)) will become apparent shortly.

Like BACON, the GLAUBER program is largely data-driven, in that the data it observes play a major role in determining the hypotheses it generates. GLAUBER's first heuristic searches for recurring patterns in these data; it may be stated as:

**If two facts have the same predicate,
and have the same value for the same attribute,
then create a pattern identical to these facts, but in
which differing values are replaced by class names**

For instance, given the two facts (reacts inputs (HCl NH_3) outputs (NH_4Cl)) and (reacts inputs (H_2SO_4 NH_3) outputs ($(\text{NH}_4)_2\text{SO}_4$)), the system would note that the symbol NH_3 occurred in the value of the *inputs* property in both cases. As a result, it would define two classes (say class-1 and class-2), and create an abstract pattern in which the class names replace the differing symbols, but which retains the constant term NH_3 . In this case, the pattern would be (reacts inputs (class-1 NH_3) outputs (class-2)). Given this strategy, GLAUBER must represent the taste of an object as a value rather than as a predicate, if it is ever to note recurring tastes.

As soon as the system creates such a pattern, a second heuristic comes into play, which is responsible for assigning appropriate substances to the classes that have been defined. It may be paraphrased:

**If you have just created a pattern,
then find all facts that fit this pattern,
and add the values which differ in those facts
to the appropriate classes.**

This rule iterates through the list of known facts, checking each one to see if it obeys the new pattern. This occurs when a fact has the same predicate as the pattern, and when all non-class symbols are the same. Minimally, the two facts that first led to the pattern's creation will match, and the symbols that differ in these facts will be added as members of the class mentioned in the pattern. In our example, the chemicals HCl and H_2SO_4 would be stored as members of class-1, while NH_4Cl and $(\text{NH}_4)_2\text{SO}_4$ would be stored with class-2. Since the substance NH_3 occurs in both inputs, it would not be added to any class at this time.

Table 3 presents a set of facts that were generally known by chemists in the 17th Century. This set includes information about reactions between chemicals, as well as information about the tastes of individual substances. When GLAUBER was presented with this set of propositions, it arrived at a class of sour-tasting substances, a class of bitter-tasting ones, a class of salty ones, and some 12 classes defined in terms of reactions with other chemicals.

```
(reacts inputs (HCl NaOH) outputs (NaCl))
(reacts inputs (HCl KOH) outputs (KCl))
(reacts inputs (HCl  $\text{NH}_3$ ) outputs ( $\text{NH}_4\text{Cl}$ ))
(reacts inputs ( $\text{HNO}_3$  NaOH) outputs ( $\text{NaNO}_3$ ))
(reacts inputs ( $\text{HNO}_3$  KOH) outputs ( $\text{KNO}_3$ ))
(reacts inputs ( $\text{HNO}_3$   $\text{NH}_3$ ) outputs ( $(\text{NH}_4)\text{NO}_3$ ))
(reacts inputs ( $\text{H}_2\text{SO}_4$  NaOH) outputs ( $\text{Na}_2\text{SO}_4$ ))
(reacts inputs ( $\text{H}_2\text{SO}_4$  KOH) outputs ( $\text{K}_2\text{SO}_4$ ))
(reacts inputs ( $\text{H}_2\text{SO}_4$   $\text{NH}_3$ ) outputs ( $(\text{NH}_4)_2\text{SO}_4$ ))
(has-quality object (HCl) taste (sour))
(has-quality object ( $\text{HNO}_3$ ) taste (sour))
(has-quality object ( $\text{H}_2\text{SO}_4$ ) taste (sour))
(has-quality object (NaOH) taste (bitter))
(has-quality object (KOH) taste (bitter))
(has-quality object ( $\text{NH}_3$ ) taste (bitter))
(has-quality object (NaCl) taste (salty))
(has-quality object (KCl) taste (salty))
(has-quality object ( $\text{NH}_4\text{Cl}$ ) taste (salty))
(has-quality object ( $\text{NaNO}_3$ ) taste (salty))
(has-quality object ( $\text{KNO}_3$ ) taste (salty))
(has-quality object ( $(\text{NH}_4)\text{NO}_3$ ) taste (salty))
(has-quality object ( $\text{Na}_2\text{SO}_4$ ) taste (salty))
(has-quality object ( $\text{K}_2\text{SO}_4$ ) taste (salty))
(has-quality object ( $(\text{NH}_4)_2\text{SO}_4$ ) taste (salty))
```

Table 3. Chemical facts presented to GLAUBER.

If the early chemists had stopped at this point, their developing science might have remained a mass of isolated regularities. However, they also noted that certain patterns tended to occur *together*, and GLAUBER uses a third heuristic to notice such co-occurrences:

**If two classes A and B
have enough members in common,
then combine them into a single class C,
and store the patterns for A and B with C.**

This rule would note, for example, that the class of HCl-reacting substances has the same members as the class of HNO_3 -reacting substances, and would define a new class (let us call it the class of *alkalis*) that subsumes both of the original classes. The more specific classes would be abandoned, but their associated patterns would be stored with the alkali concept.

Thus, the alkali class would have two associated patterns - one stating that alkalis tend to react with HCl, and another stating that they also tend to react with HNO_3 . After making this combination, the same heuristic would note that the alkali class and the class of substances that react with H_2SO_4 also have identical members. The process would be repeated, this time replacing alkalis and the H_2SO_4 -reacting class with another class (say *alkalis'*). This new class would have three associated patterns, one referring to reactions with HCl, another to reactions with HNO_3 , and a third to reactions with H_2SO_4 . GLAUBER continues to apply the combination heuristic until all classes with enough members in common have been combined. The exact meaning of the phrase "enough in common" depends

on the setting of a system parameter; however, we have kept it close to 100 percent to avoid the combination of all classes into one superclass.

Note that when two classes with non-identical members are combined, GLAUBER is effectively deciding to incorporate a pattern into the definition of a concept even though that pattern does not hold for all examples of the concept. Thus, if most alkalis tasted bitter – but not all of them – the system would still add this as one of the main characteristics of alkalis, even though exceptions to the rule exist. This is an important characteristic of GLAUBER's approach to discovery, since there are few concepts in any branch of science that do not include some exceptions. In order to avoid overgeneralizations, the system stores the names of substances obeying each particular pattern.

A closely related heuristic applies when the members of one class appear to be a subset of the members of another. In such cases, the subset class is retained, but its associated patterns are copied over to the superset class and stored with it as well. With respect to the acid/alkali data shown in Table 3, this situation occurs for various classes of salts. For instance, GLAUBER defines one class consisting of substances that have salty tastes, and another composed of compounds that result from reactions with HCl. Since the second of these classes is a subset of the first, the latter's pattern is copied (with the first class's name being substituted for the second) and added to the first class's set of associated patterns. Table 3 presents the three main classes that GLAUBER settles on using its two heuristics for combining classes, along with their members and associated patterns; these classes are very similar to the 18th Century chemists' notions of acids, alkalis, and salts.

ACIDS: {HCl, HNO₃, H₂SO₄}
 (reacts inputs (*acid* NaOH) outputs (*salt*))
 (reacts inputs (*acid* KOH) outputs (*salt*))
 (reacts inputs (*acid* NH₃) outputs (*salt*))
 (has-quality object (*acid*) taste (*sour*))

ALKALIS: {NaOH, KOH, NH₃}
 (reacts inputs (HCl *alkali*) outputs (*salt*))
 (reacts inputs (HNO₃ *alkali*) outputs (*salt*))
 (reacts inputs (H₂SO₄ *alkali*) outputs (*salt*))
 (has-quality object (*alkali*) taste (*bitter*))

SALTS: {NaCl, KCl, NH₄Cl, NaNO₃, KNO₃, NH₄NO₃,
 Na₂SO₄, K₂SO₄, (NH₄)₂SO₄}
 (reacts inputs (*acid* NaOH) outputs (*salt*))
 (reacts inputs (*acid* KOH) outputs (*salt*))
 (reacts inputs (*acid* NH₃) outputs (*salt*))
 (reacts inputs (HCl *alkali*) outputs (*salt*))
 (reacts inputs (HNO₃ *alkali*) outputs (*salt*))
 (reacts inputs (H₂SO₄ *alkali*) outputs (*salt*))
 (has-quality object (*salt*) taste (*salty*))

Table 4. Classes retained by GLAUBER after combination.

At this point, GLAUBER has arrived at a set of classes that accurately summarize the data, but its patterns are still fairly specific. Thus, it has defined a class of substances with sour, bitter, and salty tastes, but it does not yet have a general law relating acids, alkalis, and salts. To arrive at such a law, GLAUBER must employ a final heuristic that is very similar to BACON's rule for recursing to higher levels of description; it may be paraphrased:

If you have found all the patterns that you can,
 and combined all the classes that you can,
 then treat the known patterns as facts,
 and apply the other heuristics to them.

This rule leads the program to examine the patterns shown in Table 4 and to treat them as if they were data, thus generating new classes and patterns. For instance, given these data, the first heuristic discussed above would note the recurring symbols *alkali* and *salt* in the "facts" (reacts inputs (HCl *alkali*) outputs (*salt*)) and (reacts inputs (HNO₃ *alkali*) outputs (*salt*)). As a result, it would define a new class (say *alkali-reactors*) and use it in the higher level pattern (reacts inputs (*alkali-reactor* *alkali*) outputs (*salt*)). After this, the second heuristic would add the chemicals HCl, HNO₃, and H₂SO₄ to this class, since each of them occurs in a second level fact that fits the pattern. Later, after other higher level patterns and classes have been defined, our third heuristic would note that the class of *alkali-reactors* has the same members as the class of *acids*, and combine the two categories. When this process is complete, GLAUBER is left with basically the same classes as shown in Table 4, except that each has the associated pattern "acids react with alkalis to form salts".³

When GLAUBER is presented with facts about metals (such as their texture and reactions) as well as information about alkalis, the system follows a somewhat more complex, but basically similar, path. It formulates a class of HCl-reacting substances, consisting of both metals and alkalis, very early on, as well as identical classes based on reactions with the other acids. The class of alkalis (as distinct from metals) is originally defined entirely in terms of taste, while metals are defined in terms of texture. Since both of these classes are subsets of the various acid-reacting classes, their associated patterns are copied over to the supersets. After combining overlapping classes and recursing multiple times, GLAUBER arrives at concepts for alkalis, metals, and bases which are similar to those defined by the early chemists.

Limitations and Extensions

In its present form, GLAUBER has a number of important limitations, and these should be remedied in future versions of the system. For one thing, the existing representation of qualitative laws is ambiguous. For instance, the pattern (reacts inputs (HCl class-1) outputs (class-2)) contains implicit quantifiers on the terms class-1 and class-2, but it is not clear whether these are universal or existential. The following interpretation seems to be a useful one. Whenever GLAUBER generalizes by replacing constant terms with class names, a number of alternate patterns should be produced. In each of these, one of the class names is universally quantified, with the remaining classes being existentially quantified (with the universal quantifier occurring first). As a special case, if only one class is created, it is universally quantified.

Thus, if N classes are introduced, N alternate patterns are created. For instance, in the above example, two statements would result: "for all members X of class-1, there exists a member Y of class-2, such that HCl reacts with X to form Y", and "for all members Y of class-2, there exists a member X of class-1

³We have used the names "acid", "alkali", and "salt" in the table for the sake of clarity, though of course GLAUBER does not generate them.

such that HCl reacts with X to form Y ". Upon comparison with the data, an extended version of GLAUBER might decide that only one of these statements was accurate, or it might decide that both were useful summaries of the observed phenomena. The ambiguity of the current representation has not yet caused difficulties because both interpretations hold for the theory of acids and bases. However, one can imagine domains in which only one interpretation describes the data, and the system's representation of laws and its decision processes should be extended to deal with these cases.

A second limitation is that the existing version fails to find recurring relations between *conjunctions* of facts. For instance, suppose one observed (as Mendel did) that when certain green garden peas were self-fertilized, they produced only green offspring, but that when other green peas were self-fertilized, they produced both green and yellow children. In this case, we would like GLAUBER to divide the green peas into two classes based not on their own directly observable features (since these are identical), but based on the features of their offspring. Thus, in looking for patterns, the system would have to examine not only single facts, but pairs of facts, triples of facts, and so forth. Such a strategy, though much more expensive than the current one, would enable the program to note that some green peas have only green offspring, while others have mixed offspring, and to classify them on this basis.

A final difficulty with GLAUBER relates to the system's search through the space of patterns and classification schemes. The current version completes a run with a number of classes, some having many associated patterns and others having only a few or even one attached pattern (such as the sub-class of salts that are formed from reactions involving HCl). From the early chemists' viewpoint, some of these classes were clearly more important than others, while the GLAUBER treats all classes as equal. Since the system has so far only been tested on carefully crafted data, this strategy has not yet caused any difficulties. However, if the program is ever presented with large set of facts (some of which contain minor or even spurious regularities), it will require some method for focusing attention on some classes in preference to others. In future versions of GLAUBER, we plan to use the number of patterns associated with a class as one measure of importance, since this seems to have been used by chemists in focusing on the concepts of acids, alkalis, and salts rather than on the outputs of reactions involving HCl , NH_3 , and so forth.

Another relevant factor relates to the *empirical content* of a pattern. For instance, consider the statements "acids taste sour" and "acids dissolve metals". The first of these refers to a single property of acids, while the second relates acids to metals. If three substances are known to be acids, while four substances are known to be metals, then the first statement predicts only three facts, while the latter statement predicts twelve different facts. Presumably, patterns that make many predictions should be given greater weight than those making few predictions. Thus, one can imagine measuring the importance of a class as a linear function of its associated patterns, with the empirical content of each pattern acting as coefficients. The details of this scheme remain to be elaborated, but the basic idea of focusing on classes that account for the most data seems to be a plausible approach.

Determining Components of Substances

The forms of discovery we have examined in earlier sections were all concerned with finding empirical laws that summarized observations. However, an equally important form of discovery is concerned with finding *explanations* of phenomena. The distinction between description and explanation is not clear-cut, but the laws generated by BACON and GLAUBER definitely seem to lie at the description end of the spectrum, while such constructs as the phlogiston theory, the atomic theory, and the kinetic theory of gases lie more toward the explanatory end. In our attempt to understand the nature of explanatory discovery, we have implemented two programs that attempt to determine the underlying structure of reactions. In contrast to BACON and GLAUBER, these systems employ knowledge about the nature of reactions to aid in the explanation process, so they are not as general as the earlier empirical discovery programs. In this section we describe STAHL, a system that attempts to determine the components involved in a set of reactions. In the following section we consider DALTON, a program that uses information about the components of substances to construct molecular models of reactions.

Heuristics for Determining Components

A primary goal of the early chemists was to determine the components of various substances, and information about the reactions between substances proved quite useful in this regard. We have implemented a system that infers such components from a set of data about reactions. The program is named STAHL, after G. E. Stahl (1660 - 1734), who played an important role in formulating the phlogiston theory, which was primarily concerned with the components of combustible bodies. Like GLAUBER, this program accepts qualitative information as input, and generates qualitative statements as output. In fact, this information can be represented in the same format, with the input being statements like (reacts inputs (hydrogen oxygen) outputs (water)) and the output being statements like (composed-of substance (water) components (hydrogen oxygen)). However, since STAHL's conclusions relate to the internal structure of substances, they can be viewed as explanations rather than simple descriptive summaries.

As with the earlier systems, STAHL's approach to discovery can be summarized as a set of heuristics. The most basic of these heuristics deals with simple synthesis and decomposition reactions, and lets the system infer the components of a compound. It can be stated:

If A and B react to form C,
or if C decomposes into A and B,
then infer that C is composed of A and B.

An obvious example of this rule's use involves determining the components of water. Given the information that hydrogen reacts with oxygen to form water, STAHL would infer that the latter substance is composed of the first two. Note that STAHL does not draw any conclusions about the *amount* of hydrogen and oxygen contributing to water, but only that they contribute something. Of course, the rule is not limited to reactions involving pairs of elements, but can also deal with cases in which three or more substances unite to form a compound.

If all chemical reactions were of the simple form shown above, STAHL's task would be simple indeed. However, more complex reactions are common in chemistry, so STAHL includes additional heuristics for dealing with these more complex situations. The purpose of these rules is to simplify such reactions, with the goal of transforming them into a form that can be matched by the synthesis/decomposition rule shown above. One such rule is responsible for "canceling" out substances occurring on both sides of a reaction; this "reduction" rule can be paraphrased:

**If A occurs on both sides of a reaction,
then remove A from the reaction.**

This heuristic leads directly to a simplified version of a reaction. For instance, if STAHL is told that "A, B, and C react to form D and C", this rule would apply, giving the simplified reaction "A and B react to form D". This revised relation would then be used by the synthesis/decomposition rule to infer that D is composed of A and B. One can imagine cases in which this approach would lead to errors, as when different amounts of a substance are observed before and after a reaction. However, one can also imagine more conservative versions of the heuristic that require equal amounts of the canceled substance to occur on each side.

STAHL incorporates a third heuristic that initially leads to more complex statements of reactions, but may make it possible for the reduction rule to apply. This heuristic draws on information about the components of a substance that have been inferred earlier; it can be stated:

**If A occurs in a reaction,
and A is composed of B and C,
then replace A with B and C.**

For instance, the system may know that X is composed of Y and Z, and that "X reacts with W to form V and Z". In this case, the substitution rule would let us rewrite the second relation as "Y, Z, and W react to form V and Z". Given this formulation, the reduction rule would lead to "Y and W react to form V", and the synthesis/decomposition rule would lead to the conclusion that V is composed of Y and W. As before, this rule is not restricted to substances composed of two elements, but works equally well for more complex structures.

Two final heuristics let the system determine when two substances that were originally thought to be different are in fact identical. The first of these rules may be stated:

**If A is composed of B and C,
and A is composed of B and D,
then identify C with D.**

This heuristic applies when one learns that a compound can be decomposed in two different ways, but in which those decompositions differ by a only single substance. The second rule is very similar, except that it applies when two apparently different compounds are found to have the same components. It can be paraphrased:

**If A is composed of C and D,
and B is composed of C and D,
then identify A with B.**

The history of chemistry abounds with cases in which a new substance was discovered in two different contexts, was originally thought to be two distinct substances, and was eventually combined into a single concept. We will see an example of such identification shortly.

One of the interesting features of STAHL is the manner in which its heuristics interact. Note that the substitution rule requires knowledge of a substance's composition, so that some inferences about composition must be made before it can be

used. However, we have also seen that complex reactions must be rewritten by the reduction and substitution rules before some composition inferences can be made. This interdependence leads to a "bootstrapping" effect, in which inferences made by one of the rules enables further inferences to be made, these allow additional inferences, and so forth, until as many conclusions as possible have been drawn. This process generally begins with one or more simple reactions, but after this the particular path taken depends on the data available to the system. Below we consider a simple example of STAHL's behavior in relation to the phlogiston theory.

Discovering the Phlogiston Theory

The theory of phlogiston originated early in the 18th Century, and after undergoing several transformations, was widely accepted until the 1780's. This theory adopted the ancient view that fire, heat, and light are different manifestations of a common principle that leaves a body during combustion. Therefore, any reaction involving combustion was viewed as a decomposition; for instance, burning coal was interpreted as decomposing into the matter of fire (another term for phlogiston) and ash.⁴ Early phlogisticians were not able to isolate phlogiston, but the disengagement of fire during combustion seemed to be a good observational reason for admitting the disengagement of a substance from the burning body. Later, as the notion of phlogiston proved useful in explaining many additional reactions, the existence of this substance was supported by a substantial body of evidence.

After they began to study combustion within closed vessels, chemists realized that air was necessary for combustion to occur. However, they did not assume that air changed its chemical identity during this process. Rather, they decided that air played an auxiliary role, similar to that played by water in reactions involving acids, alkalis, and salts. Thus, even starting with empirically more complete descriptions of combustion, such as "in the presence of air, carbon burns to release phlogiston and to form ash", they employed the reduction heuristic to remove air and simplify the relation. Given such data, STAHL makes similar "errors" in reasoning, so that it provides a simple explanation of the process by which chemists developed phlogiston-based models of combustion reactions. Such confusions are common in the history of chemistry, and a similar error led the followers of Lavoisier (around 1810) to believe that sodium was a compound of soda and hydrogen.

Let us examine the path taken by STAHL in arriving at results that are very similar to the phlogiston theory. Suppose we present the system with two facts: (reacts inputs (coal air) outputs (matter-of-fire ash air)) and (reacts inputs (calx-of-iron coal air) outputs (iron ash air)).⁵ One may question the exact representation of these facts, but clearly something very much like them was believed during the period in which the phlogiston theory was developed. Given this information, STAHL would immediately apply its reduction heuristic, giving the revised reactions (reacts inputs (coal) outputs (matter-of-fire ash)) and (reacts inputs (calx-of-iron coal) outputs (iron ash)). The first of

⁴Several decades later, in the second half of the 18th Century, fixed air (carbon dioxide) was discovered, and recognized as the product of burning coal in place of ash.

⁵Calx of iron was the current name for iron oxide; we have used the original terminology because the modern term is based on the oxygen theory developed by Lavoisier.

these revisions, combined with the synthesis/decomposition rule, leads directly to the inference that coal is composed of matter-of-fire (or phlogiston) and ash, which was one tenet of the early phlogiston theory. Having arrived at this conclusion, STAHL would apply the substitution rule, generating the expanded relation (reacts inputs (calx-of-iron ash matter-of-fire) outputs (iron ash)). At this point, the reduction rule is used to remove ash from both sides of the equation, giving (reacts inputs (calx-of-iron matter-of-fire) outputs (iron)). Finally, the synthesis/decomposition heuristic leads STAHL to infer that iron is a compound composed of calx-of-iron and the matter of fire.

Next, let us consider how STAHL employs its identification heuristics with respect to the phlogiston theory. Suppose we give the system the following additional data: (reacts inputs (iron sulfuric-acid water) outputs (vitriol-of-iron inflammable-air water)) and (reacts inputs (calx-of-iron sulfuric-acid water) outputs (vitriol-of-iron water)).⁶ Given these facts, STAHL removes the water from both reactions using the reduction rule. This sufficiently simplifies the second reaction so that one can apply the synthesis heuristic, inferring that vitriol-of-iron is composed of calx-of-iron and sulfuric-acid. This fact is substituted into the first reaction, giving (reacts inputs (iron sulfuric-acid) outputs (calx-of-iron sulfuric-acid inflammable-air)). After eliminating sulfuric-acid from both sides of this expression, STAHL infers that iron consists of calx-of-iron and inflammable air. However, the system knows from the other reactions described earlier that iron can also be decomposed into calx-of-iron and phlogiston. Using the first of its identification heuristics, the system infers that inflammable-air and phlogiston are identical. Both the reasoning and conclusions of STAHL in this example are the same as those of Cavendish and other phlogiston theorists during the 1760's.

Constructing Molecular Models

Once the substances involved in a reaction have been determined, one can begin to construct molecular models of the reaction, in which one specifies the *number* of times each substance occurs in the transformation. After John Dalton set forth the general form of his atomic hypothesis in 1808, the design of such molecular models became of central importance to chemists. Dalton employed his *rule of greatest simplicity* to apply the atomic theory to specific cases, and though this heuristic worked in many cases, it led him to incorrect conclusions in others. For instance, it led him to conclude that water was composed of a single hydrogen atom and a single oxygen atom. In contrast, Avogadro (1811) employed Gay-Lussac's data on combining volumes, along with the assumption that equal volumes of gases contained equal numbers of particles. Using this information, he inferred diatomic models for hydrogen and oxygen and a different structure for water. Although we accept Avogadro's hypothesis today, it was rejected by his contemporaries, since they believed that different atoms of the same element would repel, rather than attract, each other. The explanation for diatomic molecules did not appear until well into the 20th Century, when the theory of covalent bonds was developed.

⁶The formula for vitriol of iron is FeSO_4 , while the modern name for inflammable air is hydrogen.

Searching the Space of Models

In order to further understand the process by which the early chemists arrived at such models, we implemented another discovery system that constructs molecular models of reactions. We have named this program DALTON, since Dalton was the first proponent of an atomic theory that explicitly used structural models. DALTON knows that two quantities are important in any reaction – these are number of *molecules* of each substance that take part (in the simplest form of the reaction), and the number of *particles* in each molecule.⁷ Suppose the system is told that the element hydrogen reacts with the element oxygen to form the compound water, and is asked to construct a model of the components of this process. In this case, the program must determine the number of hydrogen, oxygen, and water molecules, and the internal structure of each type of molecule. The system operates by starting with a very abstract model in which no commitments are made, and successively refines this model as it proceeds. Let us examine what happens at each stage in the search through the space of models.

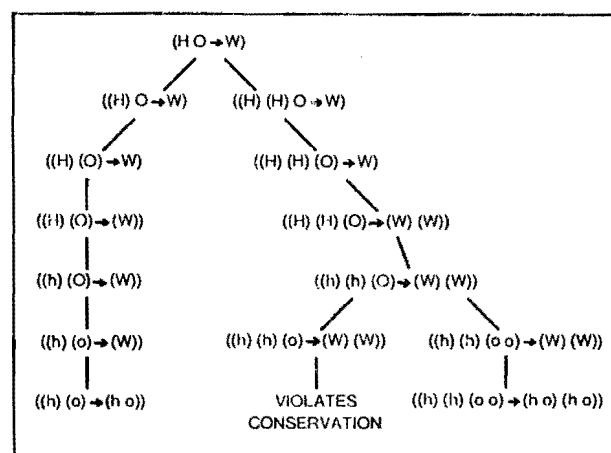


Figure 1. DALTON's search for a model of the water reaction.

Starting with an abstract model of the form $(H O \rightarrow W)$, the program first considers the number of hydrogen molecules involved. Lacking any theoretical bias, the system chooses the simplest hypothesis and assumes a single hydrogen molecule is required. If this choice later causes difficulty, the model-builder can back up and try another path. Similar initial choices are made for oxygen and water, so that the partially specified model includes one molecule each. This is represented by the proposition $((H) (O) \rightarrow (W))$, in which each molecule is enclosed in parentheses.

Now DALTON must determine the internal structure of each type of molecule, and it decides to assume initially that both hydrogen and oxygen consist of a single elementary particle (say *h* and *o*), giving the model $((h) (o) \rightarrow (W))$. At this point, the program invokes the strong theoretical assumption that the total number of particles in any reaction is conserved. This assumption is stated in terms of a condition-action rule that checks to see if the model can be finalized in such a way that

⁷Thus, the program begins with a better notion of the true situation that did its namesake, since Dalton did not make the distinction between atoms and molecules.

conservation is obeyed. If this is possible, DALTON outputs the completed model and halts, but if the conservation principle cannot be satisfied, the system backs up and considers other possibilities. In this case, the conservation heuristic tells DALTON that the water molecule must be composed of h and o particles, and that the final model must have the form ((h) (o) → (h o)). This model is equivalent to the one originally formulated by the human Dalton. Figure 1 presents some of the paths available in the space of molecular models. In arriving at the monatomic model just described, DALTON takes the left path, and since this leads to an acceptable solution, no backtracking is required.

Directing the Search Process

As we have seen, DALTON's basic strategy is to carry out a depth-first search through the space of models, ordering the search so that simple models are considered first. However, when enough of the model has been specified, a theory-driven heuristic (implementing the conservation assumption) takes over and finalizes the model. DALTON can also employ theory-driven methods at other stages in its search process, and these methods can alter the system's behavior in significant ways. Thus, in the above run, the system had no theoretical biases other than a belief in conservation of particles and a desire to construct as simple a model as possible. However, if we give DALTON some additional information about the water reaction, its behavior changes drastically. Avogadro was aware of Guy-Lussac's results, and firmly believed that the combining volumes he observed were related to the number of molecules involved in the reaction. To model this knowledge, we can add the heuristic:

If you want to know the number of molecules of X
that are involved in a reaction,
and the combining volume of X was V,
then consider only multiples of V as possibilities.

Given this assumption (and knowledge of the combining volumes), our program would instead postulate two molecules of hydrogen and water (and if this was later found to be unsatisfactory, four and then six), while it would retain the assumption of one oxygen molecule. Thus, at the third level in the search tree, DALTON would have the partially specified model (H) (H) (O) → (W) (W)).

At this point the system would move to consider the internal structure of the hydrogen molecule, assuming it was composed of a single particle (say h), and would make a similar assumption for oxygen. However, for the resulting model, ((h) (h) (o) → (W) (W)), there exists no decomposition of water in terms of h and o that satisfies the conservation assumption, so the program would back up and consider some other alternative. DALTON would next hypothesize the oxygen molecule as composed of two particles, and since this does allow conservation to be satisfied, a final model would be constructed in which oxygen is diatomic and hydrogen is monatomic: ((h) (h) (o o) → (h o) (h o)). (These two search paths are shown on the right side of Figure 1.) While this model differs from the modern-day one, it is consistent with Guy-Lussac's data and encounters difficulty only when other reactions are considered. For example, the monatomic assumption does not work for the ammonia reaction.

Like most of the programs we have described, DALTON is stated as a production system. In default mode, the system uses a few simple rules to formulate simpler models first, and more complicated ones as necessary. However, if new condition-action rules are added to the system, they take precedence over

the default rules and can direct search down paths that might otherwise not be considered. Thus, one can insert a rule that would match if the combining volume of some substance is known, and use this information to determine the number of molecules used for that substance in the model. The conservation assumption is implemented in a similar fashion, so that it generates a molecular structure of a reaction's output that uses all particles occurring in the input.

Once DALTON has generated a successful model for a reaction, it converts this knowledge into productions. For instance, having arrived at the diatomic explanation of water given above, the program would store one rule concerning the molecules of hydrogen involved, another for oxygen molecules, and a third for the water molecules. If the system is asked to explain the water reaction at a later date, it will be able to recall the number of molecules without search. DALTON also constructs productions describing the internal structure of various molecules, and while this knowledge is useful in re-explaining the water reaction, it is useful in other cases as well. For instance, when asked to model the ammonia reaction, the system would immediately guess that hydrogen was monatomic, based on the success of this assumption in its model of water. None of the models incorporating this assumption satisfy conservation, so the system would back up, hypothesize that hydrogen is instead diatomic, and eventually arrive at the correct model for ammonia.

However, DALTON must also update its model of the water reaction. Since the system knows that its monatomic hydrogen rule was responsible for leading it astray, it removes this rule from memory and replaces it with a diatomic rule for hydrogen. It then retrieves the reaction that led it to construct the monatomic production, and checks to see if the replacement rule works here as well. In this case it does, but had it run into difficulty, the process would have been repeated, with DALTON considering ever more complicated molecular structures (up to a limit), until both the water and the ammonia reaction had been successfully explained by a single rule.

Extensions of the Approach

Although DALTON's methods are concerned with reactions, they are not limited to the chemical domain. For example, the field of elementary particle physics is also concerned with reactions, and with the formulation of "molecular" models to explain those reactions. The most widely accepted theory in this domain accounts for the internal structure of protons, neutrons, and other hadrons in terms of a small set of hypothesized particles called "quarks". In its present form, DALTON cannot discover the quark theory, but two relatively simple extensions should enable the system to arrive at the basic tenets of this framework.

First, the current version of DALTON assumes that molecules of elements (such as hydrogen and oxygen) are composed of a single type of particle (such as hydrogen atoms or oxygen atoms), and this assumption considerably simplifies the search process in the chemical domain. However, there are no directly observable "elements" in the field of particle physics, and in order to explain particle interactions, one must postulate more complex models of hadrons. For example, the basic proton "molecule" is viewed as composed not of three proton "atoms", but as composed of two *u* quarks and one *d* quark. In order to regenerate the quark theory, DALTON must be modified to search the larger space of models in which such decompositions can occur.

An equally important aspect of the quantum theory involves the conservation of mass and of the various quantum numbers, such as spin and electric charge. In order to generate these features of the theory, DALTON must attempt to explain quantitative attributes of directly observable substances in terms of attributes associated with inferred substances such as quarks. Presumably, these constraints can be stated as theory-driven heuristics, much as the conservation of particles assumption is implemented in the current version.

A less obvious application of DALTON involves the field of classical genetics. For example, the hereditary rules for garden peas, first enumerated by Gregor Mendel in 1866, can be viewed as reactions in which characteristics of the parents are transformed into characteristics of the offspring. Given the first extension described above, along with a suitable replacement for the conservation assumption (since this does not apply in reproductive systems), DALTON should be able to arrive at the two-trait model originally formulated by Mendel. Suppose that the system is provided with genotypic statements of the result of inbreeding and crossbreeding, which might be induced by another discovery system (like GLAUBER) from phenotypic descriptions of these reactions.

For example, if we let G stand for green peas that produce only green offspring, Y stand for yellow peas that produce only yellow offspring, and G' stand for green peas that produce mixed offspring, then four basic reactions suffice to describe Mendel's observations: $G G \rightarrow G$, $Y Y \rightarrow Y$, $G Y \rightarrow G'$, and $G' G' \rightarrow G G' Y$. Given these reactions, an extended version of DALTON should be able to infer that two primitive traits (say g and y) are required, and to decide that the genotype G can be modeled by the "molecular" pair ($g g$), that Y can be modeled by the pair ($y y$), and that G' can be modeled by the pair ($g y$). As we envision it, the system's explanation of these reactions would not involve the notion of dominance, nor would it predict the proportions in which the various genotypes are observed, but it would account for the basic qualitative relations between parents and offspring.

Toward an Integrated Discovery System

While each of the systems described above accounts for an important part of the discovery process, a more profitable long-range goal would be to combine these programs into a single, integrated discovery system. One advantage of this approach is that it would increase our understanding of the relations between the various forms of discovery. Another benefit is that it would decrease the degrees of freedom in the individual programs, since the outputs of some would be forced to conform to the inputs of others. In this section, we consider the potential for constructing such a unified system, by exploring the possible interactions between pairs of the individual programs described earlier.⁸

GLAUBER and BACON

Let us begin by examining the potential connections between our two empirical discovery systems, GLAUBER and BACON. Since the first of these deals entirely with symbolic data and the second generates numerical laws, there seems no manner in which BACON's output could be used by GLAUBER. However, the possibilities in the other direction appear much better. The

reader will recall that GLAUBER produces two forms of symbolic knowledge – abstract laws and hierarchies of classes used in those laws – while BACON employs symbolic values for some of its independent terms. One can imagine BACON making two uses of GLAUBER's output. First, it might employ the qualitative laws generated by the latter to design its own experiments, rather than rely on its programmer for this information. For example, suppose BACON were told by GLAUBER that nitric oxide, nitrous oxide, and nitrogen dioxide were all substances that resulted from reactions between nitrogen and oxygen. Given knowledge of this class of compounds, an extended version of BACON might design an experiment in which the substances entering a reaction (oxygen and nitrogen) were held constant, while the output of the reaction was varied. If quantitative variables such as the weights of the substances were examined, the resulting experiment would lead BACON to Dalton's law of multiple proportions (as described in an earlier section).

The second use of GLAUBER's output relates to BACON's generalization process. As it is currently implemented, BACON initially associates intrinsic values with all potentially relevant symbolic conditions, and generalizes by removing conditions whenever it finds that a set of intrinsic values are useful in a new context. However, the availability of the classification hierarchy generated by GLAUBER presents an alternate generalization method. Rather than removing conditions entirely, one can generalize by replacing the symbolic value in a condition with the name of a class containing that value. For instance, suppose BACON has stored a set of intrinsic values, and one of the conditions for retrieval is that one of the substances entering the reaction is HCl. Now suppose that the system finds these intrinsic values useful when the substance is H_2SO_4 instead of HCl. Rather than inferring that this condition is irrelevant, BACON might decide that the intrinsic values should be retrieved whenever an *acid* is involved in the reaction (provided that the concept of acid has previously been defined by GLAUBER). This is a more conservative approach to generalization, and would allow BACON to express a larger class of hypotheses than it currently can. Of course, the system could eventually decide to remove this condition entirely, should the intrinsic values prove useful for non-acids as well.

GLAUBER and STAHL

We have seen that GLAUBER treats the symbols *reacts*, *inputs*, and *outputs* in the same manner that it treats other symbols, while STAHL gives them very special consideration due to its knowledge about reactions. Nevertheless, both systems are capable of dealing with information about particular reactions, so there is considerable room for interaction. Let us first consider the path from GLAUBER to STAHL. Although in our examples of GLAUBER's operation, we have assumed data like (reacts inputs (hydrogen oxygen) outputs (water)), note that substances like hydrogen, oxygen, and water are not really primitive concepts. Rather, these names represent *classes* of objects that share certain recurring features and enter into similar relations. Given instances of substances such as these, GLAUBER would have no difficulty in formulating these classes and finding patterns such as (reacts inputs (hydrogen oxygen) outputs (water)). In this revised scheme, GLAUBER would generate the patterns (which we originally viewed as data) that STAHL requires as input. Similarly, STAHL can also accept higher level patterns as input, and make inferences about the structures of higher level classes. For instance, given the pattern (reacts inputs (acid base) output (salt)), the system would decide that salts are composed of acids and bases.

⁸We will not discuss the relation between BACON and STAHL, or that between GLAUBER and DALTON, since no obvious interactions present themselves.

Interactions can occur in the other direction as well. STAHL's major outputs are conclusions about the components of substances, and there is nothing to prevent this information from being represented in a form that can be inspected by GLAUBER. For example, suppose that STAHL determined that HCl was composed of H and Cl, while H_2SO_4 was composed of H and SO_4 . These inferences might be represented as (has-components object (HCl) components (H Cl)) and (has-components object (H_2SO_4) components (H SO_4)). Given this information, GLAUBER would create a class containing the substances HCl and H_2SO_4 , which it might then combine with the acid class. This would lead to the pattern (has-components object (acid) components (H acid-components)), which is equivalent to stating that acids have hydrogen as one of their components. Thus, useful knowledge could flow in both directions between these two systems.

STAHL and DALTON

As we have seen, STAHL focuses on determining the components of chemical substances, while DALTON is concerned with the number of particles involved in a reaction and with their internal structure. Thus, STAHL can be viewed as laying the groundwork for a detailed structural model, with DALTON being responsible for finalizing the model. In fact, DALTON employs knowledge about the components of a substance in testing its conservation assumption, and it is natural to assume that this information comes from STAHL. However, since DALTON's outputs are actually more detailed versions of the models produced by STAHL, it is unlikely that any communication from DALTON to STAHL would be useful.

BACON and DALTON

Although we have seen how DALTON's search through the space of molecular models can be altered by theoretical assumptions stated as productions, we have not discussed the origin of the data matched by those rules. For instance, Avogadro's heuristic relied on knowledge of the combining volumes of gases. As the reader will recall, such information can be generated by BACON's common divisor heuristic, and it is interesting to think that BACON's data-driven heuristics could lead to theory-driven behavior in a system such as DALTON. The origin of DALTON's theory-driven heuristics is more difficult to explain, and we have no plausible account for them at present. Since DALTON's outputs are molecular models and BACON's inputs are numerical data, we foresee no major information passing from the former to the latter, though it is conceivable that DALTON's models might lead to predictions of empirical laws which BACON could then test.

Conclusions

In this paper, we have examined four AI programs that address different facets of the discovery process. The details of the individual systems should by now be clear, and we hope the reader has gained some insight into our long-range plans through our discussion of the possibilities for an integrated discovery system. Even within our expanded framework, many important aspects of discovery have been omitted, such as the formulation of mechanistic explanations, and the role of structural analogies [Winston, 1980, Gentner, 1982]. Thus, even our goal of an integrated discovery system involving BACON, GLAUBER, STAHL, and DALTON is limited in some important

respects. However, limiting one's focus of attention is a venerable and useful tradition in the history of science, and there will be ample time to incorporate these additional facets of discovery after we better understand the relations between the four existing systems.

Before closing, we should mention some of the earlier AI research on discovery, and its relation to the programs we have described. BACON is most closely related to Gerwin's [1974] heuristic curve-fitting system, though BACON can deal with more complex functions and employs a different set of heuristics. GLAUBER bears some similarity to Brown's [1973] system for discovering kinship relations, though this focused on eliminating redundancies from a set of facts, rather than on forming abstract laws. GLAUBER is somewhat more closely related to Michalski and Stepp's [1981] conceptual clustering program, which also generates classification schemes based on qualitative data, though the details of the two systems differ. DALTON's search for molecular models can be viewed as a simplified version of DENDRAL's [Feigenbaum, 1971] search for organic compounds to explain mass spectrograms, though again the details of these programs are quite different. We are not aware of any obvious analogy in the AI literature to STAHL's inference rules, so this system appears to incorporate a unique approach to scientific discovery. Thus, while most of our discovery systems are related to earlier work in the area, they also differ in significant ways, and unlike most earlier programs, show potential for being combined into a more complete, integrated theory of discovery. This approach is perhaps most akin to Lenat's [1977] account of mathematical discovery, in which a variety of heuristics interact in complex ways to generate an ever richer set of concepts.

In conclusion, we now have four independent AI discovery systems. The first of these, BACON, focuses on the discovery of quantitative empirical laws, and has been extensively tested in a number of different domains. Of the other systems, GLAUBER is most similar to BACON in that it relies on very general, data-driven heuristics and generates summaries of the qualitative data it observes. STAHL and DALTON incorporate more domain-specific knowledge, since they employ knowledge of reactions, but this does not mean they are limited to the chemical domain. These systems can be viewed as producing simple explanations, since they construct models of the data they receive as input. The three new systems are not nearly such finished products as BACON, and they should be extended in directions that we have already discussed. Once this has been accomplished, we will be ready to attempt combining the programs into an integrated discovery system that involves significant interactions between its components. Whether or not we succeed in this effort, we feel that it will lead us to both a deeper and broader understanding of the process of scientific discovery.

References

- Bradshaw, G., Langley, P., and Simon, H. A. BACON.4: The discovery of intrinsic properties. *Proceedings of the Third National Conference of the Canadian Society for Computational Studies of Intelligence*, 1980, 19-25.
- Brown, J. S. Steps toward automatic theory formation. *Proceedings of the Third International Joint Conference on Artificial Intelligence*, 1973, 20-23.
- Feigenbaum, E. A., Buchanan, B. G., and Lederberg, J. On generality and problem solving: A case study using the DENDRAL program. In *Machine Intelligence 6*, Edinburgh: Edinburgh University Press, 1971.
- Gentner, D. A structure-mapping approach to analogy and metaphor. *Proceedings of the International Conference on Cybernetics and Society*, 1982, 75-79.
- Gerwin, D. G. Information processing, data inferences, and scientific generalization. *Behavioral Science*, 1974, 19, 314-325.
- Langley, P. Data-driven discovery of physical laws. *Cognitive Science*, 1981, 5, 31-54.
- Langley, P., Bradshaw, G., and Simon, H. A. Data-driven and expectation-driven discovery of empirical laws. *Proceedings of the Fourth National Conference of the Canadian Society for Computational Studies of Intelligence*, 1982, 137-143.
- Langley, P., Bradshaw, G., and Simon, H. A. Rediscovering chemistry with the BACON system. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence View*, Palo Alto, CA: Tioga Press, 1983.
- Lenat, D. B. Automated theory formation in mathematics. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977, 833-842.
- Michalski, R. S. and Stepp, R. E. An application of AI techniques to structuring objects into an optimal conceptual hierarchy. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, 1981, 460-465.
- Winston, P. H. Learning and reasoning by analogy. *Communications of the ACM*, 1980, 23, 689-703.

COGNITIVE ECONOMY

In a Fluid Task Environment

Douglas B. Lenat
C.S. Dept., Stanford U.
Stanford, Ca.

Frederick Hayes-Roth
Teknowledge, Inc.
Palo Alto, Ca.

Philip Klahr
The Rand Corporation
Santa Monica, Ca.

ABSTRACT

Intelligent systems can explore only tiny subsets of their potential external and conceptual worlds. To increase their effective capacities, they must develop efficient forms of representation, access, and operation. If forced to survive in a changing task environment, most inferential systems could benefit from *learning* about that environment and their own behavior. For example, they could exploit new schemata or different slot-names to simplify and restructure their knowledge. In this paper we describe how Eurisko (a discovery program) automatically extends its schematized representation of knowledge, by defining appropriate new slots dynamically.

1. Introduction

Computer programs, no less than biological creatures, must perform in an environment: an externally imposed set of demands, pressures, opportunities, and regularities. *Cognitive economy* is the degree to which a program is adapted to its environment, the extent to which its internal capabilities (structures and processes) accurately and efficiently reflect its environmental niche. If the environment changes frequently and radically, then the program can benefit from monitoring those changes and adapting itself to them dynamically. This is cost effective when the penalty for failure to adapt exceeds the sum of the following costs: (i) The continuous overhead for monitoring to detect changes, (ii) The onetime cost of providing a mechanism by which self-modification can occur, and (iii) The occasional expense of employing that mechanism to respond to a change which has been observed. This is expressed as Equation 1, below.

There are two evident directions for exploring this phenomenon:

(i) Make the preceding equation less mushy. Investigate what features of the performer (program), task environment, changes in that environment, etc., make the "monitoring & self-modification" process cost effective. Clearly, in some cases the penalty for non-adaptation is high enough; as examples spanning several time scales consider biological evolution (millenia), the educational system of a culture (years), the business meetings of a corporation (months), the immune system of an organism (hours), and the nervous system of an organism (milliseconds). In other cases, the above equation tips *against* adaptation as being cognitively economical; to date, almost all computer programs (and cognitive models in general) have been constructed to deal with a fixed task, or at least with a fixed task environment [Newell & Simon 1972]. But the magnitude of the phenomena we seek to model continues to grow, and we begin to feel the confines of any single, unchanging model we might hypothesize. In the years ahead, our models --- be they in Lisp, equations, or prose --- must become increasingly responsive. In the conclusions of this paper, we provide some further thoughts on this research direction.

(ii) Operationalize the vague phrase "monitoring & self-modification." In the case of computer models, how might they select (or *discover*) timely new knowledge, new control algorithms, new knowledge representations? What are the difficulties encountered?

This paper focuses on direction (ii). Section 2 assumes that programs can monitor their runtime environment, and it presents techniques by which they can modify themselves to heighten their fitness to it. Section 3 describes how one particular program, Eurisko, dynamically defines useful new slots automatically, at appropriate times. Appendix 1 discusses cognitive economy in existing programs; Appendix 2 presents some of the assumptions we are making in this paper; Appendix 3 illustrates some specific techniques for having programs dynamically monitor their task environment.

(1)	Penalty for not adapting	>	Overhead in monitoring	+	One-time cost of getting self-modification	+	Occasional cost of responding to observed changes
-----	--------------------------------	---	------------------------------	---	--	---	---

Equation 1. When it is cost-effective to have software that models itself, monitors its own execution, thereby notices changes in its runtime environment, and reacts to such changes by modifying itself.

2. Dynamic Self-Modification

Summary: We illustrate various ways in which a program might utilize knowledge gleaned dynamically: selecting (or perhaps even discovering) new data structures, representations, and algorithms, which seem better suited to the current runtime environment, the current user, the current problem, etc.

In order for a program to dynamically modify its own processing, it must examine, integrate and apply knowledge of the task domain, knowledge about programming in general, knowledge about the actual runtime environment, and knowledge gathered from observing its own processing behavior. Appendix 4 sketches ways in which dynamic program performance could be self-monitored. This section draws attention to the possibility of applying such usage data to modifying --- and in the extreme re-synthesizing --- the program.

Tasks can be specified anywhere from a pure "What" (desired input/output behavior) to a pure "How" (precise algorithms, data structures, representations, formats). It has been observed [Feigenbaum 1977] that one of the goals of AI research is to push our ability from the How end toward the What end of that spectrum. This often takes the form of designing a new veryⁿ⁺¹ high level language: i.e., higher than any that hitherto existed, one which takes over more of the program implementation. We can see this as far back as assemblers replacing machine language, and more recently in KRL [Bobrow and Winograd 1977], MRS [Genesereth 1981], LOOPS [Stefik *et al* 1982] and other extensions to Lisp.

Assuming, then, that the user desires merely to specify What, an intelligent language (automatic code synthesizer) must decide How. Upon what can it base its design? Previous efforts have employed knowledge about program transformations [Burstall and Darlington 1977] [Balzer *et al.* 1977], programming techniques and task-specific knowledge [Green and Barstow 1978], and --- if the input language is at all complex --- information to enable the successful translation of the input specification into a sufficiently precise specification of What [Lenat 1975]. Some efforts attempt to guide the decision-making during code synthesis by algorithmic analysis [Kant 1977] or even by aesthetics [Cohen 1979][Kahn 1979] [Lenat 1976]. But all these methods are *static*: they feed off a fixed base of initially-supplied knowledge (heuristics, constraints, suggestions, facts). The user supplies a specification for the desired program (via a formal spec, a dialogue, etc.), and the knowledge base is employed to produce that target.

What might it mean for a program to modify itself dynamically? The simplest kind of adaptive abilities along this line would be to *select* a data structure, a representation, or an algorithm from a set of well-known choices --- and to reserve the right and the ability to change that decision later. One could gather knowledge about the relative strengths and weaknesses of each choice [Kant 1977]. This knowledge could be in the form of rules which ask not only about static data, but which sample the runtime environment of the program as well. See Figure 1.

"If one conjunct appears to be false more often than any other,
Then make it the first one tested in the conjunction"

"If each person will use the program only once, briefly,
Then it is not worth building a model of each such user"

"If a function is used frequently and changed rarely,
Then it should be compiled."

"If items always appear to be accessed in the same order,
Then a sequential data structure is probably adequate for them."

Figure 1: Environment-driven selection rules

Such design questions could in most cases be answered initially (i.e., during program synthesis) but our suggestion of automating the gathering of such data is a further step toward the "What" of artificial intelligence.

There is another possibility, much more exotic than the previous one, which deserves attention: the automated discovery --- in "real time" --- of new types of data structures and representations which would be useful for the program in its current environment.

Defining a new data structure is not as difficult as it may first appear: a data structure can be thought of abstractly in terms of a set of operations one wants to be able to perform on it (e.g., First, Last, Next, and Previous, for lists). The set of desired operations need not be minimal nor concern itself with which operations are most important, etc. Such efficiency considerations are used, of course, in *selecting* an appropriate data structure, and in *implementing* it. The run-time usage of the currently-employed data structure can be sampled, and that can guide the search for an ever more appropriate implementation (e.g., in one usage, it might be very useful for Last to be efficient, and Interlist's ICONCing might be chosen as an implementation; in another environment, it might be useful for Next and Previous to be efficient, and a doubly-linked implementation would be best).

The two points here are (i) the implementation of a data structure may depend upon how it is commonly used in a particular spot in a program, by a particular user, and (ii) a new kind of data structure may be defined abstractly merely by selecting a set of operations; this choice, too, can be made by examining the run-time data as it accumulates (e.g., the final rule in Fig. 1.)

Defining a new *representation* is not quite so neat. In fact, humans have only managed to find a handful of representations to date (graphs, wffs, images, text, etc.) In lieu of any constructive suggestions along that line, we shall focus on *extending* a given representation: This appears to be intermediate in difficulty between selecting a representation and discovering a new one.

2.1. Extending a Schematized Representation

Summary: The Eurisko program extends its schematized representation by defining new types of slots. It uses a very simple grammar for defining new slots from old ones (legal moves), and a corpus of heuristic rules for guiding and constraining that process (plausible moves).

For a schematized representation, "extension" could occur by defining new types of slots. The Eurisko program (an extension of the AM program [Lenat 1976]; for more details see [Lenat 1983a,b]) has this capability, and we shall briefly describe how this mechanism was developed, step by step.

First, we isolated several ways in which new slots are defined from old ones:

STAR (e.g., Ancestor =_{df} Parent*, which means a Parent of a Parent of... of a Parent of me; the general case should be a parent);

UNION (e.g., Parent =_{df} Father $\cup$ Mother);

COMPOSITION (e.g., First-Cousin =_{df} ChildoSiblingoParent; i.e., any child of any sibling of either of my parents);

DIFFERENCE (e.g., Remote-ancestor =_{df} Ancestor - Parent).

These four operators (*, $\cup$, o, -) which define new types of slots are called *slot-combiners*. Other slot-combiners will be mentioned later.

Next, we added to Eurisko's knowledge base a concept for each type of slot, and a concept for each type of slot-combiner. Figure 2 shows some of the Generalizations concept (a slot) and the Star concept (a slot-combiner). Note in particular the way in which Generalizations is defined as Genl: --- i.e., immediate generalizations, *their* immediate generalizations, and so on.

GENERALIZATIONS

ISA: Slot
 SPEC: Extreme-Generalizations, Genl, Proper-Generalizations
 WORTH: 875
 AVG-SIZE: 7
 INVERSE: Specializations
 ANALOGS: Supersets, Relaxations, Enlargements
 SLOT-COMBINER: Star
 BUILT-FROM: Genl (i.e., Immediate-Generalizations)

STAR

ISA: Slot-combiner
 WORTH: 488
 ANALOGS: Repeat, Closure
 COMBINER-FOR: Generalizations, Specializations, Ancestor, ...
 SLOT-TYPE: Star
 DEFN: $\lambda (s) \text{ (SUBST } s \text{ for slot in } [\lambda (c) \text{ (CONS } c \text{ (self (GET } c \text{ slot))})])}$

Figure 2: Parts of the Generalizations and Star concepts

Finally, we modified the way in which slot entries are accessed. To illustrate this, we choose a simple task in mathematics, whose paraphrase is, "What are all the generalizations of the concept 'primes'?" The traditional way in which (GET 'PRIMES 'GENERALIZATIONS) would work is to examine the property list of PRIMES, looking for the attribute GENERALIZATIONS; if found, the entries listed there would be returned. If, as in Fig. 3, there is no such attribute, the call upon GET will return NIL (the empty list).

PRIMES

ISA: SetOfNumbers
 DEFN: $\lambda (n) \text{ (for } i \text{ from 2 to } n-1 \text{ (NOT (DIVIDES } i \text{ } n)))}$
 GENL: Numbers
 SPECIALIZATIONS: OddPrimes, PalindromePrimes
 WORTH: 744
 EXS: (2 3 5 7 11 13 17 19 23 29 31)

Figure 3: Part of the Primes concept. Note that it has an attribute called GENL, which specifies its *immediate* generalization, namely Numbers, but it has no GENERALIZATIONS attribute.

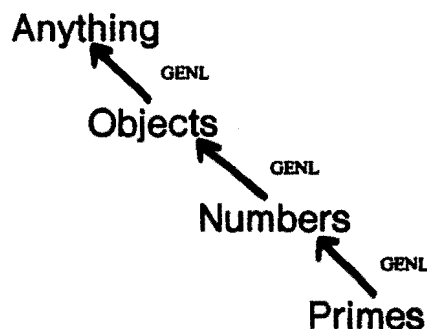


Figure 4: Network of concepts from Primes upwards, along Genl links.

We modified the way in which any retrieval request of the form (GET C F) operates. In case the F attribute of C has no entries (or doesn't exist), Eurisko examines the definition of F and --- if one exists --- tries to use it to compute the entries that could legally be stored on the F attribute of C. More precisely, before quitting we try to (GET F 'DEFN), and if that succeeds we apply it to C. Let's continue the example of (GET 'PRIMES 'GENERALIZATIONS). As we see in Fig. 3 above, there are no entries recorded on the GENERALIZATIONS slot of the frame for Primes. So GET now calls itself recursively; our original call is replaced by ((GET 'GENERALIZATIONS 'DEFN) 'PRIMES). But as Fig. 2 shows, there is no slot labelled DEFN on the concept for GENERALIZATIONS. So we recur one more time. By now our call has become

((GET 'DEFN 'DEFN) 'GENERALIZATIONS) 'PRIMES).
 I.e., get the expression stored in the Defn slot of the Defn concept; call it Δ. Δ should exist, and should be a lambda-expression (a function) which takes the name of a slot S as argument, and (when evaluated) computes a function which tells how to read slot S for any concept. Eurisko applies Δ to Generalizations, and that yields as a result a new function, a lambda-expression, call it Γ. Γ tells how to find generalizations of any concept. In particular, when Eurisko applies Γ to Primes, the result is a list of the legal entries for the Generalizations slot of Primes.

DEFN

ISA: Slot
 SPEC: Necessary-Defn,
 Suff-Defn,
 Operational-Defn
 WORTH: 975
 AVG-SIZE: 1.3
 ANALOGS: Statement, Algorithm
 DEFN: λ (x) (APPLY*
 (GET x 'Slot-Combiner) 'Defn)
 (GET x 'Built-From))
 GENL: Predicate
 MyISA: EuriskoUnit
 MyCreator: Lenat

Figure 5: Part of the DEFN concept

Luckily, DEFN does have a DEFN slot, as we see from Figure 5, so we end the recursion (rather than entering an infinite loop at this point). Applying the entry stored there (in the DEFN slot of DEFN) to the argument "GENERALIZATIONS," we see our original call transformed into

```
(GET (GET 'GENERALIZATIONS 'SLOT-COMBINER) 'DEFN)
  (GET 'GENERALIZATIONS 'BUILT-FROM))
  PRIMES]
```

We see from Fig. 2 that the slot-combiner of Generalizations is "Star," and the argument (old slot) from which it is built is "Genl." So the entire call shrinks into

```
((GET 'STAR 'DEFN) 'GENL) 'PRIMES).
```

The Star concept has an entry for its Defn slot, as we see in Figure 2. This turns the preceding call into

```
((λ (c) (AND c (CONS c (self (GET c 'GENL)))))) 'PRIMES).
```

In evaluating this expression, the first thing called for is

```
(GET 'PRIMES 'GENL).
```

which (see Figure 3) is just Numbers. That is, the immediate generalization of Primes is Numbers. The expression then calls itself recursively on Numbers, seeking *their* immediate generalization:

```
(GET 'NUMBERS 'GENL)
```

which we see from Figure 4 is simply Objects. The next recursive call asks for (GET 'OBJECTS 'GENL), which is ANYTHING. Finally, the next recursive call terminates when it is discovered that ANYTHING has no GENL (no immediate generalization.) See Fig. 4. The list CONSTRUCTED and finally returned is therefore (PRIMES NUMBERS OBJECTS ANYTHING). These four items are indeed the valid entries for the GENERALIZATIONS slot of PRIMES, according to the distributed definition of GENERALIZATIONS as GENL*.

Notationally there is no distinction between slots which are "primitive" (values actually stored as attributes on a property list) and slots which are "virtual" (values must be computed using the slot's definition). A heuristic might refer to the Generalizations of Primes without knowing, or caring, whether that initiated a single access or a dizzy chase. Thus we are decoupling, disentangling knowledge (the rules) from the specific implementation of its representation (the choice of which slots are primitive and which are virtual.) This is required because knowledge remains invariant under changes in the environment which effect the worth of various representation schemes. Donald Michie recognized this long ago with his memo functions.

To define a new kind of slot, now, Eurisko merely specifies one of the then-known slot-combiners and a list of the old pre-existing slots from which it is built. Thus a naive user sat down and defined a new slot, by creating a new concept (calling it "DG"), filling its Slot-combiner slot with the entry "Difference", and filling its "Built-from" slot with the arguments "Generalizations Genl." This thereby defined DG as a new kind of slot, one which returned all generalizations of a concept except its immediate generalizations; the call (GET 'PRIMES 'DG) then made sense, and returned the list (PRIMES OBJECTS ANYTHING).

After carrying out the above steps, to explicitly represent slots and slot-combiners in Eurisko, an interesting benefit accrued. With no extension at all, Eurisko was then able to define new kinds of slot-combiners. For instance, it proposed one which took two old slotnames as arguments, f and g, and defined a new slot which was f*ogof*. This could be extremely useful (e.g., in database searches: see Fiksel and Bower [1976]). In particular the crucial slot "Examples" is defined as Spec*oExsoSpec* (Exs are immediate examples, much as Genl's are immediate generalizations). The important slot-combiner PLUS (in the sense of regular expressions, not arithmetic) was also discovered by Eurisko. Eurisko defined it as the difference between f* and f. Later, Eurisko decided that Genl* was more worthwhile than Genl*, and (partly for this reason) PLUS itself ultimately achieved a higher Worth than STAR.

2.2. CACHING to reclaim lost efficiency

Summary: "Caching" the results of computations can dramatically improve the performance of many programs. Reasoning can be brought to bear to decide whether to cache, if so what to remember, and (later) whether or not to ignore the cached value and recompute it. Caching reclaims the efficiency that could otherwise be sacrificed by the highly indirect representation-interpreting scheme described in the previous section.

The reader may object that Eurisko's scheme for indirectly interpreting the definition of each slot is a horribly inefficient way to operate, since the slots' definitions change so rarely. It seems wasteful to squander cpu time recomputing (GET 'PRIMES 'GENERALIZATIONS) each time, and it seems absurd to

recompute (GET 'GENERALIZATIONS 'DEFN) each time. After at most one or two repetitions, we would want our program to have "learned" an efficient way to get an answer.

Indeed, a general tool for cognitive economy --- software caching --- can be exploited to reclaim most of this efficiency. By a "general" tool, we mean one which is applicable in almost all environments, not merely those being focused upon in this paper (fluid ones.)

Immediately upon computing (GET c f), Eurisko stores the computed value v in the f slot of concept c (*a la* Michie's memo functions.) Thus, the next time (GET c f) is called, it will find v stored right there, and return with it immediately. After the first time (GET 'PRIMES 'GENERALIZATIONS) was called, the list (PRIMES NUMBERS OBJECTS ANYTHING) was stored on the GENERALIZATIONS slot of PRIMES. Recall that Eurisko had to recursively call (GET 'GENERALIZATIONS 'DEFN), so the λ -expression that *that* call returned would be stored on the DEFN slot of the concept called GENERALIZATIONS. When we later call (GET 'DUCK 'GENERALIZATIONS), that call will run much faster, since the definition of the Generalizations slot has already been computed and stored away in the "proper place." The definitions of slots are very slowly --- if ever --- changing, hence the recomputation of Defn of Generalizations is quite a rare event. Caching that value *must* be cost-effective in the long run.

In general, we see that caching a value for slot f of concept c is most applicable when the value can be expected to vary quite infrequently. In Eurisko, this gives us the flexibility to redefine a slot's definition if we wish, but (since this change of representation will be rare) the program will run just about as efficiently as if that capability were absent. This is analogous to compiling: so long as the definition of a function doesn't change too often, it's undisputedly worthwhile to compile it. The caching of values for frames' DEFN and GENERALIZATION slots is not in any way special; the value of any virtual slot can be cached.

We saw earlier that notationally there was no need to distinguish an access of a primitively-stored slot from an access of a virtual, computed slot. For instance, a heuristic could refer to "Generalizations of PRIMES" without knowing or caring whether that initiated a single access or a long rippling search. Now we see that once the value is computed and cached away, there may be no telling it from a primitive one. Thus, with but a small one-time cost, our program runs as quickly as if all slots were primitive. The inefficiency introduced by the interpretive slot-combiner scheme appears to have been eliminated.

Of course, the values that Eurisko caches away may become stale and incorrect after a while. When doing a GET, therefore, when a cached value is encountered, Eurisko must decide whether to accept it, or else to ignore it, recompute it, and store the new value there. One possible answer to this "updating problem" is the following conservative one: whenever a value for a slot of a concept changes, discard any cached values that might *possibly* have depended on it. Arbitrary amounts of reasoning may be brought to bear to decide which values needn't be discarded to maintain consistency of the knowledge base, but of course if too much time is spent in this pursuit the efficiency advantages of caching disappear. Eurisko carries on AM's tradition of living with contradictions, with a plausible but not "correct" knowledge base. Namely, it has some heuristics (see Fig. 7) for deciding whether to ignore or accept any cache it encounters during a GET operation.

We added some extra arguments to GET, parameters which describe a cost/benefit curve: for each amount of resources expended and each degree of precision of the result, this curve specifies precisely how acceptable that resource consumption / accuracy of solution pair is. One extreme of this is to provide an ironclad limit on resources and say "Do the best you can within these time/space/... limits." Another extreme (most common in present-day programs) is to specify an ironclad goal criterion ("Find an answer which is at least *this* good, no matter how long it takes to get.") We are calling attention to the space in between the two extremes.

To bring us back from the lofty heights of generality, let's see in more detail how we got Eurisko to do this. First, we linearized this space: we picked a few significant parameters of it, and defined a description to be merely a list of values for these parameters. When a call on GET was executed, and a cached value was encountered, a formula relating these extra arguments to GET would then determine whether to accept that cache, or to ignore it and recompute the value. Eurisko's parameters (extra arguments to GET) are: cpu time to expend, space to expend, whether the user can be asked about this, how fresh the cache must be, minimum amount of resources which must have been expended at the time the cache was written.

So (GET PRIMES GENERALIZATIONS 200 40 NIL 3 0) would allow any cached value to be accepted if it appeared that recomputing would take more than 200 milliseconds, or would use up more than 40 list cells, or if the value had been put there less than three Cycles ago. (In Eurisko, a "Cycle" is the time it takes to work on the top-rated task on the top-rated agenda.) Otherwise, the cache would be ignored, a newer value would be computed, and it would be stored away. With it would also be recorded the following information: (i) the fact that it was a cached value, not a primitive one, (ii) how long it took to compute, (iii) how many list cells it used up in computing this value, (iv) the current Cycle (number of tasks worked on so far). The above call on GET might result in the following value being stored on the GENERALIZATIONS slot of PRIMES:

(*cache* (PRIMES NUMBERS OBJECTS ANYTHING) 54 6 9).

2.2.1. Storing and Updating Cached Values

The caching process involves storage and updating. For both of these aspects, we can discuss details of when, why, how, and what. The decisions that arise are made with the guidance of heuristic rules, many of which are illustrated below in Figs. 6 and 7. Those lists have omitted many of the very general "common-sense" heuristics, and those which deal with details of Lisp programming. Additionally, they present the rules in English, and freely use informal expressions such as "recently"; in the actual Interlisp encoding they are of necessity operational, hence more precise (and concomitantly a bit less valid).

Just as each heuristic is not always relevant, so to the corpus applies only to some (although we hope many) AI programs, running on large uni- (or few-) processing computers, running in an environment such as Interlisp's, and with a human user watching and interacting. There would be other rules for other machine architectures and languages and man-machine interfaces. In other words, one could say that we have only partially specified the IF-part (left hand sides) of the following heuristic rules. An alternative would be to build up an entire knowledge base of heuristics just to determine the applicability (the domains of relevance) of the following (as a function of machine and language.)

When to cache:

Every time you have a call to a lower order function.
You called a function and it took a long time to return.
The value returned is the same as last time.
Some function F is called with the same value x often.
The amount of time to recompute the value would be high.
The cached values can be used as data to induce upon.
Time is dearer than space.
The value is not so critical you'd always favor recomputing.
This number is high: $\frac{(\text{freq. of usage}) \times (\text{avg. cost to recompute})}{(\text{freq. of change})}$

Where:

There should be a canonical storage place to facilitate lookup. Instead of an assemblage database (with no structure) we prefer to add structure: namely, slots (including domain-specific ones)

How:

Given a request to compute $F(x)$, point to the canonical place(s) where cached value might be; if it's there, return it; if it's not there, compute it and store it there for next time.
 Caller may constrain you
 Discarding any cache results only in slowdowns, not errors.

What:

Store the returned value
 Store a copy of the call that got this value
 Store important intermediate subgoals worked on to get the value
 Store a description of what situation would make this value stale
 If no cache is stored, store a footnote explaining why

Figure 6: PRINCIPLES FOR STORING CACHES

When:

The current request would have produced a different answer.
Usually: ...would have produced a *better* answer; e.g.:
 The current request specifies more resources can be brought to bear.
Most frequently: There is no cached answer there, yet.
 Either x or $\text{Defn}(F)$ has changed since $F(x)$ was last cached.

How:

Choose: discard old value, or keep it around to induce or back up.
 Use demons to flag that caches are out of date.
 Just because a cache is stale, don't recompute it (just erase it).
 Keep track of dependencies of one cached value upon another.
 This at least enables timely discarding; at best, intelligent updating.

Figure 7: PRINCIPLES FOR UPDATING CACHES

2.3. The Spectrum of Data Reductions

Caching, as we have described it, is merely the lowest-level of a tier of data reduction processes, techniques for compiling hindsight into usable, more efficient, more procedural forms.

We often refer differently to "caching" depending upon what it is that's being retained: open-ended, inductive searches can be condensed in hindsight (i.e. cached) into heuristics, deductive searches can be cached into much less branchy algorithms, subroutines can be cached into tables of frequently called arguments and resultant values, and variable quantities have their value cached simply by the process of binding. To see this, consider the progression of ever more sophisticated tasks which programs perform [see Fig. 8]: accessing a datum, calculating a value from a

fast algorithm, deducing a result by searching for a solution or proof in a well-defined space, inducing a result by a very open-ended search process. Often there is the opportunity to take experiences at one of these levels of sophistication and abstract them into an entity which is one level lower, yet which appears to contain almost the same power.

Consider first the way in which we reduce induction. After much experience with inducing, we produce a heuristic rule~ which enables the same inferencing to be done in a much more controlled space, in a much more deductive way. Heuristics reduce discovery tasks to mere (!) symbolic manipulation within a well-defined system (as in AM [Lenat 1976]).

Deduction is reduced similarly. By many painful deductions, we gain knowledge of the space we are searching, e.g. by deriving many theorems about it. Those theorems can then be used to vastly simplify our search. Ultimately, the process becomes efficient calculation. A short, optimal algorithm may be found, e.g., which replaces a search for a solution (this has recently occurred with the "*n queens*" chess problem.)

Now we see that our value-caching process is the next step in this progression. It reduces a calculation (e.g., to traverse a particular access path to gather all Ancestors of a node) by a lookup, by a single access of a datum in memory.

All the data reduction processes in the progression are environment-independent techniques for increasing cognitive economy. Often, an apparent loss due to a specific expressive feature can be rectified by applying some of these kinds of methods. In our case, we opted for interpretive defining of slots, but regained efficiency by caching those slot definitions.

2.4. Contrasts with Psychological Ideas of Economy

When semantic networks first appeared in cognitive psychology, they led to the idea that function should follow form. In particular, since taxonomic knowledge structures were hierarchical, inference processes would presumably follow the same inter-category network paths each time the corresponding relation needed testing. It was for this reason that Collins & Quillian [1969][1972] conjectured that the time required to verify semantic relations would correlate with distance in a corresponding "economical" semantic network. Thus, they showed that people ordinarily verified "a canary is a bird" faster than they could verify either of the two more remote relationships, "a canary is an animal" or "a canary has wings."

Subsequent research, however, showed that the needs of particular retrieval experiences, not *a priori* categorizations, determined which inference paths humans would follow and the associated response times. A simple counterexample to the presumed hierarchical retrieval schemes was found by Rips, Shoben and Smith [1973]. They showed that people access a familiar but so-called "indirect" relation such as "a dog is an animal" much faster than the supposedly immediate but less frequent relation "a

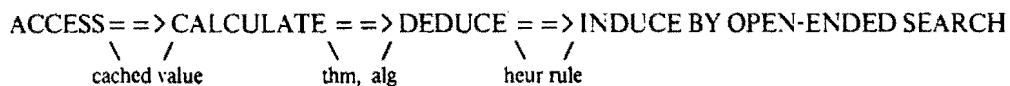


Figure 8: Progressively more sophisticated caching: of values, of algorithms, of heuristics.

dog is a mammal. Using experimental materials and methods, Hayes-Roth & Hayes-Roth [1975] found that human memory seemed plastic and adaptive in many of the same ways we have proposed for intelligent machine programs. Their results led them to "an adaptive model of memory, in which all learned relations are represented directly, with strength proportional to experience. There is also good evidence for redundancy in the network, with multiple routes connecting nodes representing particular pairs of concepts" [1975, p. 519].

In another study, Hayes-Roth and Hayes-Roth (1977) found some evidence that people understand and remember English texts directly in terms of high-level lexical concepts, rather than exclusively as low-level or primitive concepts. They argued that cognitive processes would presumably benefit if they could build directly upon the words or other meaningful units of familiar language. Any system that interpreted and stored data only in terms of primitive units, as is often assumed in language understanding theories, would encounter many slower and costlier searches. Both the empirical evidence and theoretical principles support the idea that human language understanding exploits many kinds of caching.

In short, we have chosen the term "caching" to refer to these types of redundant storage. Caching, to capture and exploit repetition and regularity, will benefit both humans and machines of limited processing capabilities.

3. When to Define a New Kind of Slot

Summary: We have discussed how a new slot might be defined; we consider now how the Eurisko program does decide when and how to define a new kind of slot.

Our basic reply to *When?* is that a slot is defined only when a heuristic rule explicitly suggest it as being useful; that is, heuristics are used as plausible move generators (as opposed to their typical use [Nilsson 1972] as implausible move pruners). Our basic reply to *How?* is that the heuristic which suggests a new slot must provide (constraints for) its definition.

This merely pushes the problem back one layer, namely to the way in which the heuristics know when and how to define new slots. But now we are talking the language of AM and Eurisko, and we can provide examples and general characterizations of the slot-defining heuristics used in Eurisko. Since each type of slot is represented internally as a full-fledged (schematized) concept, the new slots defined can eventually be evaluated (and changed or discarded if they are not utile). Each heuristic is also represented as a full-fledged concept, and its worth can be determined based on the worths of the slots it suggests: thus each slot-suggesting heuristic is occasionally modified (or even discarded). Let's look at some cases:

Sometimes the new type of slot is first proposed purely for aesthetic, analogy-completing, or exploratory reasons. One such heuristic is the following (note that "I" is the identity mapping):

"If R and S are partial inverses of each other, Then it's plausible to define and investigate the following: RoS, SoR. RoS - I, SoR - I"

There are several noneffective slots provided for the above heuristic in Eurisko, such as its reason (in this case, "aesthetic"), its average running time (7 seconds), which other rule(s) created it originally (IsolateDifferences), instances of its use, pointers to slightly more general and more specific heuristic rules, etc.

Note that the heuristic applies when R and S are mathematical functions, heuristics, or slot definitions. Let's examine an example of the latter case, which is of concern to us. This occurred when R was the slot "Generalizations" and S was the slot "Specializations." The heuristic rule concluded that it was aesthetic to define the specializations of the generalizations of a concept (what we would

call "siblings"), and the generalizations of the specializations of a concept (what we would call "inlaws"). Eventually, the Siblings slot is seen to be much more useful than the Inlaws, which is discarded.

The new kind of slot-combiner defined in the previous section [$\lambda(f,g) \text{ fogof}$], was derived purely from symmetry (using a rule similar to this one), and then it was noticed that some of the slots which are most useful in the system (e.g., Isas, Examples) share that common form.

There are other ways in which new types of slots are defined by Eurisko. The definition of a new slot is often based soundly upon need -- or the absence of need. For example,

"If many concepts have a large number of entries for their F slot, Then consider creating some new specializations of F."

To make this rule effective (executable), Eurisko monitors the sizes of slots on its units. When applied, the rule leads Eurisko to create several new slots, each having fewer entries on the average than F had, to cover the original F slot. E.g., after noting that the Examples slot was heavily used, it would be plausible to consider creating new slots like Extreme-examples and Typical-examples. An actual case where this rule was used was in the design of three-dimensional VLSI circuits: many units (representing chips) had enormous entries for their Terminals slot; Eurisko fired this rule and produced several new, more specialized slots; one useful such one was XOR-InputTerminals (its value is defined to be a list of sets of terminals, each set having the property that precisely one element can be an input terminal of the device).

A slightly more specialized heuristic says:

"If you frequently perform 'Remove x from f(x)', Then it's plausible to define g as $f - I$ "

Here, I is the identity, so $g(x)$ is simply $f(x)$ with x removed. This led to the definition of the proper ancestor slot (Genl-(c) -- c) from the simpler Genl (immediate generalizations) slot. It also led Eurisko to define the new slot-combiner PLUS from STAR. We envision how it might lead to the definition of many common, useful mathematical functions, including (although Eurisko doesn't actually do this) operations on matrices with main diagonals removed.

The above examples have tried to illustrate the manner in which heuristic rules can generate timely new kinds of slots. A very important point is that the rules are not specific to defining slots; they apply equally well to defining new heuristics, new mathematical functions, new lab procedures, and so on. One of the bedrock philosophies of the Eurisko effort is that a single large core of commonsense knowledge can adequately guide discovery processes operating on all these levels.

4. Conclusions

If forced to survive in a changing world, most inferential systems could benefit from *learning* about their task environment and their own behavior. For example, they can exploit new schemata or different slot-names to simplify and restructure their knowledge. Self-modification would be *cognitively economical*.

If forced to explore large search spaces with some repetitive regularity, a cognitive system could benefit by employing *caching* to save partial results. We described a variety of techniques to implement caching, and explained how caching specific results is one of a spectrum of methods (algorithms, heuristics) that can make trade-offs among precision, speed, certainty, and generality.

Since most environments contain much that is extraneous to the direct solution of a task, a problem solver in such worlds can gain cognitive economy by *expectation-filtering* (see Appendix 4.1.) In

general, intelligent systems need to exploit their knowledge about *typicality* to reduce their cognitive load and increase their attention to important data. In many situations, we believe that expectations can both reduce processing requirements for handling ordinary events as well as simplify the identification of surprising events.

If the tasks are complex, and the resources allotted vary dramatically in magnitude, then *multiple levels of abstraction* (MLA) becomes an *economical* representation (see Appendix 4.2.) In many simple environments, the advantages of MLA are minimal and may not even justify their costs. Thus, routine tasks may warrant the kind of knowledge compilation that would convert an initial, expressive MLA knowledge-base into a fast, integrated, and largely uninterpretable code. Conversely, as task complexity and variability increase, MLA increasingly provides a basis for intelligent and rapid restructuring.

4.1. The Need for Further Research

In the years to come, AI programs will employ greatly expanded knowledge bases and, as a consequence, they will explore increasingly open-ended problem spaces. Already, a few existing systems show signs of having more potentially interesting things to do than they have resources to pursue. In the past decades of intelligent systems research and development, several design concepts have emerged in response to contemporary needs for creating ever larger knowledge bases. For example, many researchers proposed multiple levels of abstraction and automatic property inheritance as keystones of efficiency or "cognitive economy." We believe that the value of such mechanisms derives largely from their usefulness in describing initial knowledge bases. Once an intelligent system begins to explore the consequences of its knowledge and to solve novel problems in a dynamic environment, it needs to adapt its knowledge to achieve faster and more profitable retrievals.

A theory of cognitive economy should explain under what conditions (and why) knowledge needs to be adapted, and should prescribe systematically how to do it. In this paper, focusing upon fluid environments, we have tried to motivate and lay the groundwork for such a theory. We presented Eurisko's automatic defining of plausible, useful new slot types. Many of the design heuristics we mentioned embody some gross features of an eventual theory and suggest numerous paths for future research.

We have tried to show what cognitive economy *is* and *is not*. It *does not* consist of a set of static knowledge-base design principles, such as those proposing taxonomic concept structures with automatic property inheritance. Rather, cognitive economy is a feature of those intelligent systems that learn to solve problems efficiently and consequently realize more of their lifetime potential. As knowledge bases expand and basic software obstacles are overcome, AI systems will increasingly address the same question facing intelligent humans: "What would I most like to accomplish next, and how can I do that economically?" As has always held true with human beings, the answer for AI systems increasingly involves some type of learning.

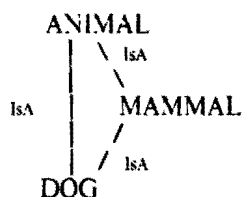


Figure 9. Although the IsA link from Dog to Animal is redundant, it is worth having.

Appendix I. Cognitive Econ. in Existing Programs

To date, most programs, including our own AI behemoths, have *not* had to cope with runtime environments that changed significantly. Referring back to Equation 1 (in Section 1), the costs for monitoring and building in self-modification abilities have been prohibitive, and the need for adaptiveness has been minimal. The environments have been *complex* but not *changing*. To be "fit" in such worlds, programs have had to employ sophisticated architectures, knowledge bases, control strategies, representations, and expressive languages in which they are specified; but programs have *not* needed to reorganize themselves dynamically, to adaptively alter those sophisticated design components. Even in those situations in which this flexibility must be required in the long run, it has been avoided, because (i) self-modification capabilities are tough to build in: many programs have simple self-reprogramming as their sole task, and (ii) monitoring is an expensive overhead in both space (detailed record-keeping) and time (finding regularities in such histories). This raises the issue of efficiency:

Efficiency

When we build a new, large AI program, we often find ourselves caught in the following tradeoff: On the one hand, we want to develop our initial systems quickly and painlessly, since they are experimental and ephemeral. We want to be able to modify them later with little or no redesign or reprogramming manually. On the other hand, we want our systems to run as efficiently as possible, to minimize the demands for cpu time and primary and secondary memory. We are torn between *expressibility* and *efficiency*.

Many AI researchers *cum* language designers have focused on *expressibility*, the problem of rapidly constructing a working experimental vehicle. This objective is apparent in the goals of EMYCIN [Feigenbaum 1977], KLONE [Brachman 1978], KRL [Bobrow and Winograd 1977], PSI [Green 1977], RITA [Anderson and Gillogly 1976], ROSIE [Waterman et al. 1979], and SAFE [Balzer et al. 1977]. The results are some superlative software (such as the systems just mentioned), and also exemplified by Interlisp's DWIM, File, and Break packages. Four fundamental techniques utilized in highly *expressive* programs are: (i) reliance upon a very-high-level language, (ii) planning and reasoning at multiple levels of abstraction, (iii) inference by pattern-directed knowledge sources, and (iv) minimal, nonredundant representation, as in a canonical generalization/specialization hierarchy.

Both within the body of this paper (Caching, Section 2.2) and within these appendices (Expectation-Filtering and Levels of Abstraction, Appen. 4.1 and 4.2) we present various techniques which do not sacrifice expressibility, yet enable programs to (semi-)automatically improve themselves and thus increase their productivity. One point of this paper is to show that the second goal of AI programming, short-term runtime efficiency, is not sacrificed by these techniques.

The basic source of power is the ability to predict the way that the program will be used in the future, and to tailor the program to expedite such uses. This approach is sometimes used *manually*, as when a program is quickly coded in an expressive but inefficient form, used heavily, and then recoded in a much more optimized form. Thus, e.g., Ray Carhart spent 1977 rewriting DENDRAL (streamlining its CONGEN module) for a minicomputer.

The traditional approach to program optimization has assumed that static analysis of programmer's expectations (whether implicit in his code or explicit in his formal assertions) can identify the significant optimization opportunities. Three types of methods for analyzing program descriptions in this way include: (i) analyzing program flow and structure [Knuth 1968] [Dijkstra 1976], (ii) designing data structures to be appropriate, and (iii) compiling (as in the FORTRAN H compiler) and optimizing transformations of the program (as in [Darlington and Burstall 1973 1977] or [Balzer et al. 1977].)

In a changeable environment, we advocate the use of methods more *dynamic* than these. Rather than improving the static description of a program, we propose that the program modify its own structure to adapt it to its operational environment. Some earlier automatic programming systems [Darlington and Burstall 1973] [Lenat 1975] [Low 1974] [Kant 1977] [Barstow 1977] were designed to improve programs' efficiency, and many of their ideas have found their way into the techniques we now describe. Those systems had program construction, transformation, and optimization as their primary task; we are proposing general methods to provide self-improving abilities to other kinds of AI programs (understanders, theory formers, expert simulators, planners, etc.).

One valuable source of prediction comes from the program's experiences as it runs. If a program can "learn" from its experiences, it might try applying each piece of newly acquired knowledge to the task of specializing itself, modifying itself to run more efficiently in the current runtime environment. We believe that a program's "intelligence" can be increased in this way; that is, by increasing its ability to acquire appropriate knowledge, to organize that knowledge, and to refine the conditions under which that knowledge is recognized to be applicable. For any fixed quantity of manpower resources we expend, there is a limit to the size/complexity of programs which can be successfully implemented. This barrier might be overcome by programs which are self-extending, which actively do things to enlarge their input domain, their capabilities.

Notice that representing a corpus of knowledge as a minimal (canonical) generalization/specialization hierarchy, with interpretatively-computed inheritance, may or may not be cognitively economical: this technique favors expressibility, compaction of representation, and efficiency in updating and altering the knowledge base, but at the expense of performance. It is true that a dog is a mammal, and a mammal is an animal, and from those two we could compute that a dog is an animal (see Fig. 9), but if the same arc is being repeatedly recomputed then it is more cognitively economical to store one redundant arc. Studies by ourselves and others [Hayes-Roth and Hayes-Roth 1974] [Rips et al. 1975] indicate just such redundancies being created and strengthened by repetitions.

Obviously, the economy of specific representations and related inference methods depends heavily on underlying machine architectures and costs. We assume that intelligent systems aim chiefly to produce useful results (e.g., novel hypotheses) as rapidly as possible. In short, they should search cleverly and fast. Thus, *a priori* notions of aesthetics or design economy have little bearing on cognitive economy. Massive redundancies in storage, processors, and inference processes seem more directly relevant. Furthermore, we suppose that, like people, intelligent software should eliminate slow, interpretive searches for previously solved problems (especially if the same problems are repeated often). To this end, cognitive economy may be achieved through dynamic compilation of action sequences. Such compiling would reduce the time-cost to effect the corresponding behavior; on the other hand, this efficiency would appear to be gained only at the cost of interpretability, interruptibility, and modifiability. Those are desirable attributes iff a program potentially needs continual changing. In typical situations, both efficiency of compiled forms and accessibility to declarative forms are intermittently needed. These different needs point again to the economic benefits of maintaining simultaneously alternative and redundant representations.

When It Pays to Adapt

We can abstractly characterize the situations in which an AI program would find it cognitively economical to monitor and adapt itself: the resources available for specific tasks vary significantly; or the specific tasks have (at least locally) some similarities, some structure which enables them to be solved quickly in the same particular way. Here are some examples:

The load average of a time-sharing system fluctuates with time of day, but users demand that their speech understanding system always provides the best guess it can in real time.

A hospital simulation program can run at excruciating levels of detail, and it lives on a dedicated machine. But there's been a bad accident nearby and within one minute the police must have an estimate of how many emergency cases the hospital can accept.

A program is given an enormous amount of resources to prepare for each task, but little resources between the time a task is presented and an answer is called for. It may be cost effective for the program to spend some of its preparation (inter-task) time trying to *discover* new algorithms, representations, etc. We are struck by this in nature (evolution), though as yet just glimpse it in AI programs (such as Eurisko).

A system is sold to several customers, each with a distinct but complex task environment in which it will live. It then becomes cost effective to spend some resources streamlining the program (either manually or automatically) to fit that user. In terms of equation 1, the environment changes only once, so no continuous monitoring and modification is performed: the onetime cost of the modification can be amortized throughout the lifetime of the program (i.e., the penalty for non-specialization is integrated over time.) The well-known SYSGEN generation of tailored operating systems is a simple example of this.

While maintaining a file system, the program notices the patterns in each user's accesses to his files, and of the community as a whole, and occasionally selects a new data structure for some part of the system to make it run more efficiently. See [Low 1976].

Appendix 2. The Assumptions

Summary: The claims we make elsewhere in this paper presuppose that "intelligence" can be adequately modelled as heuristic search, guided by pattern-directed rules, implementable on a uniprocessor.

Every scientific theory is constructed in a rich context of surrounding theories, methods, and standards determining which experiments are reasonable ones to perform and which point of view to take when interpreting the results --- in short, a *paradigm*. We feel it useful to articulate the "core" of one's paradigm: the assumptions one's theory rests upon. For that reason, this appendix is devoted to a presentation of our assumptions, including: a model for intelligence (Appendix 2.1), a model for how intelligent computer programs may be organized (Appendix 2.2), and a model of the characteristics of present computing engines (Appendix 2.3). Briefly:

(i) We continually face searches in more or less immense spaces; intelligence is the ability to bring *appropriate* knowledge to bear, to speed up such searching. Increasing intelligence then comprises increasing knowledge, improving its organization, and refining the conditions for its applicability. How are these new discoveries made? Empirical induction (generalizing from experimental and other observations), analogy, and the criticism of existing theory all lead to new conjectures, new possibilities.

(ii) Intelligent systems can make the applicability of their knowledge explicit by representing that knowledge as condition-action rules IF *situation*, THEN *appropriate reaction*. Such pattern-directed inference systems (PDIS) can benefit from a schema representation (frame, unit, being, script, etc.), because this adds structure which the rule descriptions can exploit.

(iii) Current computing technology presents us with limited cycles, cheap storage, and expensive knowledge acquisition.

Appendix 2.1.: A Model of Intelligence

Summary: Since "intelligence" depends upon bringing relevant knowledge to bear, it can be increased not only by adding new knowledge, but also by better organizing the existing knowledge.

Many human cognitive activities, including most of those commonly referred to as "requiring intelligence," can be cast as searches, as explorations through a search space, meandering toward some goal. We call a problem-solver "more intelligent" if it can work efficiently toward a solution even in the face of an immense search space and an ill-defined goal. Somehow, it is imposing more structure on the problem, and using that to search more effectively. How might it do this? According to our model of human intelligence, it accomplishes the task by bringing knowledge to bear, knowledge not supplied explicitly in the problem statement. This knowledge can be about problem-solving in general (e.g., how to recognize and break cultural blocks [Adams 1974]) or about the problem domain specifically (e.g., which groups of chemical atoms can usually be treated as aggregate superatoms [Feigenbaum 1977]).

This implies that a problem solver can become more effective by (i) increasing its knowledge, (ii) better organizing its knowledge, and (iii) refining its criteria for deciding when various pieces of knowledge are applicable. In terms of production (If-Then) rules, these correspond to adding new rules, modifying the data structure in which rules are held, and tuning the conditions (If parts) of existing rules.

One very basic assumption we are making is that of *continuity*: the problem-solving environment does not change its character too radically, too rapidly. That is, our environments are *fluid* but not gaseous. Let's see why this assumption is required. If the program abstracts from its experiences a plausible heuristic *H*, it may be able to analyze past scenarios to verify that possessing and obeying *H* would definitely have improved its performance (e.g., led to the current state of knowledge quicker); but its only justification for believing that *H* will help it in the *future* is the empirical evidence for the continuity of the environment. Learning can be translated effectively into action only when the structure of the environment has not altered too greatly.

A related assumption is that of *irreversible knowledge accretion*. Our body of facts and observations continually grows and only occasionally shrinks. Certainly the abandonment of a whole system of thought is a rare event indeed (see [Kuhn 1972]), whereas the adoption of new systems of thought is part of the standard educational process.

New knowledge is discovered by specializing some broadly applicable but weak principles, or by generalizing particular experiences. The latter includes abstraction (condense some experiences into a heuristic which would have greatly aided us in the past if only we'd had it) and analogy (draw parallels to other existing facts and heuristics, and also to the *parhs* which led to their discovery.) One sometimes "spirals in" [Lakatos 1976] toward precision and competence (see Fig. 10).

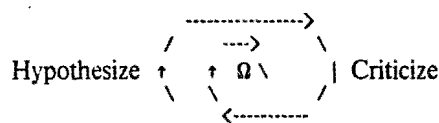


Figure 10. The Lakatosian loop of criticism and improvement of hypotheses, spiralling ever closer to truth and understanding.

Appendix 2.2.: Model of Intelligent Program Organization

Summary: Since we want our AI programs to access, reason about, and expand their own knowledge bases, it is useful to represent such knowledge in a clean, modular form (e.g., employing any one of the current schematized representations.)

The preceding remarks about intelligent problem solving apply equally to "hardware" and "wetware" alike. To be intelligent, computer programs must ultimately grapple with the tasks of knowledge acquisition, representation, and refinement. We cannot provide an absolute answer to how they should be organized, but we can posit some design constraints which have proven useful so far.

A very general heuristic in AI programming is the following:

If your program is going to modify its own *X*,
Then make *X* as separable, clean, and explicit as possible.

In our case, *X* can be instantiated as "knowledge," or as "applicability conditions for each piece of knowledge." In this case, the heuristic advises us to represent our knowledge in a separate, clean, explicit form, say as knowledge modules having some fixed internal structure, and also advises us to keep the applicability conditions for each knowledge module separate from the rest of the knowledge it contains.

This naturally leads us to a pattern-directed inference system, in which knowledge is broken into separate modules, each with an explicit set of relevancy tests [Waterman and Hayes-Roth 1978]. Such systems arising in Pittsburgh and London may champion syntactic purity, while those arising in California may tend toward the baroque, but such variations should not obscure their common source of power. The PDIS architect breaks his program's knowledge into a set of condition-action production rules.

Having a clean representation for rules means having a clean, precise, elegant language in which to express them. By structuring the conceptual knowledge of the system, by partitioning each module's knowledge into several categories, a rule can condense an entire cumbersome description into a simple reference (often a single pointer). The popular schematized representations of knowledge (scripts for episodes, frames for static situations, beings for specialists, units for everything) enable a rule like "If there are no known methods specific to finding new examples of prime numbers, then..." to have its condition coded as a simple null-test on the "To-get" subplot of the "Examples" slot of the schema called "Prime Numbers." By a judicious choice of slot types and subplot types, the system builder can reduce most triggering conditions to such quick checks on the state of various (subplots of) slots of schemata.

Additional knowledge can be brought to bear to locate efficiently all the rules which *might* have their conditions satisfied in a given situation, and also to decide which rules to execute (obey) among those whose IF- parts have triggered (been satisfied).

Appendix 2.3.: A Model of (Present) Computers

Summary: Since we are considering the problem of building computer models of intelligent behavior, many of our assumptions deal with the characteristics of present-day computers. They are the symbol manipulators we use, but were not designed for that general purpose.

The foremost quality of computing machines, as we know them today, is their limited "inferential bandwidth." Although they provide virtually (*sic*) limitless storage capacity, these machines can process only a small fraction of this information in a reasonable amount of time. Two primary reasons for this disparity between storage and generation capacities include: (1) knowledge acquisition proceeds slowly and expensively; and (2) inference processes depend on operations that ordinarily require cycles of uniprocessor systems.

The first problem places a *cultural* limitation on the development of intelligent machines. This development is restrained by our inability to express what we would like our machines to do or to convert these expressions into effective machine interpretable codes.

The second problem reflects a *technological* constriction of our potential. The so-called von Neumann bottleneck, caused by machine architectures that sequentially interpret program steps through a single logical processor, strongly motivates concern for programs that can minimize unnecessary computations.

In some future environment, perhaps neither of these problems will persist. In the meantime, however, we assume that two of the best ways to increase AI system productivity, derived from these current limitations, will be to expedite knowledge acquisition and to avoid unnecessary computing (for instance, by opting to "store & recall" rather than "forget & recompute").

Appendix 3: Monitoring the Task Environment

Summary: The body of this paper has amply illustrated how a program might profit from knowledge it gathered dynamically. This suggests that rather than working on the given problem exclusively, a program may be better off to expend some time learning: learning about the specific problem, its broader domain, or problem-solving in general. Note this suggestion would encompass traditional education (being told), empirical research (observing), and theoretical exploration (predicting). While such "very high-level" problem-solving strategies typify human problem-solvers (especially those of us who have rationalized spending twenty or more years "in school"), very few programs to date have employed any of these forms of learning to improve their operation.

We discussed in Section 1 the "environmental fluidity" conditions under which it could be cost effective for an intelligent system to be able to improve its performance dynamically. To accomplish this, a program must first be able to monitor its processing, i.e., to sense, record, and analyze its representations, data structures, algorithms, inference procedures, and the way they are currently being relied upon. It can then use this gathered knowledge to modify or redesign itself to perform more efficiently.

Some programs (and wish-lists for program capabilities) have included learning more about the problem being worked on. For instance, McCarthy's advice-taker and Balzer's dialogue system (to get airline reservation programs written) would be told, Gelernter's theorem-prover (employing diagrams as analogic models) could observe, Lenat's AM could observe and predict. In this paper, however, we are stressing learning about the ways the program is being currently employed.

Learning by Being Told

There are many situations in which a user could provide advice to a program:

(a) The user might want to convey to the program some task-specific item: "Integration by parts won't work on this problem"; "This is a very hard problem." This can easily be input as part of the user's statement of the problem for the system to work on; e.g., the user might designate some methods as being preferred or constrained, he might estimate a resource allotment, etc.

(b) Another type of advice will apply permanently to the nature of the program's functioning (e.g., "Answers, even if uncertain, must be produced in real time;" "Hard problems will be starred.") Such knowledge can be reflected permanently in the design of the program, in its code.

(c) But there is an intermediate level of advice, some information the user knows will concern the next several tasks to be submitted --- not just the present one and not all future ones: "The next four problems will be supplied in order of expected difficulty;" "Several of these problems share common assumptions and raise related issues;" "The user for the next entire session will be a physician;" "Until I find out why F34 returned NIL, I'll be in debugging mode, not production mode." Such advice is rarely communicable even to AI programs.

We believe this type of intermediate-level advice would improve programs if they had the opportunity to accept and use it. In general, we would like to advise the program which capabilities to consider and which to ignore during a prolonged period of solving closely related problems. A fixed (static) approach, for example, would be to define a few modes (e.g., Debug, Demo, Production, Input,...) and permanently build into the code all changes required to specialize the system for each mode. A flexible (dynamic) approach would provide a larger language in which the user could describe his current mode of usage of the program (e.g., by typing "USAGE of KNOWLEDGE-BASE will be INCREASED VERY-MUCH"). The program would translate such statements into changes in its data structures, algorithms, explanations, core management strategies, etc. In the current example, heavy usage of the knowledge base might cause the program to swap out its compiler, optimizers, and other code, enabling it to hold many more knowledge chunks at a time in core. The ability to hold more knowledge in memory would directly affect the rate and probability of productive inferences. (Similar effects have been shown for humans: [B. Hayes-Roth 1979, *J. Cog Sci*])

Learning by Observing

Spending a little time/space reconfiguring to reflect the current state of the world (the nature of the recent and succeeding inputs) seems worthwhile; the problem is how to know what that abnormal nature of the environment is. In the last subsection, we discussed the case where the user explicitly tells the program. But this is often impossible (as when the user is unaware of an impending irregularity, or when the "user" is Nature), and almost always a nuisance (learning the language in which to express such advice, remembering to do it, taking the time to do it).

The next step seems to be to allow the program to *infer* such facts about its run-time environment from observations. What kinds of self-monitoring can be usefully employed?

The method which comes to mind first, perhaps, is to save a complete record of everything that ever happened to this program: all the inputs it received (when, from whom), function calls and accesses it made internally, paths it chose/rejected (and why), and outputs it generated. The program could in principle induce, from such an exhaustive history-list, patterns such as "very light usage of compiler;" "SQRT is always called on the result of SUM-OF-SQUARES."

The space costs of such an archive, and the temporal cost of the procedures to infer useful patterns, are surely prohibitive. The more economical approach is to analyze in advance (and change slowly throughout the program's lifetime [Samuel 63]) certain features we would wish to look for in a complete history record, and then to have the program record just enough dynamically to gather and maintain those features. Some of these will be domain-independent and commonly useful: Frequency of calls on various functions, requests for subtasks, etc.; Patterns in the arguments to calls on certain functions and inputs to the program; Patterns in the sequence of calls on functions; Patterns in functions' results (e.g., F34 always seems to return a User-Name). Some of the features will be domain-dependent specializations of the preceding features ("Average degree of input equation" is a special "pattern in the input.")

Learning by Predicting

The theoretician and the engineer share a powerful technique: manipulating a model of a system to generate predictions and identify those possible outcomes that would be desirable. This is a third route to learning. How might programs produce it? What sorts of theories, or models, can be employed to generate new information?

Models of the program's users can be valuable. This includes maintaining and extending profiles of individual users and whole groups. When a user opens a dialogue with Eurisko with the word "Let", it is a safe bet that he's a mathematician. Later, when he

uses the word "function," his membership in that group makes it easy to determine which meaning he has in mind (instead of, say, what a physiologist or a computer scientist or sex therapist might mean by "function.") When the program wishes to mention the square root of -1, it uses its MATHEMATICIAN model to choose an appropriate notation ("i" for mathematicians instead of, say, "j" for engineers). Models can be arranged in a hierarchy, with inheritance of features from more general groups to their subsets, a lattice ultimately terminating in models of individuals. User inputs would be recognized by various models, which would then hypothesize membership in that group (or explicate some future conditions which would confirm or deny such inclusion). Once triggered in this way, the user's apparent groups (and its generalizations) would actively filter his inputs and outputs, could modulate the program's actions (e.g., by changing the worth of the "Proof" concept, which in turn would change the priority rating of tasks on an AM-like agenda and thus alter the program's choice of what to work on next.) New user models can be created either top-down (e.g., when a model has many diverse instances or when it's proven historically to be very valuable, then try to split it into several specializations, each having a share of the general model's instances) or bottom-up (i.e., notice that a group of individuals share a set of common characteristics, and create a new user group out of them).

The preceding kind of prediction is based on a theory of user types, embodied in a set of "typical representative" models. The same source of power can be used in many ways: stereotyping of events (as in scripts), stereotyping of static complexes (as in frames), stereotyping of communities (as in actors, beings, and contract nets). In all these cases, the source of power lies in quickly recognizing probable membership in (applicability of) some schema, after which all the rest of the schema (plus knowledge about all its generalizations) is presumed to be relevant. Programs might build up and utilize scenarios of how problems are solved typically in their domain. The building-up could be top-down via specialization or bottom-up via generalization, or even "sideways" via analogy. The models would be used for prediction (recognize relevant models and activate them, have them hypothesize probable future situations). Together, the models would form a theory of how problems are solved in that domain.

Appendix 4: Cognitive Economy in Any Environment

Summary: A coherent theory of cognitive economy must characterize the sources of power of its methods, and describe ways in which they specialize for environments having various properties. While the body of this paper concentrated upon cognitive economy in fluid environments, we hope to see such a comprehensive theory develop in the future. For that reason, we shall present (briefly, here in an appendix) a few techniques for enhancing cognitive economy in arbitrary environments. We have already described in great detail (Section 2.2) the caching of intermediate results (intelligent redundancy) and will now treat two additional methods: expectation-simplified processing (intelligent focus of attention) and multiple levels of abstraction (intelligent knowledge structuring.)

Appendix 4.1.: Expectation-Simplified Processing

Summary: For efficiency's sake, an intelligent system should be willing and able to add new facts, but should be eager to add surprising new facts. Surprises can only be noticed by contrast with expectations, so an intelligent system should maintain a context of expectations and filter incoming observations against that. Furthermore, expectations and surprises can aid an intelligent system in comparing its model and processing of the domain to the real world. Through such monitoring, discrepancies may be found and diagnosed, leading to changes in the model making it more consistent with observed behavior. Our discussion here centers on the importance of using such expectations to focus and filter intelligent processing.

The world bombards our senses with data, much more than we can process in detail in real time; yet we can't live in a hundred times real time. We survive by ignoring most of that data, or, more precisely, by knowing (almost immediately deciding) what can be ignored. We maintain a set of expectations, against which the incoming torrent is matched. Almost all of it will match those expectations, and we then need merely process the unexpected inputs, the surprising observations. We reserve our computing for those opportunities which promise us genuine new facts, rather than reconfirmation of known ones.

Such concentration on the abnormal is a natural side-effect of being guided by expectations --- albeit usually stereotyped ones --- about situations, events, and people. One is able to zip through most of what is encountered, match it immediately to the active schema, and move on. The only processing which is slow and tricky is that dealing with the remaining anomalies, exceptions, puzzles.

The more general position is that each slot indicates not merely a default, but also an indication of how important it is to confirm that value. Some slots will have meta-comments like "I'm worth computing or verifying", or just the opposite. For example, in our "Room" frame, we have a default Ceiling, but in everyday life we know to not squander time checking that each room we're in has a ceiling. Similarly for emergency exit signs: they usually exist, and we rarely feel it's cost-effective to confirm that. We have in both situations a very fast algorithm we can apply if ever we are in a situation in which we need that information. For ceilings, it's $\lambda()$ (look upwards); for exit signs, it's "look for red on white lettering." While this may seem to be a much more complex program to execute, recall that if we truly need to run it --- say in a sudden power failure --- we expect those lit signs to be one of the few things we could still see.

The policy for whether, when, and how to check a default slot value can be arbitrarily complex. The program may be designed to reason at great length about how much time to expend reconfirming the expected. It may reason about the task of reasoning about how much time to spend, and so on.

Several increases in efficiency are realizable through expectation-filtering:

(a). Perceptual set: Expectations allow you to see what you expect with less effort. For emergency exit signs, when we need to find them, we know to look for horizontal and vertical red lines; this is 99% effective. When we look at our red LED watch and have a perceptual set to read the time, we of course would not mistake the image for that of the EXIT.

(b). Surprise: If we observe Y when we expected X, our attention is captured. In one way, sensitivity to data inconsistent with expectations has been heightened. We saw in (a), above, that reliance on frames will dull our chance of noticing a surprising datum (since we will tend to interpret any data as consistent with the current frame). This point (b) says that if we do notice it, then it will have a significant impact. Carefully building up an expectation and then abruptly shattering it with a jarring surprise is the archetypal template for a good joke. As Arthur Koestler observes, the jarring of conflicting matrices of thought is what powers art (ah!), science (aha!) and humor (haha!).

(c). Defer: By relying upon expectations, upon defaults, we can usually defer processing to find the true value in the current case forever. At least, when we do finally have to make a particular perception, it may at that time be easier to make. When the ceiling is falling, it's easier to see; when all the lights go out except for the EXIT signs, it's easier to see them. Generalizing, we may state that it's worth a little computing to reason out whether (and maybe even how long) a slot-filling can be ignored.

(d). Predict and Prepare: This includes the whole continuum we discussed in the Caching section: bind a variable, store the triple $(F.x.F(x))$, store enough information so that $F(x)$ could be recomputed very quickly, store information which would make

$F(x')$ more easily computable (for x' similar to x , and F' similar to F), store away a partial model of the world with good and bad features tagged. Notice that moving along this continuum, programs grow more flexible, more descriptive (less scalar), more dynamic. In all cases, the motivation for doing this storage is to (i) speed up the recognition of (the need to do) similar computations in the future, and (ii) speed up or even eliminate those computations when you decide they are relevant and you want the values they would return. These are the two uses of scripts, but also of cached constants (e.g., caching the value returned by OWNED-BY, a function from CARS to PEOPLE). A better example is *user models*: they range from a single bit (E.g., a NOVICE/PRO toggle), to a few numbers (e.g., in PARRY), to a dynamic concept-modelling scheme (e.g., in Eurisko).

In the interests of cognitive economy, we conclude that programs should build up expectations, monitor them, and change their models as needed. A natural mechanism for doing this is the use of pattern directed knowledge modules (PDMs), which can react to changing situations, to sudden disconfirmations of predictions. These can be used as triggers or demons; e.g., if you hear someone shout "Fire!" in a theater, that takes precedence over Jane Fonda. The second use of PDMs arises when an expectation is not met: the program then also has some chance of modifying the culprit rule(s) that made that false prediction (and/or adding new rule(s) which would make the right one.)

Appendix 4.2.: Levels of Abstraction

Summary: *An intelligent system's domain model is typically an approximation of reality. It is often the case that knowledge exists at several levels of detail and abstraction. The amount of detail required in intelligent processing depends on the problem-solving context. By appropriately structuring knowledge, a problem-solver can take advantage of abstraction by reasoning at the level appropriate to the given situation. We argue here that abstraction is crucial for intelligent systems which must process over large knowledge bases or within widely-varying time constraints.*

In many real-life problem solving situations, the "goal" is not a single, precise answer. Rather, there is a cost-benefit tradeoff between the accuracy of the solution and the amount of resources consumed in producing it. Computer programs which are designed only to return exact answers lie at one extreme end of this curve, and rarely near the optimal point (of diminishing returns). As McCarthy and others have observed, when a coffee cup spills, it's not optimal to carefully integrate the equations of hydrodynamics to decide whether or not to jump away from the table.

This notion of a cost-benefit curve presupposes a kind of "continuity" in the environment, as we discussed in Appendix 2.1. Namely, truncated search is assumed to be better than random selection of a legal move. Almost paradoxically, if the environment is mercurial, then the optimal point on the tradeoff will cause our program to be very expressive.

Cognitive systems which must cope with the world have an omnipresent need to approximate reality --- a need which is well met by maintaining multiple representations of the world at various levels of abstraction; i.e., an entry in the knowledge base has all its generalizations also stored explicitly, redundantly.

Multiple levels of abstraction ("MLA") are useful for quickly approximating reality. In Section 1, we described a situation in which a hospital simulation program might be required upon occasion to provide as good an estimate as possible to a query within a given (short) time period. Another example is the spilling of a coffee cup: the decision of whether to jump back, of what items will get soaked, is made very rapidly, heuristically, not by integrating the hydrodynamic differential equations.

MLA also facilitate analogizing: they can make that process easier, faster, and occasionally even more valid. Analogies can be recognized when two distinct concepts coincide at some abstract

level. Conversely, if (corresponding slots of) two apparently related concepts (such as two homonyms) do not share related structure at higher levels of abstraction, that suggests that an analogy between them would be superficial and lacking in power.

Multiple levels of abstraction cost a certain amount to implement, both initially (man-hours of programming) and as the program runs (cpu time spent maintaining the redundancy, choosing which level to work on next, etc.) For problems of all sizes, this feature aids in the expression of new problems to the program, since they may be input at whatever level(s) seem appropriate to the user. Thus multiple levels of abstraction will always add to expressiveness, and will either detract or add to efficiency depending upon how small or large the task is. MLA is very useful when the program's resources are curtailed or, even better, varied dramatically in magnitude from run to run (a program which was *always* limited to a 15-second execution time would be designed specifically for its task, but a program which might get anywhere from 2 seconds to 5 hours for the same simulation question could better benefit from multiple levels of abstraction.) MLA contributes more to efficiency as the quotient (task size)/(resource allotment) increases in size, and as its variance increases.

There is a more general result here: the methods which humans find useful for easy expression of problems may be useful to programs attacking very large problems. For instance, consider the decision about whether to do inference by a system of pattern-invoked experts, or, instead, to preprocess the triggering conditions into a large discrimination network. The former approach is more expressive, the latter more efficient --- at least for small problems. For very large tasks, the need to remain flexible looms very large. It is then more economical to retain pattern-directed control; the compilation of the patterns is so complex the whole procedure would have to be redone frequently during the course of execution, if we opted for that alternative instead. A case in point is the HARPY speech understanding program [Lowerre & Reddy 1979]: whenever the grammar changes --- at any level --- the entire speech recognition network must be recompiled. HEARSAY II [Lesser and Erman 1977], while slower in performance, has sufficient flexibility (from several independent pattern-directed knowledge sources) to obviate such monolithic integrative recompilation delays.

Acknowledgements

We wish to thank Dave Barstow, John Seely Brown, Bruce Buchanan, Ed Feigenbaum, Cordell Green, Russ Greiner, Greg Harris, Barbara Hayes-Roth, Allen Newell, Herbert Simon, Mark Stefik, and Don Waterman for several discussions which have left their mark in this paper.

References

- Adams, J. L., *Conceptual Blockbusting*, W.H. Freeman: San Francisco, 1974.
- Anderson, R. H. and J. J. Gillogly, "Rand intelligent terminal agent (RITA): design philosophy," R-1809-ARPA, The Rand Corporation, Santa Monica, Calif., 1976.
- Balzer, R., N. Goldman, and D. Wile, "Informality in program specifications," *Proc. of the 5th Int. Joint Conf. on Artificial Intelligence*, Cambridge, Mass., 1977, 389-397.
- Barstow, D., "A knowledge-based system for automatic program construction," *Proc. of the 5th Int. Joint Conf. on Artificial Intelligence*, Cambridge, Mass., 1977, 382-388.
- Berliner, H. J., "Chess as problem solving: the developments of a tactics analyzer," Ph.D. Dissertation, Carnegie-Mellon

- University, Pittsburgh, 1974.
- Bobrow, D. G. and B. Raphael, "New programming languages for artificial intelligence research," *Computing Surveys*, 6, 1974, 153-174.
- Bobrow, D. G. and T. Winograd, "An overview of KRL, a knowledge representation language," *Cognitive Science*, 1, 1977, 3-46.
- Brachman, R., "Theoretical studies in natural language understanding", BBN Report 3888, September, 1978.
- Burstall, R. M. and J. Darlington, "A transformation system for developing recursive programs," *J. ACM*, 24, 1977, 44-67.
- Cohen, H., "What is an Image?", *Proc. Sixth International Joint Conference on Artificial Intelligence*, Tokyo, August, 1979, 1028-51.
- Collins, A. M. and M. R. Quillian, "Retrieval time from semantic memory," *J. Verbal Learning and Verbal Beh.* 8, 1969, 240-247.
- Collins, A. M. and M. R. Quillian, "How to make a language user," in (E. Tulving and D. Donaldson, eds.) *Organization of Memory*, Academic Press: New York, 1972.
- Conrad, C., "Cognitive economy in semantic memory," *J. Experimental Psychology*, 92, 1972, 149-154.
- Darlington, J. and R. M. Burstall, "A system which automatically improves programs," *Proc. of the 3th Int. Joint Conf. on Artificial Intelligence*, Stanford, Calif., 1973, 479-485.
- Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall: Englewood Cliffs, N.J., 1976.
- Feigenbaum, E. A., "The art of artificial intelligence: themes and case studies of knowledge engineering," *Proc. of the 5th Int. Joint Conf. on Artificial Intelligence*, Cambridge, Mass., 1977, 1014-1029.
- Fikes, R. E., P. E. Hart, and N. J. Nilsson, "Learning and executing generalized robot plans," *J. Artificial Intelligence*, 3, 1972, 251-288.
- Fiksel, J. R. and G. H. Bower, "Question answering by a semantic network of parallel automata," *J. Math. Psych.*, 13, 1976, 1-45.
- Gelernter, H., "Realization of a geometry-theorem proving machine," in (E. A. Feigenbaum and Julian Feldman, eds.), *Computers and Thought*, McGraw-Hill: New York, 1963, 134-152.
- Green, C., "A summary of the PSI program synthesis system," *Proc. of the 5th IJCAI*, Cambridge, Mass., 1977, 380-381.
- Green, C. and D. Barstow, "On program synthesis knowledge," *J. Artificial Intelligence*, 10, 1978, 241-279.
- Hayes-Roth, B. and F. Hayes-Roth, "Plasticity in memorial networks," *J. Verbal Learning and Verbal Behavior*, 14, 1975, 506-522.
- Hayes-Roth, B. and F. Hayes-Roth, "The prominence of lexical information in memory representations of meaning," *J. Verbal Learning and Verbal Beh.*, 16, 1977, 119-136.
- Hayes-Roth, B. and C. Walker, "Configural effects in human memory," *J. Cognitive Science*, 1979.
- Hayes-Roth, F., P. Klahr, J. Burge, and D. J. Mostow, "Machine methods for acquiring, learning, and applying knowledge," P-6241, The Rand Corporation, Santa Monica, 1978.
- Hayes-Roth, F. and V. R. Lesser, "Focus of attention in the Hearsay-II speech understanding system," *Proc. of the 5th IJCAI*, Cambridge, Mass., 1977, 27-35.
- Kahn, K., "Making Aesthetic Choices", *Proc. of the 6th Int. Joint Conf. on Artificial Intelligence*, Tokyo, 1979, 448-50.
- Kant, E., "The selection of efficient implementations for a high level language," *Proc. ACM SIGART-SIGPLAN Symp. on Artificial Intelligence and Programming Languages*, SIGART Newsletter #64, 1977, 140-146.
- Klahr, P., "Planning techniques for rule selection in deductive question-answering," in (D. A. Waterman and F. Hayes-Roth, eds.), *Pattern-Directed Inference Systems*, Academic Press: New York, 1978.
- Knuth, D., *Fundamental Algorithms (The Art of Computer Programming, Volume I)*, Addison Wesley, 1968.
- Kuhn, T., *The Structure of Scientific Revolutions*, 1972.
- Lakatos, I., *Proofs and Refutations*, Cambridge University Press: Cambridge, 1976.
- Lenat, D. B., "Beings: knowledge as interacting experts," *Proc. of the 4th Int. Joint Conf. on Artificial Intelligence*, Tbilisi, USSR, 1975, 126-133.
- Lenat, D. B., "AM: an artificial intelligence approach to discovery in mathematics as heuristic search," Memo AIM-286, Stanford AI Lab, 1976.
- Lenat, D. B., "The Nature of Heuristics II and III", *J. of Artificial Intelligence*, 20, 1, March, 1983.
- Lesser, V. R. and L. D. Erman, "A retrospective view of the Hearsay-II architecture," *Proc. of the 5th Int. Joint Conf. on Artificial Intelligence*, Cambridge, Mass., 1977, 790-800.
- Low, J., "Automatic coding: choice of data structures," Memo AIM-242, Stanford University, 1974.
- Lowerre, B. and D. R. Reddy, "The Harpy system," in (W. Lea, ed.), *Recent Trends in Speech Recognition*, Prentice-Hall: Englewood Cliffs, N. J., 1980.
- McCarthy, J., "The advice taker," in (M. Minsky, ed.), *Semantic Information Processing*, MIT Press: Cambridge, 1968.
- Mostow, J. and F. Hayes-Roth, "Machine-aided heuristic programming: a paradigm for knowledge engineering," N-1007-NSF, The Rand Corporation, Santa Monica, Calif., 1979.
- Newell, A., J. C. Shaw, and H. A. Simon, "Chess-playing programs and the problem of complexity," in (Feigenbaum and Feldman, eds.) *Computers and Thought*, McGraw-Hill: New York, 1963, 39-70.
- Newell, A. and H. A. Simon, *Human Problem Solving*, Prentice-Hall: Englewood Cliffs, N.J., 1972.
- Nilsson, N., *Problem Solving Methods in Artificial Intelligence*, 1972.
- Rich, E., "Building and Exploiting User Models", *Proc. of the 6th Intl. Joint Conf. on Artificial Intelligence*, Tokyo, 1979, 720.
- Rips, L. J., E. J. Shoben, and E. E. Smith, "Semantic distance and the verification of semantic relations," *J. Verbal Learning and Verbal Behavior*, 12, 1973, 1-20.
- Samuel, A., "Some Studies on Machine Learning of Checkers," in (E. A. Feigenbaum and Julian Feldman, eds.), *Computers and Thought*, McGraw-Hill: New York, 1963.
- Sandewall, E., private communication, August 22, 1979.
- Stefik, M., "An examination of a frame-structured representation system," Memo HPP-78-13, Stanford University, 1978.
- Waterman, D. A., R. H. Anderson, F. Hayes-Roth, P. Klahr, G. R. Martins, and S. Rosenschein, "Design of a rule-oriented system for implementing expertise," The Rand Corporation, Santa Monica, 1979.
- Waterman, D. A. and F. Hayes-Roth, *Pattern-Directed Inference Systems*, Academic Press: New York, 1978.

THE ROLE OF EXPERIMENTATION IN THEORY FORMATION

Thomas G. Dietterich

Bruce G. Buchanan

Department of Computer Science
Stanford University
Stanford, CA 94305

Abstract

This paper presents an analysis of the role of experimentation in theory formation. Experimentation serves three purposes: (a) hypothesis testing, (b) gathering of new data to constrain the theory generator, and (c) manipulation of the external system to reveal its structure. A theory formation system, EG, is described that employs experimentation and observation techniques to develop a theory of the UNIX file system and executive-level file commands. This theory formation task is more complex than previous efforts, and the goal of the project is to determine which existing theory formation methods are applicable and what new methods need to be developed. Previous techniques are reviewed, and none of them are found to be applicable. A new technique, based on controlled experimentation, is described, and a hypothetical trace of EG's execution is presented.

Key terms: Theory formation, generate-and-test, controlled experimentation, exploratory experiments, observation experiments, hypothesis-test experiments, credit-assignment, new terms.

1 Introduction

Despite major advances in recent years, Artificial Intelligence models of learning and scientific theory formation are still primitive. This paper describes the EG project—whose goal is to investigate theory formation in a domain that bears a closer resemblance to ordinary scientific domains than previous AI theory formation tasks. We have chosen the task of constructing a theory of the UNIX¹ operating system, because it provides ample opportunities for experimentation under program control. In this paper, we describe this theory formation task and then review previous approaches to theory formation including generate-and-test, version spaces, Meta-DENDRAL, and AM/Eurisko. From this review, we conclude that heuristic search techniques similar to those used in Eurisko are applicable to this domain with modifications to increase the role of experimentation. Three basic functions of experimentation are (a) to test hypotheses, (b) to gather more data for guiding the theory formation process, and (c) to explore the various states of the system being modelled. The paper concludes with a description of the EG program and a hypothetical example of its operation.

1.1 Description of the theory formation task

The main goal of the EG project is to investigate a large-scale theory formation problem to determine which existing AI techniques can be scaled-up and what new techniques need to be developed. The

theory formation task that EG must solve is to develop a theory of the UNIX) operating system by designing, performing, and analyzing experiments with UNIX executive commands. The theory to be developed has four components (see Figure 1): input/output syntax, executive-level commands, system calls, and file-system data structures.

-
- Input and output syntax. (E.g., `Incommand ::= 1n source dest`)
 - Abstract programs for each executive command. (E.g., `1n command`, `cp command`)
 - System calls. (E.g., `Create-file`, `Delete-file`, `Change-working-directory`)
 - File system data structures. (E.g., file directories, protection codes, current working directory)

Figure 1: Four components of EG's theory of UNIX

Input-output syntax includes the syntax of strings typed by the user (commands, file names, arguments, switches, and so on) as well as the syntax of strings printed by the computer (directory listings, prompt characters, error messages, and so on).

The command component of the theory involves describing the behavior of each executive command as an abstract program. There are two basic classes of commands: observation commands, which simply display information about the current state of the system, and manipulation commands, which perform some action on the file system. Manipulation commands typically involve such steps as checking the prerequisites of the action to be performed, printing error messages, requesting confirmation for dangerous actions, and, of course, invoking the appropriate system calls to perform the actions.

The final two components of the theory are closely coupled. The file system is described as a set of mappings and data structures that implement those mappings. The system calls are then defined as operations on these data structures. For example, there is a many-to-one mapping from file names to files, and this mapping is implemented as a table in a directory file. The `Create-file` system call adds a new entry to this table (as well as writing a file on the disk).

This theory is substantially different from any theories developed by previous learning programs in the inter-connectedness of the concepts in the theory. "Theories" formed by Lenat's AM [1976] and Eurisko [1982] programs, for example, are simpler in that they investigate systems of nearly-independent concepts. Each concept can

¹UNIX is a trademark of Bell Laboratories.

be defined and evaluated without modifying the definition of any other concepts. In EG, however, a change in the definition of a data structure, for example, can have far-reaching effects on the definitions of system calls, commands, and even I/O syntax.

In defining this particular task, we were guided by an analogy with the way people learn about operating systems. People are rarely taught the details of an operating system. Instead, they read manuals and then conduct experiments. An examination of any operating systems manual will reveal many incompletenesses. For the UNIX manuals we have inspected [Kernighan and McIlroy, 1976; Lomuto and Lomuto, 1983], the syntax and main behavior of each command are well-specified. However, the underlying structure of the file system and the exact behavior of executive commands in unusual situations are very incompletely described. By analogy, EG starts with a fairly complete model of the I/O syntax of UNIX. It is given partial models of the more common executive commands and very little information about the underlying data structures and system calls. It also has a body of general knowledge about computers (e.g., data structures for implementing a many-to-one mapping) and knowledge about the kinds of actions that one would normally want to perform on files and similar data objects.

1.2 Review of previous work in theory formation

Given this knowledge about operating systems in general and UNIX in particular, there is still an immense space of possible theories. We believe that clever experimentation is required in order to search this space efficiently. Before describing the approach that EG employs, let us review existing techniques for searching large hypothesis spaces.

The most naive approach to searching any space is to generate all candidate solutions and test them to see if they are indeed solutions. In this domain, this approach corresponds to generating all theories of UNIX consistent with the starting knowledge base and testing them through experiments. This approach is grossly inefficient, because it does not use the results of hypothesis testing to improve the search as it progresses.

Mitchell's [1978] candidate-elimination algorithm is a technique that completely incorporates information obtained from each test into the generator of possible hypotheses. We believe that the version space approach would not work in this domain, however, because the theory of UNIX is too large to be usefully viewed as a single "concept." The UNIX theory has many nearly-independent subcomponents (e.g., individual commands) that give the space of possible theories a very branching partial order. Consequently, the candidate-elimination algorithm would not perform well in this space.

The approach employed in Meta-DENDRAL appears initially to be applicable. Meta-DENDRAL is given a set of molecules (with known structure) and their corresponding mass spectra, and its goal is to develop a theory that predicts how molecules will break apart inside the mass spectrometer. It employs what has been termed the "single-representation trick" [Dietterich, et al., 1982] to search the space of possible cleavage rules. First, each data point in each mass spectrum is translated into a set of specific cleavage rules that could have caused that data point to appear. Then syntactic pattern matching methods are applied to generalize these cleavage rules to obtain a small set of general cleavage rules that explain most of the data. An analogous approach in EG's domain would start by gathering training instances of commands and their behaviors. Then the set of maximally-specific UNIX theories that could explain these behaviors would be computed. Finally, syntactic methods would be applied to generalize these theories to obtain a single theory that explained all of the behavior. Unfortunately, this approach is not feasible because the set of

maximally-specific theories that explain a given command is too large to compute. Furthermore, it is not obvious which data should be gathered initially to guide the theory formation process. A more dynamic approach appears to be called for.

The AM and Eurisko programs employ a dynamic, opportunistic approach to discovery tasks. It is worthwhile investigating how analogous methods might be applied to EG's task. AM and Eurisko are discovery systems that search a space of possible constructs looking for interesting ones. A direct application of Eurisko to the operating system domain would involve giving Eurisko general knowledge about operating systems along with heuristics for the design of good operating systems and then letting it run to see what interesting systems it discovers. To convert Eurisko into a theory formation system we need to replace the "interestingness" heuristics with techniques that force Eurisko to generate theories that explain the observed behavior of UNIX. This is roughly the approach we are taking. In place of "interestingness" heuristics, EG plans and carries out observation experiments to reveal important structure in UNIX. This structure is then incorporated into the growing theory of the operating system.

1.3 The role of experiments in learning and theory formation

Experimentation (or active instance selection) has been employed in several learning and theory formation systems (see [Buchanan, Mitchell, Smith, and Johnson, 1977]). It provides two basic functions: testing hypotheses and gathering further information to constrain the hypothesis generator.

The simplest learning methods rely only upon the ability of an experiment to test the truth of some proposition and return a YES/NO answer. In pure hypothesis testing, this answer merely indicates that the proposition (or supporting theory) is correct or incorrect. However, it is possible to use this information to help guide the theory generator. For example, as we mentioned above, Mitchell's [1978] version space approach guarantees that the generator will generate only theories that are consistent with the data. Hence, every YES/NO answer returned from an experiment is completely incorporated into the generator of theories. In this role, experiments can be viewed as providers of new data from which the theory is developed. Even this role has not been developed much in existing learning systems. Substantial speedups in learning can be obtained by clever use of experiments to "split the hypothesis space in half" (see [Mitchell, 1978]).

It is possible to exploit YES/NO answers further through credit assignment. If credit and blame can be assigned to individual components of the candidate theory that was tested, then the theory generator can be instructed to modify only the parts of the theory that are still incorrect. There are many techniques for assigning credit and blame. One approach described by Dietterich and Buchanan [1981] is to perform controlled experiments. The generator can generate a set of hypotheses by varying only one component of the hypothesis at a time. Each hypothesis is then tested, and a YES/NO answer is obtained. Because only one component has been varied since the last test, all credit and blame can be assigned to that component for any changes in the overall evaluation of the hypotheses. This strategy only makes sense if it is possible to factor each complete hypothesis into independent subcomponents. If this can be accomplished, then each subcomponent can be tested independently prior to combining the subcomponents into one complete hypothesis. If such a factoring does not exist, then this strategy reduces to exhaustive generate-and-test.

In some domains—including the proposed operating system domain—more than a simple YES/NO answer is available from experiments. In Arthur Samuel's [1967] book-move version of his

checkers learning system, for example, each experiment not only tested whether a proposed move was the correct one—it also returned the correct move as a result. This information was then employed to adjust the current hypothesis.

The more information an experiment provides, the more the learning or theory formation system can learn from that experiment. For example, in the LEX learning system, each experiment returns a complete trace of the efforts of the problem solver to solve the given integration problem. A powerful analytic technique has recently been developed for LEX that involves analyzing this trace and attempting to "invert" each integration operator that was part of the optimal solution path. If this inversion can be accomplished, then the evaluation criteria for what it means to solve a problem can be "pushed through" these inverted operators to compute directly the heuristics for each of the operators in the solution sequence. Utgoff [1982] has even employed this technique to compute the definitions of new terms to be included in the generalization language of the system.

We hope to show with EG that similar techniques can be employed in a more complex theory formation task. A very important difference between EG and all previous theory formation systems, however, is that the system being experimented upon is external to the theory formation system. Consequently, it is not possible to observe directly its internal state. This makes evident a third function of experimentation—it permits the theory formation program to manipulate the external system in order to reveal its internal structure. In the UNIX domain, for instance, experiments can force UNIX into a new state that has never been previously observed. For example, a command to change the current working directory makes a variety of new behaviors observable including changes in the relative names of all files in the system. Once a new state is entered, observation commands can be issued to learn more about that state. From this perspective, it can be seen that EG's task is to explore two spaces: the space of possible operating system theories and the space of possible states of the external system.

1.4 Summary

We are now in a position to state the two basic hypotheses of the EG project:

1. A heuristic search technique similar to that employed in Eurisko is sufficient to discover theories in this domain, and
2. Planned experimentation is essential in large domains in order to avoid combinatorial explosion in the search for theories.

Both of these hypotheses rest on a more fundamental claim: Theory formation can only be accomplished if the system under study can be factored into weakly-interacting components. The essence of heuristic search is to subdivide the overall theory formation task into small subtasks to which heuristics can be applied. The essence of experimentation is to isolate some subcomponent of the system under study in order to observe its behavior (and thus provide guidance to the heuristic search for theories).

This project can be viewed as an attempt to test these hypotheses by developing a new experimentation-based theory formation method.

2 Forming theories about operating system commands

As described in section 1, EG starts with a partial theory of the UNIX file system and incrementally improves this theory by conducting online experiments with a UNIX system over an Ethernet connection. The starting theory of UNIX includes the basic commands for listing file directories, copying files, and deleting files. Also included is an initial theory of the file system and the system calls that manipulate it. This initial theory is incomplete and oversimplified, but it includes enough information to enable EG to manipulate and observe the state of the UNIX system. EG's task is to fill in the missing parts of the theory and deepen the oversimplified parts in order to obtain a complete and accurate theory (at the given level of description).

This section starts by describing the architecture of the EG program. Then a hypothetical example of EG's behavior is presented.

2.1 Program architecture

EG is designed as a set of knowledge sources that interact through a global database and a global agenda. The global database (also referred to as the blackboard) contains a representation of the current theory and any alternatives under consideration. The global agenda is an ordered list of outstanding tasks to perform. An executive routine repeatedly analyzes the agenda, chooses a task, and invokes a knowledge source to perform the task. As the knowledge source runs, it may post new tasks on the agenda.

Tasks fall into two major categories. One kind of task is placed on the agenda when some new information has been posted on the blackboard. This kind of task asks that some knowledge sources be invoked to compute the consequences of this new information and propagate them to other components of the theory. The second kind of task identifies some hole in the theory. It asks that some knowledge sources be invoked to design and execute an experiment to help fill this hole.

Tasks are ranked on the agenda according to their importance. Importance is determined by a collection of heuristic rules such as "If some other task is depending on the results of this task, then this task is important" or "If this task might indicate a serious error in our theory, then this task is important."

From the most abstract viewpoint, the seven knowledge sources in EG can be grouped into two categories: those that generate theories and those that test theories. There are four generator sources, one for each component of the theory:

- TIO—Input/Output theorist
- TC—Command program theorist
- TSC—System call theorist
- TDS—Data structure theorist

Each of these has the task of generating possible modifications to the corresponding component of the UNIX theory. These modifications may involve expanding the theory to include a finer level of detail or revising the theory to correct errors.

There are three knowledge sources that test theories and perform experiments. These are

- The experiment proposer—which examines a problem and attempts to find an experimental method that will help solve it.
- The experiment planner—which develops a specific experiment to perform, and
- The experiment monitor—which performs the experiments and gathers the raw responses from UNIX.

The general behavior of the system is to repeatedly identify a hole in the theory, design and execute an experiment to gather some data, and then invoke appropriate theory-generator knowledge sources to analyze the data and modify the theory.

2.2 An extended example: the `ln` command

This section presents an extended example of EG as it attempts to understand the `ln` command. This command permits the UNIX user to create an alias name for an existing file. After the command `ln source dest` is executed, both names `source` and `dest` point to the same file, so that changing one file changes the other.

We will start by describing the state of EG's UNIX theory at the point where it begins to work on the `ln` command. Then, we will present a hypothetical sequence of knowledge source invocations.

Suppose that EG has been working for a while on the structure of the file system. It has discovered that the file system is structured as a tree of directories and now it is investigating commands (whose names and syntax it already knows) to see what effects they may have on the file system. One of these commands is the `ln` command. Here is the state of knowledge at this point in EG's researches.

Recall that the theory of UNIX is built of four components: file system data structures, system calls, command programs, and I/O syntax. EG's current theory of data structures states that the UNIX file system is structured as a tree, where each interior node in the tree is a directory and each leaf node is a file. There is a global state variable that points to a particular directory in this tree called the current working directory. A file is designated by giving a path through this tree from the current working directory to the desired file or directory. A downward link in the path is specified by giving the name of the file or directory. An upward link is specified by giving the special name `".."`. Each directory implements a mapping between file names and actual files. This mapping is many-to-one and converts file names into unique global identifiers of files. The mapping is implemented in the directory as a table in which file names and file identifiers are juxtaposed. A unique file identifier is an address that points to the actual string of characters that makes up the file. Each file can be thought of as having a canonical path name that starts at the root of the file structure.

The second component of EG's current theory—the system call component—includes the following primitive actions:

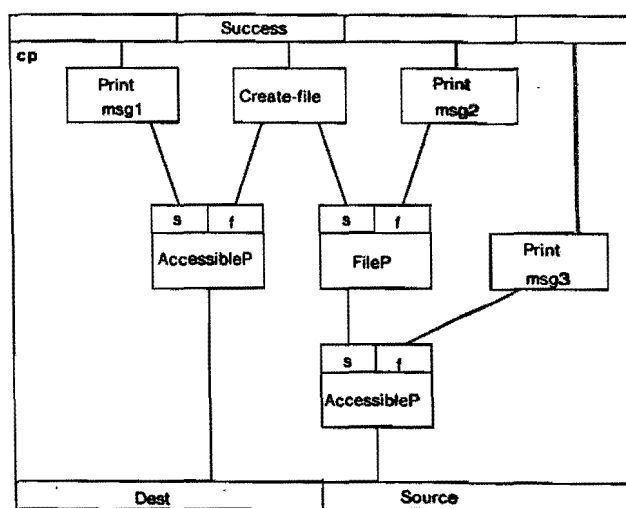
- **create-file(name, contents)**
Create a new file with the indicated name and contents. This is defined as creating a new unique file identifier, storing the **contents** at that address, and adding a new row **<name, unique identifier>** to the appropriate directory.
- **delete-file(name)**
Delete the file with the indicated name. This is defined as removing the directory entry for **name**.

- **rename-file(oldname, newname)**
Change the name of a file. This just changes the name field in the appropriate directory entry.
- **addlink(source, dest)**
Create a new name (**dest**) for an existing file (**source**). This is defined as adding a new entry to the appropriate directory associating **dest** with the same unique identifier as **source**.
- **filep(name)**
Check to see if **name** is the name of a file and not a directory.
- **accessiblep(name)**
Check to see if **name** is defined as either a file or directory.

For each of these primitive actions, there are prerequisites that can be inferred from the current theory of the file-system data structures. For instance, **Create-file** only works if the name does not already exist. **Addlink** requires that the **source** exist and that the **dest** not exist. **Filep** has **Accessiblep** as a prerequisite.

The third component—the command procedure component—includes procedures for the following commands:

- **ls**
Print a list of the file names in the current working directory.
- **ls -ld name**
Print file information about the file or directory indicated by **name**.
- **cp source dest**
Create a new file **dest** whose contents are copied from **source**.
- **rm source**
Delete the file **source**.



msg1: Dest: File exists
msg2: Source is a directory
msg3: Source: No such file or directory

Figure 2: Level 2 theory of the `cp` command.

For each of these commands, EG has an abstract program that defines the command in terms of the system calls given above (see Figure 2). To represent these programs, I am using a slight modification of the programmer's apprentice representation for "deep plans" (see [Rich and Shrobe, 1976]). In this notation, a program is represented as a partially-ordered graph of boxes. Each box denotes a primitive action to be performed. The boxes are linked by control paths that indicate that one box must be performed before another. Conditional boxes have multiple outputs, one for each possible outcome of the action. Although not shown in this simple example, this notation also provides methods for representing iteration and recursion.

Finally, the syntax component of EG's UNIX theory includes a complete grammar for file and directory syntax and for command syntax. The following BNF rules show a portion of this grammar.

```
command ::= lncommand | cpcommand
          | othercommand

lncommand ::= ln sp filename sp filename

cpcommand ::= cp sp filename sp filename

filename ::= legalchar
          | filename legalchar
            {repeat up to 13 times}

sp ::= " "
     | sp " "
```

Notice that the syntax for the **ln** command is provided. The syntax for several commands is given to EG at the start. However, the BNF rule for **command** also includes the nonterminal **othercommands**, which has no rules defining it. Hence, EG knows that its list of commands may not be complete.

Given this body of knowledge, let us suppose that EG has performed an **ln** command and inductively inferred that the purpose of the **ln** command is to perform an **Addlink** operation on its two arguments. Here is one sequence of knowledge-source invocations that might occur.

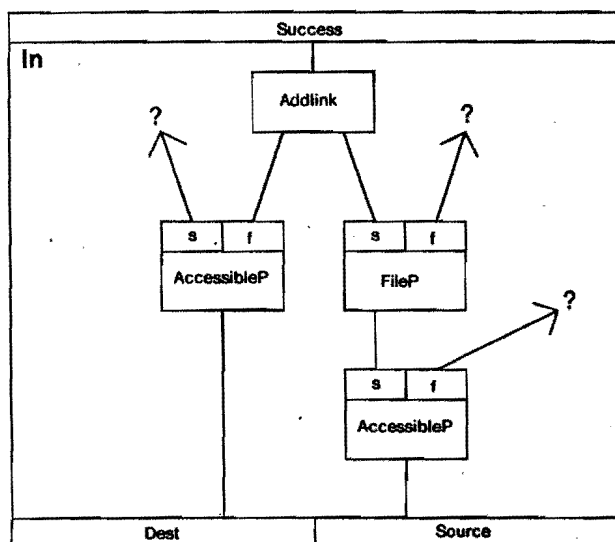


Figure 3: Initial theory of the **ln** command.

Step 1. The TC knowledge source is invoked to build a theory of the **ln** command. In other words, it is called upon to develop all possible abstract programs that might implement the **ln** command using the available system calls. TC produces only one theory for **ln**, which is shown in Figure 3. (In my first implementation of this system, the TC builds this theory by backward chaining from the goal of the command.) This theory says that the main step of the command is to perform an **Addlink** operation on two arguments, the source and destination file names. In order to do this, all of the prerequisites of **Addlink** must be checked. Hence, test boxes have been included that check (a) that the source file is accessible, (b) that the source file is a file, and (c) that the destination file is not accessible. Unfortunately, this is not a complete program for the **ln** command. There are three dangling links: cases where TC cannot guess what the **ln** command will do. These correspond to the three cases in which an error is present in the command's arguments. Hence, TC posts three tasks on the agenda—one for each "hole" in the theory.

Step 2. The experiment proposer is invoked to design experiments that resolve the first of these ambiguities—determining what happens when the source file is not a file. The basic plan for such an experiment is to issue an **ln** command for which **filep(source)** is false and observe what happens. Such an experiment will only be valuable if the observation can be made and the output successfully interpreted. Hence, the experiment proposer must find a set of constraints on the arguments to the **ln** command that force control out the desired branch, while at the same time preventing any other, possibly confusing, events from happening.

I have developed a simple marker-propagation algorithm that can perform this computation. It starts by applying labels to the desirable and undesirable paths in the program. Then, these labels are propagated through the various conditional boxes to determine how each conditional box should be "set." This problem is closely related to the problem of line sensitization in digital circuit test generation [Roth, 1966].

In this particular case, the solution is to choose a source name that is accessible, but that is not a file name (i.e., it is a directory name), and to choose a destination name that is not accessible. An **ln** command with those arguments will not perform the **Addlink** operation nor will it send control out any of the other dangling paths. Once the experiment proposer has developed this plan, it adds a task to the agenda suggesting that the experiment planner be invoked to complete the design of this experiment.

Step 3. The experiment planner is invoked to choose specific names for this experiment. If EG does not have an up-to-date model of the current state of UNIX, then some observation commands must first be issued to find out what files exist. Then, the choice of file names is simple. In general, the problem of choosing specific arguments for an experiment is somewhat tricky, since the choices may not be independent of one another. In such cases, techniques similar to those employed by Stefik [1980] will need to be applied to make some choices and then use those to constrain the other choices. In addition to planning the main step, the experiment planner must also plan the data-gathering commands that are to be issued. In this case, we have some heuristics that tell us that the likely effects of this command are to modify the objects named by its arguments and to display some output. Hence, the following plan is developed:

```
ls -ld d
  Should display
  "drwxr-xr-x 2 x1      32 Apr 20 21:04 d"
ls -ld b
  Should give the error message "b not found"
ln d b
  May print an error message
```

```

ls -ld d
Should display
"drwxr-xr-x 2 x1      32 Apr 20 21:04 d"
ls -ld b
Should give the error message "b not found" or else
directory information about b

```

where **d** is a directory that already exists, and **b** is a non-existing name. Observation actions (the **ls** commands) have been placed before and after the key step (the **ln** command). Notice that the experiment planner computes expectations so that the experiment monitor can determine whether things are "on-course." If **ln** were judged to be a dangerous command, then commands would have been included to create a scratch directory instead of using an existing directory.

Step 4. Now the experiment monitor is invoked to perform the experiment and collect the results. The execution of this experiment produces the following responses from UNIX:

```

% ls -ld d
drwxr-xr-x 2 x1      32 Apr 20 21:04 d
% ls -ld b
b not found
% ln d b
d is a directory
% ls -ld d
drwxr-xr-x 2 x1      32 Apr 20 21:04 d
% ls -ld b
b not found
%

```

(Here are some notes to aid the reader in understanding this trace. The % is the UNIX prompt character. The output of the **ls** command—for example "**drwxr-xr-x 2 x1 32 Apr 20 21:04 d**"—is a single line giving information about the file or directory named by its argument. This information includes the protection bits, the owner, the length in bytes, the modification date, and the name.)

Once the experiment monitor has obtained these responses from UNIX, it posts them on the blackboard and adds a task to the agenda suggesting that some knowledge source should interpret this data.

Step 5. The TIO knowledge source interprets these data to mean that the **ln d b** command had no effect except to print the message "**d is a directory**". This is interpreted as an error message, but, since EG has no ability to understand English, the only thing it can notice about the message is that it mentions the **d** argument (this is typical for UNIX error messages). It is important to note that the problem of interpreting the data is inherently unconstrained. There are two reasons for this. First, the meaning of a string depends on the context in which it appears. A given string appearing in the output, for example, could be the contents of a file, since files can contain arbitrary strings. Or, the string might contain several subfields that denote various objects in the computer, such as protection codes, user names, times and dates, and so on. Second, EG's I/O grammar is never complete, so it is always conceivable that a given string denotes some new class of objects that has never before been encountered by EG. Given this lack of constraint in the parsing process, it is critical that EG be given some knowledge about what to expect. In this context, since we are violating a prerequisite of a primitive action, we are expecting an error message or else some change in one of the command arguments.

Once TIO has interpreted the output, it places this interpretation on the blackboard and adds a new task to the agenda to propagate the consequences of this interpretation to other components of the theory.

Step 6. Hence, the TC knowledge source is invoked to update the command component of the theory in light of the fact that **ln d b**

produced only an error message and had no other effects. The **ln** command theory is updated by attaching a **Print** action of the form **Print(concat(source, " is a directory"))** to the dangling link coming from the **FileP** test.

The process of filling in the dangling links of the **ln** command program continues in this way until the plan shown in Figure 4 is obtained. This theory might serve quite well for a long time, but at some point, EG might attempt to create a link that crosses device boundaries. The starting theory of the file system sketched above is incorrect, because it assumes that a directory maps names to unique file identifiers. It turns out that directories map names to file identifiers that are unique for a given device. The combination of the device and the file identifier uniquely designates the file. As a consequence, it is impossible to create links across devices, since the linking operation involves changing only the within-device file identifier.

Suppose that EG attempted the following cross-device **ln** command:

```

% ln filea /tmp/filea
filea: Cross-device link
%

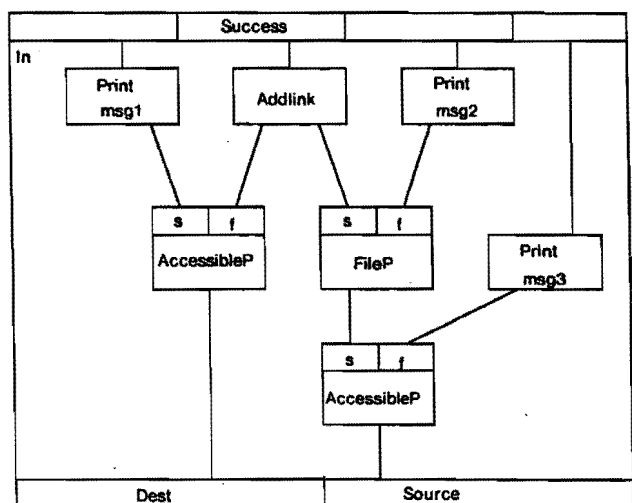
```

The line **filea: Cross-device link** is an unexpected error message. Here is the sequence of knowledge source invocations that results:

Step 45. The experiment monitor notes immediately that this message violates the expectations for the experiment in progress (i.e., this **ln** command wasn't supposed to produce any output at all). Hence, the execution of the experiment is halted, and TIO is queued to interpret this message.

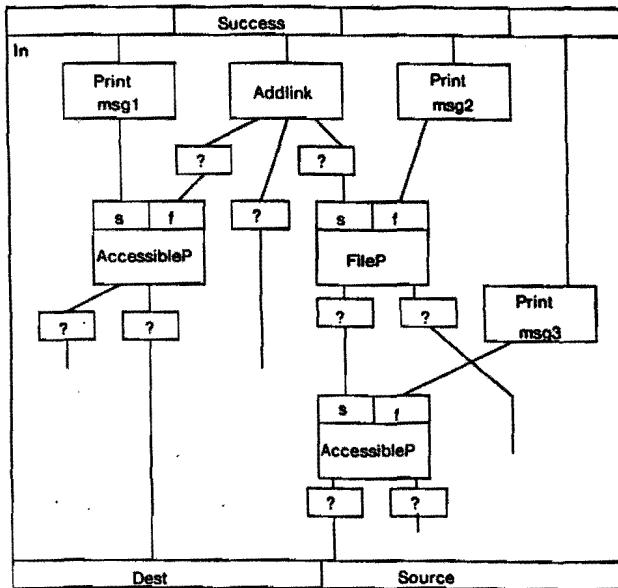
Step 46. TIO interprets this as an error message (since it is a single line and mentions one of the arguments to the **ln** command).

Step 47. TC updates the theory to include an new conditional box. This box is testing some unknown condition at some unknown point in the **ln** program and printing the error message if the test succeeds. The TC can give this condition box a synthetic name, **Cross-device**, but it can't be sure what arguments are being

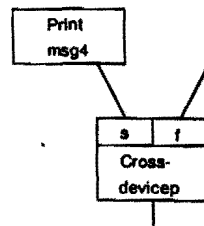


msg1: Dest: File exists
 msg2: Source is a directory
 msg3: Source: No such file or directory

Figure 4: Theory of **ln** after all dangling paths have been observed



Where ? stands for



msg1: Dest: File exists
 msg2: Source: is a directory
 msg3: Source: No such file or directory
 msg4: Source: Cross-device link

Figure 5: Possible locations of the Cross-device test

tested. Any data structure in the operating system might conceivably be checked, and EG doesn't even have a complete list of such data structures. Figure 5 shows the possible locations of the Cross-device check.

TC identifies two problems that must now be solved. First, the exact positions of Cross-device in the `ln` program need to be determined. Most error messages indicate violated prerequisites, hence, the position of each Cross-device check will indicate that some primitive action has this condition as a prerequisite. Second, once the positions have been identified, the prerequisite information must be propagated to the higher levels of the theory. These two problems are added to the agenda.

Step 48. The experiment proposer is invoked to work on the first problem. It has two methods for pinning down the position of a test. One is to hold the arguments to `ln` constant while varying the objects that they denote, and the other is to compare the `ln` command with commands that contain some of the same primitive boxes. By applying the first method, the experiment proposer proposes that `ln filea /tmp/filea` be attempted, but with `filea` now being a non-existent file.

Step 49. The experiment planner must figure out a way to accomplish this. For example, EG could simply delete `filea`. However, if the contents of `filea` are valuable, then EG should instead rename `filea`. In this case, EG chooses to delete `filea` using the `rm` command:

```
rm filea
```

Should produce no output

```
ln filea /tmp/filea
```

Should either print

filea: No such file or directory
or else filea: Cross-device link

```
ls -ld filea
```

Should print filea not found

```
ls -ld /tmp/filea
```

Should print /tmp/filea not found

Step 50. Plan execution produces the following output:

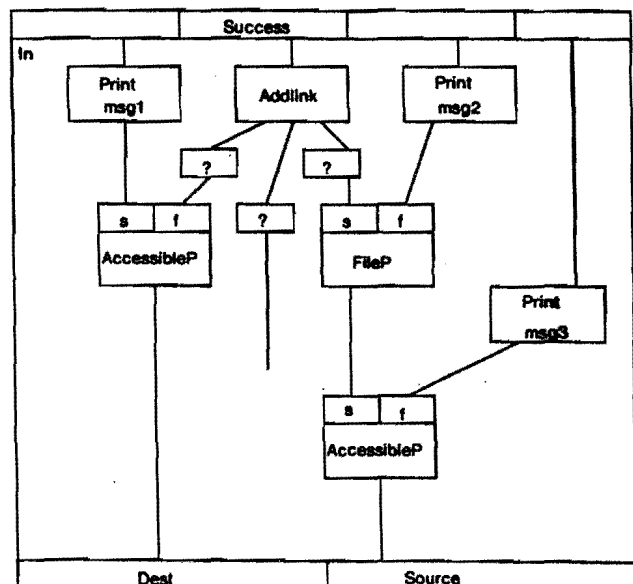
```
% rm filea
% ln filea /tmp/filea
filea: No such file or directory
% ls -ld filea
filea not found
% ls -ld /tmp/filea
/tmp/filea not found
%
```

Step 51. TIO interprets this as being the expected output of the `ln` command when the source file does not exist.

Step 52. TC can therefore conclude that, since Cross-device must be checking something besides the existence of `filea` and since nothing else changed from one experiment to the next, Cross-device should also have been true in this case. Hence, Cross-device is not a prerequisite of AccessibleP.

Similar experiments can be conducted to show that Cross-device is not a prerequisite of the other AccessibleP test or the FileP test. Hence, Cross-device must be a prerequisite of the Addlink action. The resulting theory is shown in Figure 6. Note that the exact position of the Cross-device test is still not known.

An alternative research strategy would be to compare the `ln` command with the `cp` command. Initially, it appears that these



msg1: Dest: File exists
 msg2: Source: is a directory
 msg3: Source: No such file or directory

Figure 6: Locations of Cross-device after experimentation

commands have identical prerequisite structures (compare Figures 2 and 3). The only difference is that the final step in the **cp** command is a **Create-file** instead of an **Addlink**.

Step 49 (alternative approach). The experiment proposer suggests that the command **cp filea /tmp/filea** be executed to see if the **Cross-device link** message will be printed. In this experiment (as in the previous step 49), **filea** names an existing file and **/tmp/filea** names a non-existing file.

Step 50 (alternative approach). The experiment planner develops the following experiment:

```
cp filea /tmp/filea
  Should produce no output or else
  the message filea: Cross-device link
ls -ld filea
  Should print
  -rw-r--r-- 1 x1      27 Apr 27 10:44 filea
ls -ld /tmp/filea
  Should print directory line for /tmp/filea
  or else the message /tmp/filea not found
```

Step 51 (alternative approach). The experiment monitor performs this experiment and obtains the following output:

```
% cp filea /tmp/filea
% ls -ld filea
-rw-r--r-- 1 x1      27 Apr 27 10:44 filea
% ls -ld /tmp/filea
-rw-r--r-- 1 x1      27 Apr 27 10:45 /tmp/filea
%
```

Step 52 (alternative approach). TIO interprets the two output lines as directory information.

Step 53 (alternative approach). TC interprets this as meaning that **cp** performed normally, and hence, that **Cross-devicep** was not checked. It follows that **Cross-devicep** is not a prerequisite of **AccessibleP**, **Filep**, or **Create-file**. Hence, it must be a prerequisite of **Addlink**.

Even though the position of the **Cross-devicep** operation has been determined, this does not resolve the question of what it is checking. EG now takes up this problem.

Step 65. The experiment proposer suggests performing several different experiments in an attempt to discover a pattern in the situations in which **Cross-devicep** is true. These experiments vary conditions that might possibly be relevant to **Addlink**, such as the name and location of the source and destination files in the directory structure, the size of the file, the owner of the file, and so on. These experiments are not shown here.

Step 102. Eventually, TSC (the system-call knowledge source) analyzes the data and discovers that **Cross-devicep(s,d)** is true if and only if the first names in the canonical file names of **s** and **d** are different. This turns out to correspond to device boundaries on the UNIX system where these experiments are run.

Step 103. TDS (the data structure knowledge source) attempts to find a change in the data structure theory that can explain why this would be a prerequisite of **Addlink**. One such change involves altering the definition of the mapping between unique file identifiers and files. In the starting theory, this was a one-to-one mapping represented by juxtaposition. If this mapping is extended to map from the ordered pair
 <first-name-of-the-canonical-file-name,
 unique-identifier> to files, then **Addlink** can be defined as modifying only the unique-identifier field, so that two files must share the same first-name-of-the-canonical-file-name in order for the **Addlink** operation to make any sense.

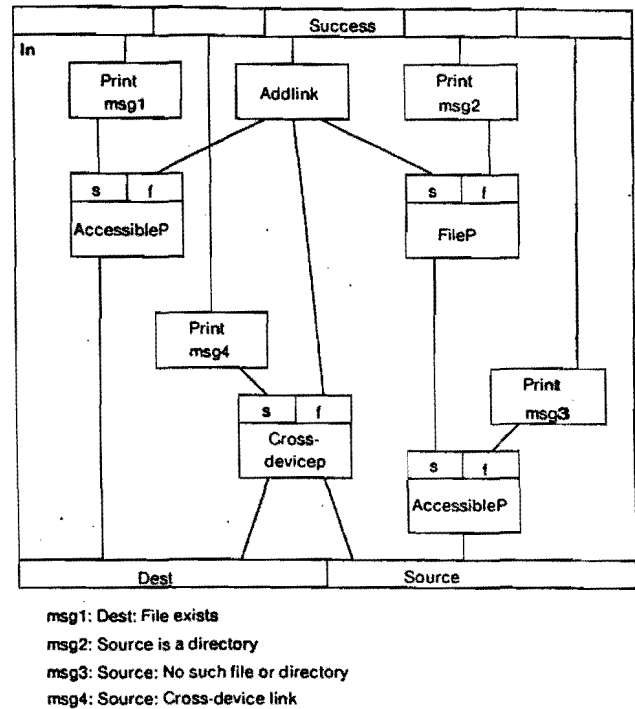


Figure 7: Final program for the **ln** command

Step 104. Given this revision in the definition of **Addlink**, TC revises the program for the **ln** command to include **Cross-devicep** as a third independent test to check the prerequisites of **Addlink**. Figure 7 shows the final program for the **ln** command.

3 Relationship to scientific theory formation

The hypothetical example shown above exhibits some interesting behavior, and one can easily draw parallels with the way in which people learn operating systems. It is worthwhile to ask, however, if EG can tell us anything about scientific theory formation.

Despite the artificial aspects of the operating system domain, we believe that there are several important similarities between this theory formation task and ordinary science.

- The theory being formed is not a single sentence or equation, but rather a description of the operating system at four distinct levels of analysis.
- Interaction with the operating system takes place at the level of strings of characters, which must be interpreted in terms of the current theory. (Observations are "theory-laden" [Hanson, 1958].)
- Data are gathered indirectly through the design, execution, and interpretation of experiments.
- Experiments go beyond simple hypothesis-test experiments—in which a single YES/NO answer is obtained—to include exploratory experiments, in which some new stimulus causes the operating system to enter an unknown state and then observation commands are issued to learn about that state.

Since EG's task resembles a scientific task, we can now ask what EG's methods tell us about scientific methods. In addition to hypothesis testing, EG demonstrates two other important roles for experimentation in science: data gathering and exploration of the system being studied. EG may also shed some light on the origin of theoretical terms. The methods by which EG revises its theory of data structures and system calls can be viewed as a technique for developing new theoretical terms. EG isolates and uses these terms before giving them a precise definition. This technique is analogous to the method employed by Utgoff [1982]. New terms at one level of description are eventually defined as composites of existing terms at a more primitive level.

4 Summary

The goal of the EG research project is to attack a complex theory formation task in order to see what existing methods can be scaled-up and what new methods need to be developed. The task facing EG is to develop a theory of the UNIX operating system that explains the observable behavior of the system at the executive level. Our hypotheses are that (a) no existing methods will work in this domain and (b) manipulation and observation experiments will be critical to controlling the search for theories in this domain.

The EG program is an agenda-driven system in which only a few current hypotheses are maintained. The agenda is a list of problems that require attention including (a) intermediate results whose consequences need to be propagated to other parts of the theory and (b) holes in the theory that need to be filled. Theory-generation knowledge sources perform type-(a) tasks, and experimentation knowledge sources design experiments to solve type-(b) problems. The theory revision process can require that new data structures and system calls be defined in order to make sense of operating system behavior. Implementation of EG has only been under way for a short time.

5 Acknowledgments

This paper owes much to extensive conversations with Jim Bennett on the generate-and-test paradigm for intelligent systems. Specific comments on an early draft of this document have been provided by Jim. The design and development of EG has benefitted from discussions with Mike Genesereth, Saul Amarel, and Lindley Darden. Jock Mackinlay originally suggested the idea of experimenting with operating systems, and Chuck Paulson helped relate the present work to work on digital circuit testing and diagnosis. This research was supported by an IBM graduate fellowship (to TGD), by ARPA grant no. N00039-83-C-0136, by NIH grant no. RR 00785, and by NLM grant number 5-PO1-LM03395.

6 References

- Buchanan, B. G., Mitchell, T. M., Smith, R. G., and Johnson, C. R., Jr., "Models of Learning Systems," in J. Belzer, A. G. Holzman, and A. Kent (eds), *Encyclopedia of Computer Science and Technology*, Vol. 11, Marcel Dekker, New York, pp. 24-51, 1977.
- Buchanan, B. G., and Mitchell, T. M., "Model-directed learning of production rules," *Pattern-Directed Inference Systems*, Waterman, D. A. and Hayes-Roth, F. (eds.), Academic Press, New York, 1978.
- Dietterich, T. G. and Buchanan, B. G., "The role of the critic in learning systems," Rep. No. STAN-CS-81-891, Department of Computer Science, Stanford University. December, 1981.
- Dietterich, T. G., London, R., Clarkson, K., and Drouney, G., "Learning and inductive inference," Chapter XIV in Vol. 3 of *The Handbook of Artificial Intelligence*, Cohen, P. R., and Feigenbaum, E. A., (eds.), 1982. Also available as Rep. No. STAN-CS-82-913, Department of Computer Science, Stanford University. May, 1982.
- Hanson, N. R., *Patterns of Discovery*, Cambridge University Press, Cambridge, 1958.
- Kernighan, B. W., and McIlroy, M. D., "The UNIX Programmer's Manual," Bell Laboratories, 1976.
- Langley, P. W., "Descriptive discovery processes: Experiments in Baconian science," Rep. No. CS-80-121, Computer Science Department, Carnegie-Mellon University, 1980.
- Lenat, D. B., "AM: An artificial intelligence approach to discovery in mathematics as heuristic search," Rep. No. STAN-CS-76-570, Computer Science Department, Stanford University, 1976.
- Lenat, D. B., "Eurisko: A program that learns new heuristics and domain concepts," Rep. No. HPP-82-26, Heuristic Programming Project, Department of Computer Science, Stanford University, 1982.
- Lindsay, R. K., Buchanan, B. G., Feigenbaum, E. A., and Lederberg, J., *Applications of artificial intelligence for organic chemistry: The DENDRAL project*, McGraw-Hill, New York, 1980.
- Lomuto, A. N., and Lomuto, N., *A UNIX* Primer*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
- Mitchell, T. M., "Version spaces: an approach to concept learning," Rep. No. STAN-CS-78-711, Department of Computer Science, Stanford University, 1978.
- Mitchell, T. M., Utgoff, P. E., Banerji, R., "Learning by experimentation: acquiring and modifying problem solving heuristics," Rep. No. LCSR-TR-31, Computer Science Department, Rutgers University, 1982.
- Rich, C., and Shrobe, H. E., "Initial report on a LISP programmer's apprentice," Rep. No. TR-354, AI Laboratory, Massachusetts Institute of Technology, 1976.
- Roth, J. P., "Diagnosis of automata failures: A calculus and a method," *IBM Journal of Research and Development*, pp. 278-281, October, 1966.
- Samuel, A. L., "Some studies in machine learning using the game of checkers. II—Recent progress," *IBM Journal of Research and development*, 11:601-617, 1967.
- Stefik, M. J., "Planning with constraints," Rep. No. 80-784, Computer Science Department, Stanford University, 1980.
- Utgoff, P. E. and Mitchell, T. M., "Acquisition of Appropriate Bias for Inductive Concept Learning," *Proceedings of the Second National Conference on Artificial Intelligence*, Pittsburgh, August, 1982.

HOW TO STRUCTURE STRUCTURED OBJECTS

Ryszard S. Michalski
Robert E. Stepp

Department of Computer Science
University of Illinois
Urbana, Illinois 61801

ABSTRACT

Past research on automated construction of classifications has been concerned with structuring objects that are characterized solely by attribute-value pairs. The methods stemming from that research did not take into consideration the structural information about the objects nor did they utilize general or problem-specific background knowledge in the process of creating a classification. This paper considers problems of generating classifications of structured objects using knowledge about the inter-relationships of various concepts relevant to describing individual objects as well as object configurations.

1. INTRODUCTION

Creating a classification is typically the first step in understanding and formulating a theory about a collection of observations or phenomena. This process is a form of learning from observation (learning without a teacher) and its goal is to structure given observations into a hierarchy of meaningful categories. Past work on this problem was done mostly under the heading of numerical taxonomy and cluster analysis. It was based on the application of some mathematical measure of similarity between objects, defined over a finite, a priori given set of object attributes. Classes of objects were taken to be collections of objects with high intra-class and low inter-class similarity. The methods assumed that objects are characterized by sequences of attribute-value pairs and that this information is sufficient for creating a classification. The methods did not take into consideration any background knowledge about the relationships among object attributes or global concepts that could be used for characterizing object configurations. Nor did they take into consideration possible purposes of classification.

Since the basis for creating classes was purely mathematical, the resulting classifications were often difficult to interpret conceptually. The problem of interpreting the results was left to the data analyst. In addition, since descriptions of objects are simply attribute-value pairs, the methods were inadequate for creating classifications of structured objects, i.e., objects whose appropriate description involves not only object attributes but also relationships among the object parts as well as attributes of the parts.

The recently developed method of conjunctive conceptual clustering [Michalski and Stepp, 1983] overcomes some of the above problems. Given a set of object descriptions, the method constructs a hierarchy of classes and their descriptions which are in the form of conjunctive concepts. The siblings of any node in the hierarchy are optimized according to a certain global criterion of classification "quality." The method, however, still relies on the attribute-value descriptions of objects. It also has only limited ability to take into consideration the background knowledge of various concepts. This paper addresses the problems of extending the above method in two directions:

- (1) how to form classifications of structured objects,
- (2) how to formulate and utilize background knowledge in creating classifications.

In the following sections, the above problems are characterized by simple examples and a methodology for their solution is outlined.

2. STRUCTURING STRUCTURED OBJECTS

In order to describe this problem in greater detail, let us consider a simple example based on rephrasing the problem known as "East- and Westbound trains" [Larson, 1977; Michalski, 1980a] shown in Figure 1. In the old formulation of the problem, given are two collections of trains, those that are "Eastbound" (A to E) and those that are "Westbound" (F to J). These trains are highly structured, each consisting of a sequence of cars of different shapes and sizes. The individual cars carry a variable number of loads of different shapes. In this formulation, this is a typical problem of *learning from examples* (or *concept acquisition*) in which the task is to automatically formulate a simple rule distinguishing between the two classes of objects.

Suppose now the class labels are removed (as in Figure 1) and the problem is to create a meaningful classification of the collection of trains. This is a form of a *learning from observation* (or *concept formation*) problem. To illustrate a possible solution for the concept formation problem, let us go back to the concept acquisition formulation of the problem and consider discriminant descriptions of Eastbound and Westbound trains found by the program INDUCE/2 [Hoff, Michalski, Stepp, 1983]. These descriptions were:

Eastbound trains:

"A train is Eastbound if it contains a short, closed car."

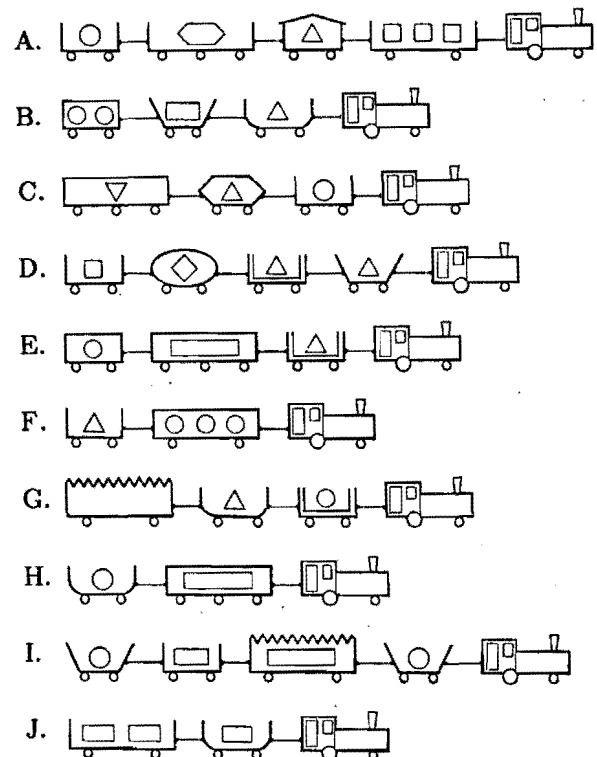


Figure 1. How would you classify these trains?

Westbound trains:

"A train is Westbound if there are just two cars in a train or if a train has a car with a jagged top."

To illustrate the description language used, below is the actual output from the INDUCE/2 program that corresponds to the above descriptions:

Eastbound: $\forall(\text{train}) [\text{contains}(\text{train}, \text{car1})][\text{length}(\text{car1})=\text{short}] \& [\text{shape}(\text{car1})=\text{closed}]$

Westbound: $\forall(\text{train}) [\text{num-cars}(\text{train})=2] \vee [\text{contains}(\text{train}, \text{car1})][\text{shape}(\text{car1})=\text{jagged top}]$

Various aspects of learning structural descriptions from examples are given in [Winston, 1977] and [Dietterich and Michalski, 1983]. The problem of concern here is to develop a general method that when applied to the collection of trains without class labels (Figure 1) could potentially restore the above classification or invent one of equal conceptual appeal. In addition it should generate descriptions of the same kind as above for the created classes, with a possible proviso that each class is described by a simple conjunction only (in this case we would have three classes of trains: those with a short closed car, those with a jagged-top car, and those with exactly two cars).

Such classification construction problems occur when one wants to organize and classify observations that require structural descriptions. Problems of this type include classifying physical or chemical structures, analyzing genetic sequences, building taxonomies of plants or animals, and characterizing visual scenes. In Sec. 4 we propose two methods for solving this type of problem, one data-driven and one model-driven. Before discussing these methods we turn to the problem of how to use background knowledge in creating a classification.

3. BACKGROUND KNOWLEDGE

It can be observed that when people create a classification of some observations or partition a system into subsystems, they employ knowledge about various concepts and relationships relevant to describing the observations or the system. Let us consider a simple example. Suppose that we are observing a typical restaurant table on which there are such objects as food on a plate, a salad, utensils, salt and pepper, napkins, a vase with flowers, a lamp, etc. Suppose a person is asked to build some meaningful classification of the objects on the table. One way to create a classification is to perform the following chains of inferences:

- salt and pepper are seasonings
seasonings are used to add zest to food
seasoned food is something to be eaten
things which are to be eaten are edible
salt and pepper are edible
- salad is a vegetable
vegetables are food
food is something to be eaten
things which are to be eaten are edible
salad is edible

A similar chain of inferences applied to "meat on a plate" or "cake on a dessert plate" will also lead to the concept "edible." On the other hand, a lamp is not food and is therefore not edible. A vase containing flowers is not food and is therefore not edible. Consequently, one meaningful classification of objects on the table is simply "edible" vs. "inedible." The problem that we pose is this: suppose we are given descriptions of objects on the table in terms of their attributes (including their structural attributes) and we want to have a program that on the basis of these descriptions would create the above classification into edible vs. inedible objects. Obviously, such a program would have to be equipped (among other

things) with the above inference rules and with the ability to use them.

4. PROPOSED SOLUTION

This section describes two methods for solving problems of the kind posed above, one data-driven and one model-driven. The data driven method is based on our previous work and reduces the problem of building a classification into a sequence of concept acquisition problems (specifically, problems of determining discriminant descriptions of objects with given class labels). Thus the method will be called RD for "Repeated Discrimination."

The model-driven method is based on selecting *classifying attributes* (attributes that are used to define classes) either from the initially given pool of attributes or from attributes generated by applying inference rules provided by background knowledge. This method will be called CA for "Classifying Attributes."

The common components of the methods are that they both use general and domain-specific knowledge in the process of constructing a classification and that they both utilize a similar criterion for measuring the "quality" of generated candidate solutions. Therefore, before we launch into a description of the methods, we will first briefly discuss our language for describing generated classes and expressing background knowledge inference rules, and then we will discuss a criterion for measuring the quality of proposed classifications.

4.1. The representation language

In order to build a classification of structured objects, we must have an adequate language for describing such objects as well as their classes. This requirement suggests that we use a predicate calculus or some modification of it. Here we have chosen a language called *annotated predicate calculus (APC)*. APC is an extension of predicate calculus that uses several novel forms and attaches to each predicate, variable, and function an *annotation* [Michalski, 1983]. The annotation is a store of information about the given predicate or atomic function. Together with all forms found in predicate calculus, the language also uses a special kind of predicate called a *selector*. A simple selector is in the form:

[atomic-function REL value-of-atomic-function]

where REL (relation) stands for one of the symbols $= \neq < > \leq \geq$. An example of such a selector is

[weight(box) > 2kg]

which means "the weight of the box is greater than 2 kg." A more complex selector may involve *internal disjunction* (disjunction of values of the same atomic function) or *internal conjunction* (the conjunction of atomic-functions having the same value). These two operators are illustrated by the two corresponding examples:

[color(box) = red $\vee$ purple]	"the color of the box is either red or purple."
[color(box1 & box2) = red]	"the color of box1 and box 2 is red."

Selectors can be combined by standard logical operators to form more complex expressions. For example, a statement "there is a green circle on top of a blue or red square" can be expressed as

$\exists(p_1, p_2) [\text{ontop}(p_1, p_2)][\text{color}(p_1)=\text{green}][\text{shape}(p_1)=\text{circle}] \& [\text{color}(p_2)=\text{blue} \vee \text{red}][\text{shape}(p_2)=\text{square}]$

(Logical multiplication is denoted by the symbol & or by concatenating selectors).

The background knowledge is expressed as a set of APC implicative rules:

CONDITION $\Rightarrow$ CONSEQUENCE

where CONDITION and CONSEQUENCE are conjunctions of selectors. If CONDITION is satisfied, then CONSEQUENCE is asserted. To illustrate the implicative statement, consider the assertion "vegetables are food" from the example in Sec. 3. It can be expressed:

$$[\text{is-vegetable}(\text{object1})] \Rightarrow [\text{is-food}(\text{object1})]$$

An alternative way to express it is

$$[\text{type}(\text{object1}) = \text{vegetable}] \Rightarrow [\text{type}(\text{object1}) = \text{food}]$$

In this expression "vegetable" and "food" are treated as elements of the structured domain of the attribute "type." This implication expresses a generalization inference rule called *climbing the generalization tree* (the domain of the attribute "type" is assumed to have a tree structure). Further details on the APC language are given in [Michalski, 1983].

4.2. Measuring the quality of a classification

Creating a classification is a difficult problem because there are usually many potential solutions with no "correct" or "incorrect" answers. The choice of a classification can be made according to some perceived goal of classification or some measure of the quality of the classification. One way to measure classification quality that has been successful in both INDUCE/2 and CLUSTER/2 is to define various elementary, easy to measure criteria specifying desirable properties of a classification and to connect them together into one general criterion called the *Lexicographical Evaluation Functional* (LEF) [Michalski, 1983]. The LEF consists of an ordered sequence of elementary criteria along with tolerances which control to what extent different solutions are considered equivalent. The elementary criteria measure certain aspects of the generated descriptions such as the fit between the classification and the objects, the simplicity of the descriptions (the reciprocal of the number of selectors), the discrimination index (the number of attributes that singly discriminate between all classes), and the dimensionality reduction (the number of attributes that are necessary to classify the objects).

4.3. How to use background knowledge

Building a meaningful classification relies on finding good classifying attributes (attributes that define classes). The data-driven and model-driven methods described in the next section both use background knowledge rules in the search for such attributes. The background knowledge rules enable the system to perform a chain of inferences that add new attributes to object descriptions during the course of classification construction. These new attributes are tested to determine if they are good classifying attributes.

The background knowledge rules can represent both general knowledge managed by the classification program and available to all problems, and problem-specific knowledge provided each time by the data analyst. In either case, the knowledge is supplied in the form of an inference rule (called a background rule, or *b-rule*). Special types of *b-rules* include expressions of arithmetic relationships (*a-rules*), such as

$$\text{girth}(\text{object}) = \text{length}(\text{object}) + \text{width}(\text{object})$$

and implicative rules that specify logical relationships (*l-rules*) such as:

$$[\text{above}(p1,p2)][\text{above}(p2,p3)] \Rightarrow [\text{above}(p1,p3)]$$

Each kind is associated with a condition indicating the situations to which the rule is applicable.

Background knowledge is handled somewhat differently in the data-driven and model-driven approaches. This distinction will be characterized below.

4.4 How to form classifications of structured objects

4.4.1. A data-driven approach: concept formation via concept acquisition

This section explains how a problem of concept formation (here, building a classification) can be solved via a sequence of controlled steps of concept acquisition (learning from examples). To begin with, let us briefly describe the program INDUCE/2 which solves concept acquisition tasks involving structured objects.

Given a set of objects described by annotated predicate calculus and arranged into two or more classes, INDUCE/2 generates a description of each class (in the form of annotated predicate calculus expressions) that covers all the objects in the described class and no objects in any other class. This is accomplished in the following manner.

The objects are divided into two sets: objects belonging to the class being described (called set F1), objects belonging to any other class (called set F0). One object at a time is selected from set F1 (the "focus of attention") and a "star" is built that covers the selected object against all objects in set F0. A star is a set of all alternative descriptions that are maximally general, cover the "focus of attention" object (the *seed*) and possibly other objects from F1, and cover no objects from F0 [Michalski, 1983]. To control computational explosion, a parameter MAXSTAR restricts the number of alternative descriptions that are retained during the star generation process. The "best" description in the star (according to a LEF criterion) is selected, and it becomes part of the solution. The objects covered by the selected description are removed from set F1 and the process is repeated by selecting another seed object (from among those not yet covered) and building a star for it. When all objects in the set F1 have been covered, the solution is complete.

The INDUCE/2 algorithm can be adapted for solving classification construction problems. Given a set of unclassified objects, *k* objects are selected and treated as individual representatives of *k* imaginary classes. The INDUCE/2 algorithm then generates descriptions of each representative that are maximally general and do not cover any other representative object.

These descriptions are then used to determine the most representative object in each class (defined as the set of objects satisfying the class description). The representative objects are then used as new seeds for the next iteration. This process stops when either consecutive iterations converge to some stable solution, or when a specific number of iterations will not improve a classification (from the viewpoint of the criterion LEF).

The control layer that provides the representatives to the concept acquisition process must be able to pick good representatives or else the resulting classifications will be arbitrary without revealing underlying patterns in the data. One technique that has been used in a variety of clustering situations is to pick representatives at random in the first step. In the next steps *central* representatives or *extreme* representatives are selected alternately [Michalski and Stepp, 1983].

This approach requires the selection of a defined number of representative objects (which corresponds to the number of classes). Since the best number of classes to form is usually unknown, two techniques are used:

- (1) varying the number of classes, and
- (2) composing the classes hierarchically.

Since the classification to be formed should be easy to understand by humans, we assume an upper limit on the number of classes that stem from any node of the classification hierarchy. This limit is assumed to be in the range of 4 to 7. Since this limit is small it is computationally feasible to repeat the whole process for

each number of classes. The solution that optimizes the score on the LEF (with appropriate adjustment for the effects of the number of classes on the score) indicates the best number of classes to form at this level of the hierarchy.

To give an example solution to the "Trains" problem from Figure 1, we show in Figure 2 a classification created by a data-driven approach (using an earlier method described by Stepp [1978]). We find this kind of solution appealing because the difference between classes is striking, yet not obvious by casual inspection.

4.4.2. A model-driven approach

In contrast to the data-driven method described in the preceding section, we will now describe a model-driven approach. The underlying goal of the model-driven approach is to find one or more classifying attributes whose observed value sets can be split into ranges that characterize each class. The important aspect of this approach is that the classifying attribute can be derived through a chain of inferences from the initial attributes during the course of constructing the classification. The classifying attributes sought are the ones that lead to classes of objects maximizing the global criterion of classification quality, (e.g., classes representing simple concepts and/or containing objects that share values for all attributes considered important).

The importance of an attribute can be determined either on the basis of the classification goal or by the fact that it implies many other attributes. For example, if the goal of the classification is "finding food," the attribute "edibility" from Sec. 3 is the important classifying attribute. The second way of determining the importance of an attribute can be illustrated by the problem of classifying birds. The question of whether "color" is a more important classifying attribute than "is-waterbird" is answered in favor of "is-waterbird" because it implies more secondary attributes than does the attribute "color" (e.g., is-waterbird implies "can-swim," "has-webbed-feet," "eats-fish").

There are two fundamental processes that operate alternately to generate the classification. The first process searches for the classifying attribute whose value set can be partitioned to form

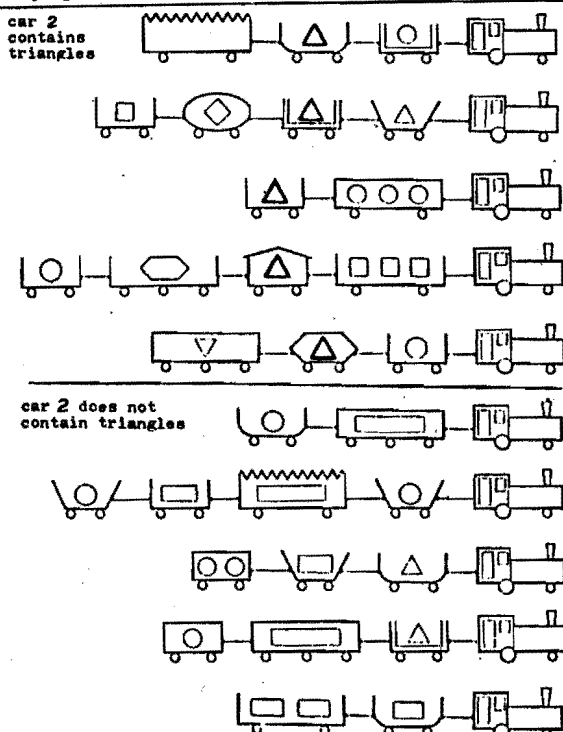


Figure 2. A sample classification of the trains found by a data-driven approach.

classes such that the produced classification scores best according to the LEF. The second process generates new attributes from logical and arithmetic combinations of other attributes, as guided by background knowledge inference rules.

The attribute search can be performed in two ways. When the number of classes to form (k) is known in advance, the process searches for attributes having k or more different values occur in the descriptions of the objects to be classified. These values are called the *observed values* of the attribute. Attributes with the number of observed values smaller than k are not be considered.

For attributes with observed value sets larger than k , the choice of the mapping of value subsets to classes depends on the resulting LEF score for the classification produced and the type of the value set.

For attributes with nominal (unordered) value sets, any combinations of values can be mapped to classes. For attributes with linearly ordered value sets, only non-overlapping closed intervals of values are mapped to classes. For attributes with generalization-tree ordered value sets [Michalski, 1983], sibling values in the generalization tree are mapped to classes.

For example, if k is 4 and the two attributes x_1 and x_2 can have 7 and 3 values, respectively, only x_1 is a candidate for determining the classification. The selection of which of the 7 values denote which of the 4 classes depends on the type of the value set. If we assume that the value set is linearly ordered, one possible classification involving 4 classes could be

$x_1 = 0..1$	(class 1)
$x_1 = 2..3$	(class 2)
$x_1 = 4$	(class 3)
$x_1 = 5..6$	(class 4)

This classification would be selected if it is best according to the quality criterion LEF (which also takes into consideration values of other attributes).

When the number of classes to form is not known, the above technique is performed for a range of values of k . The best number of classes is indicated by the classification that is best according to the LEF.

The attribute generation process constructs new attributes from combinations of existing attributes. Certain heuristics of attribute construction are used to guide the process. For example, two attributes that have linearly ordered value sets can be combined using arithmetic operators. When the attributes have numerical values (as opposed to symbolic values such as "small," "medium," and "large") a trend analysis can be used to suggest appropriate arithmetic operators as in BACON 4 [Langley, Bradshaw, Simon, 1983]. Predicates can be combined by logical operators to form new attributes through I-rules. For example, the rule

$[\text{cold-blooded}(a1)][\text{offspring birth}(a1)=\text{egg}] \Rightarrow [\text{type}(a1)=\text{reptile}]$

yields a new attribute "type" with a specified value "reptile." Using this rule and similar ones, one might classify some animals into reptiles, mammals, and birds (even though the type of each animal is not stated in the original data).

5. SUMMARY

We have discussed problems of creating a classification of structured objects and have outlined a data-driven method and a model-driven method. The first method transforms concept formation into a sequence of concept acquisition tasks. The second method forms classes by partitioning the value set of one or more classifying attributes that were determined as most appropriate according to a classification quality criterion (LEF). The classifying attributes are either selected from the initially given ones or derived through an application of inference rules defined in the problem background knowledge. Both approaches make extensive use of

attribute constructions through inference rules provided by general and problem-specific background knowledge. The capability to incorporate background knowledge in classifying structured objects adds a new dimension to the tasks of concept formation and data analysis.

REFERENCES

- Dietterich, T.G. and Michalski, R.S., "A Comparative Review of Selected Methods for Learning from Examples," chapter in *Machine Learning*, R.S. Michalski, J. Carbonell and T. Mitchell (Editors), Tioga Publishing Company, 1983.
- Hoff, W.A., Michalski, R.S., Stepp, R.E., "INDUCE 2: A Program for Learning Structural Descriptions from Examples," Technical Report, Department of Computer Science, University of Illinois, Urbana, Illinois, September, 1983.
- Larson, J., "Inductive Inference in the Variable-Valued Predicate Logic System VL₂₁: Methodology and Computer Implementation," Ph.D. Thesis, Report No. 869, Department of Computer Science, University of Illinois, Urbana, Illinois, 1977.
- Langley, P., Bradshaw, G.L., Simon, H.A., "Rediscovering Chemistry With the BACON System," chapter in *Machine Learning*, R.S. Michalski, J. Carbonell and T. Mitchell (Editors), Tioga Publishing Company, 1983.
- Michalski, R.S., "Pattern Recognition as Rule-Guided Inductive Inference," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-2, NO. 4., pp 349-361, July, 1980a.
- Michalski, R.S., "Knowledge Acquisition Through Conceptual Clustering: A Theoretical Framework and an Algorithm for Partitioning Data into Conjunctive Concepts," *Policy Analysis and Information Systems*, Vol. 4, No. 3, pp. 219-244, 1980b.
- Michalski, R.S. and Stepp, R.E., "Concept-based Clustering versus Numerical Taxonomy," Technical Report 1073, Department of Computer Science, University of Illinois, 1981.
- Michalski, R.S., "A Theory and Methodology of Inductive Learning," chapter in *Machine Learning*, R.S. Michalski, J. Carbonell and T. Mitchell (Editors), Tioga Publishing Company, 1983.
- Michalski, R.S. and Stepp, R.E., "Learning From Observation: Conceptual Clustering," chapter in *Machine Learning*, R.S. Michalski, J. Carbonell and T. Mitchell (Editors), Tioga Publishing Company, 1983.
- Winston, P.H., *Artificial Intelligence*, Addison-Wesley, 1977.

THE ARCHITECTURE OF JUMBO

Douglas R. Hofstadter
Computer Science Dept.
Indiana University
Bloomington, Indiana 47405

ABSTRACT

Jumbo is an artificial-intelligence project concerned with the reshuffling of parts to form different coherent or "well-chunked" wholes (specifically anagrams, but without using a dictionary). It is based on the concept of a parallel nondeterministic terraced scan: a somewhat Hearsay-like method of homing in, via cues at many levels, to the best category to which to assign something. The process is guided by the "happiness" of structures it creates. A structure's happiness is a measure of how easy it is to accommodate the Platonic concept it is currently considered to be an instance of, to it. For Jumbo to discover good solutions, its data structures must be fluidly reconfigurable, and this is achieved by regrouping, rearrangement, and disbanding. The overall behavior of Jumbo is not directly programmed, but is an emergent, or statistical, outcome, of the interaction of the microscopic computational activities happening in parallel.

Key terms: Perception, recognition, parallelism, artificial intelligence, terraced scan, fluid data structures, temperature, self-watching, emergence.

PROLOGUE: NOT MUMBO-JUMBO

Jumbo is an artificial-intelligence research project with a concrete purpose and an abstract motivation. It is one of several projects, all focused on various facets of one shared long-term goal [Hofstadter 1983]. In essence, that goal is to demonstrate the equivalence of cognition with "deep perception" -- the non-modality-specific layers of perception that involve funneling into highly abstract and often unverbalizable categories [Schank 1980, Hofstadter 1979, 1981b, and 1982b]. The slogan "cognition equals recognition" captures the idea.

Deep perception (recognition) [Bongard 1970, Hofstadter 1982c] is an unsolved -- in fact, almost untouched -- problem of artificial intelligence. It is a far cry from low-level vision; it is the murky area where syntactic sensations interface with semantic categories. This syntactic-semantic transition zone, I feel, is the core mystery of all of intelligence. Before we can make machines that can learn, we must understand how concepts

are structured and can be addressed and compared. Therefore Jumbo is a prelude to a more ambitious project in learning.

JUMBO AND THE ANALOGIES IT IS BASED ON

Jumbo's concrete purpose is to imitate human skills used in the anagram game "Jumble" [Arnold 1982]. In this game, a human player seeks an English word made out of a few given letters. The idea of the Jumbo system is for a computer to construct English-like word candidates ("pseudo-words") from a given set of letters. Jumbo possesses no dictionary of English; it makes its constructions purely by referring to stored knowledge defining how consonant and vowel clusters are formed out of letters, syllables out of clusters, and words out of syllables. Thus Jumbo is a building program; it begins with isolated atomic units (letters), and gradually constructs conceptual molecules or chunks ("gloms") out of them. There are molecules at various levels: clusters, syllables, and words.

Why no dictionary? Because that is irrelevant to the mental processes I am attempting to model, namely the unconscious composition of coherent wholes out of scattered parts. A dictionary could be added, but it would be irrelevant to Jumbo's serious purpose, which is to be a model of the mental process of assembly and transformation -- not an expert system for playing a word game. The hope is that these processes (and an architecture allowing their smooth integration) are notions exportable to more complex domains.

The strategy of Jumbo is based on two analogies. One is to the way that complex molecules are constructed inside a living cell [Lehninger 1975]; the other is to the way that bonds of human friendship are formed in a chaotic world.

I will describe the biology analogy first. In a cell, the lowest level of molecule is the atom. Above that, there are very small molecules, which might be compared to very tight consonant clusters (such as "th" or "ng"). The next level up is that of, for instance, amino acids. These could be compared to loosely bound clusters, such as "str" or "ngth" or "oo". Then there are linear chains of

amino acids -- polypeptide chains (often complete proteins). They could be likened to syllables (often complete words). Many proteins (words) consist of an agglomeration of several polypeptide chains (syllables).

The assembly of molecules in a cell takes place throughout its cytoplasm (the major part of the interior of the cell). For each molecule type, there is a standard chemical pathway by which it is assembled. The assembly pathway for a given molecule type may involve dozens of steps that build it piecemeal. All over the cytoplasm, that pathway is followed by separate, independent, parallel but non-synchronized assembly processes.

Next I will briefly sketch out the "friendship" analogy -- or perhaps the "romance" analogy would be a better term, since the building of romances is the main driving image. In society, the atomic units are individual persons. Then there are two-person units (couples). Above that there are friendships among couples, and larger societies. This metaphor, however, has less to do with the multi-leveledness of society than with the time-dependent manner of formation of bonds. The basic idea is that it takes experimentation ("flirtation" and "dalliance") for two people to decide if they wish to be romantically involved.

The first stage is chance-dependent: do the two individuals come near enough to notice each other? If not, there is no hope of a bond forming. If so, this initial noticing may generate the germ of a romance: a "spark" propelling the partners into a desire for further exploration. If they survive this next stage of flirtation with each other, then I call what results a "flash": a mutual infatuation not yet resulting in a bond. The next stage is that of casual dating ("dalliance"), determining compatibility in a more serious way. If they survive this stage, they can form a genuine bond -- a "romance". There is still a further stage beyond this, of course, called "marriage".

Romantic relationships often break up. Some breakups result from internal tensions in the relationship; others are instigated by the arrival of a seemingly more desirable partner. If an attractive potential partner turns up, many things have to be seriously weighed against each other. The outcome depends on the internal "happiness" of the marriage, the excitement promised by the potential partner, and the mood of society regarding divorce.

In the cell, too, breakups of molecules occur constantly. Large molecules break into smaller constituents, and from these pieces new molecules are then assembled. Assembly and disassembly are not spontaneous but are mediated by agents that are complex molecules in themselves -- enzymes. For each type of operation, there is a specific type of enzyme.

THE SIGNIFICANCE OF JUMBO'S TASK DOMAIN

Now what is significant about coming up with some English-like rearrangements of a set of letters? Well, the perceptual process is essentially one of constructing larger units out of smaller ones, with temporary structures at various levels and permanent mental categories trying to accommodate to each other. A lot of back-and-forth motion must occur -- that is, construction, destruction, regrouping, and rearrangement of structures. The architecture for a system to carry out this process is the result of many subtle decisions about how independent processes should interact, how structures should be put together or broken apart, what kinds of things form stable structures, what easy ways are of making new possible structures when old ones are seen to be inadequate, and so on [Lea 1980, Hanson 1978, Waterman 1978].

The Hearsay-II speech understanding system [Reddy 1976, Erman 1980] was a sophisticated example of such an architecture; Jumbo is an attempt to construct another such architecture, and to focus clearly on the essence of the matter. That is one reason why the domain is kept small.

JUMBO AND PARALLELISM

Jumbo is a parallel system, yet its parallelism, based on the cell, diverges from that of some models of parallelism. In a cell, many metabolic activities take place simultaneously at different spatial locations. Each activity, whether anabolic (i.e., bond-forming) or catabolic (i.e., bond-breaking), is carried out by an enzyme. The typical anabolic enzyme's action involves the bonding together of two molecules. How does the enzyme find those molecules? It has "crevices" called "active sites" that match only molecules of the right type. It wanders around the cytoplasm, and when it bumps into a structure that fits one of its active sites, it latches onto it, and then the two wander together until the enzyme's second active site gets similarly filled at random. At that point, the enzyme is triggered into action. It performs its joining operation, and releases its product. This product is then potential material for a new enzyme to graft something else onto it. Thus larger pieces get built from smaller ones, in a random way. Yet despite the randomness, specific building tasks defined by elaborate chemical pathways get carried out efficiently and reliably.

Jumbo's parallelism is modeled on the distributed parallelism of the cell. A salient difference is that Jumbo runs on a sequential computer, so its parallelism must be virtual rather than actual. However, Jumbo could be adapted to run on a distributed parallel computing architecture of the right sort.

At the outset, all the letters are "lonely" and clamoring for attention. What chooses who gets to

flirt or bond with whom? In essence, it is random, as in the cell or the real social world. To visualize Jumbo, imagine the letters floating around in a three-dimensional fluid -- the "cytoplasm". If two letters come close to each other, they can "see" each other and possibly "spark".

What determines whether a spark is successful? At the outset, when all letters are unattached, it is affinity ("chemistry", as the intriguingly accurate popular metaphor would have it). For example, "s" and "h" (in that order) are well suited, but in the opposite order, they are not. Affinity is not all or none. Some letters are very happy with each other ("s" and "h"), some only mildly happy with each other ("d" and "w", as in "dwell"), and some not at all happy together ("j" and "x"). There are all degrees of affinity. Correspondingly, some sparks are strong, some weak, and some fizzle.

What is the computational meaning of "sparking"? A spark is a data structure telling who is sparking with whom, and in what order. Manufactured at the same time as the spark and associated with it is a piece of code called a "codelet", which is placed in a queue-like structure called the "coderaack", so that it can be run. The codelet, when run, will look at the spark, evaluate its viability, and then suggest whether it is worthwhile going on with further exploration in this tentative "romance". If this flirtation fails, then both letters will go on their merry ways, each one free to spark with other partners instead. If the flirtation seems promising, though, then the codelet will create a "flash" -- the next stage of a romance.

Which codelet on the coderaack shall run first? Associated with each codelet is a number, its "urgency". The choice of which codelet to run next is a weighted but random choice, the urgency of a codelet being its weight in the coderaack. Thus a codelet with urgency 10 is five times as likely to be picked to run as a codelet with urgency 2. But there is no guarantee that it will be run first. It is all probabilistic, so that only in a statistical sense will what "should" happen more often actually happen more often.

Whenever a codelet is run, it is removed from the coderaack. The only traces it leaves behind are: (1) changes that it has caused in the cytoplasm; and (2) new codelets it has placed on the coderaack. This enables lengthy processes to be carried out in small disjoint steps, each step setting up its own possible continuation. It resembles the way that long chains of chemical reactions get carried out in independent small steps in the cell.

THE CONCEPT OF A TERRACED SCAN

Why a spark, then a flash, then perhaps a dalliance, and so on? Why not check out the whole thing at once? The answer is: real-time pressures.

If this potential union were the only possibility in the world, then time wouldn't press. But if there are many possible options to choose from, time does indeed press. One needs to circumvent what at first might seem an awful necessity: that of exploring all potential matchups to an equal degree before making up one's mind about which one works best. There is a profound need for protecting oneself from this sort of absurdity. People develop ways of quickly eliminating things of little interest to them and homing in on the good possibilities. This is the idea of the "terraced scan": a parallel investigation of many possibilities to different levels of depth, quickly throwing out bad ones and homing in rapidly and accurately on good ones.

The terraced scan moves by stages: first one performs quick superficial tests, and one proceeds further only if those tests are passed. Each new stage involves more elaborate tests, which, if passed, can lead to a further stage -- and so on. Furthermore, "passing a test" is not an all-or-nothing affair; each test produces a score, indicating how promising this line of investigation appears so far. One uses this score to determine the urgency of continuation codelets. This provides the desired layeredness to the evaluation of the goodness of a potential matchup.

Since Jumbo is serial, it must interleave the tests making up the parallel explorations, so that many possibilities can be simultaneously explored. Each stage determines the urgency of its follow-up stage, so that the speed of a long process is locally adjustable to reflect the promise it offers.

The way Jumbo works, then, is that it examines many possibilities in simulated parallel, letting the more promising ones move ahead faster than the less promising ones. Letters begin by sparking with one another. Each successful spark engenders a flash. Each successful flash engenders a codelet which -- if and when run -- will bond the given letters together into a new structure -- a glom. Gloms, like letters, also spark together, flash together, and can be bonded together into larger and larger gloms.

While two given items are sparking, two others may be flashing. Others may already have been bonded together to make a glom. Although any particular "romance" has to go through a fixed sequence of increasingly elaborate testing stages, many such explorations may be happening in parallel, and need not be in phase. Flash-evaluators, spark-evaluators, and other types of codelets commingle and coexist in the coderaack without conflict.

Furthermore, nothing prevents a given item from sparking with several other items at once. Thus parallel exploration of more than one "mate" can be carried out. Up to a point, a given item can pursue two romances at once, just as a person can pursue two

romantic possibilities at once, by interleaving the dates and letting the rival romances develop in parallel. There comes a point, however, where if things are to go any further in a romance, commitment is necessary. This means bonding.

GLOMS, OR COMPOUND STRUCTURES

The idea of bonding is to create compound structures that have stability and that are difficult to penetrate or break apart. Typical gloms (bonded structures) include consonant and vowel clusters, syllables, and words. All of these may be incomplete or imperfect. Examples of incomplete or imperfect structures are "ngt", an incomplete consonant cluster (it needs an "h" at the end), and "oot", "y", and "ies", which are incomplete syllables (they need opening consonant clusters).

A bond-making codelet is like an enzyme in a cell causing the bonding of two pieces into a larger piece. The new piece contains the two previous pieces as parts. The parts are still accessible to the outside world, but not as easily as before. Moreover, the new structure now can enter into new kinds of relationships that neither part was entitled to.

Any new glom in the cytoplasm is given a name. Accessible through that name are facts such as the names of the constituents of the glom, the type of the glom (is it a syllable, word, etc.), the "happiness" of the glom (is it complete or not? how happy are its parts?), and so on. To the lowest approximation, being encapsulated inside a glom renders an item inaccessible to the outside world, much as being married -- at least in theory -- prevents a person from having new romances. In that sense, a new glom effectively replaces its pieces in the cytoplasm.

However, the creation of a new glom does not set off a massive search in which all codelets referring to parts of the new glom are purged. Those codelets are allowed to stick around, and when run they will usually just fizzle harmlessly. It is like learning that someone you find attractive is involved; usually you just give up with a sigh.

It is possible, though, that a piece inside a stable glom sparks so strongly with an external piece that the result challenges the bond holding the first piece in. Depending on the "happiness" of the glom, the challenge may be defeated or it may emerge victorious, which will result in the dissolution of the bonds making the old glom, and the establishment of a new glom. Thus in Jumbo, being "committed" (bonded) acts as a damper, but not a total inhibitor, of exploration of rival romances -- as is the case with people.

PARALLEL CONTEMPLATIONS VERSUS PARALLEL REALITIES

Jumbo aims at converting the contents of the cytoplasm from an initial array of unconnected and unrelated letters into a single final structure -- a coherent and hierarchically structured assembly of parts, a candidate for "wordhood". Hearsay II also built hierarchical structures out of an initially formless chaos, but with one salient difference: it constructed multiple top-level structures -- rival interpretations of what the speaker was saying -- at once. An analogous vision program would construct several top-level interpretations for a scene in parallel.

This strikes me as implausible. We do not experience a cognitive superposition of states, seeing (or hearing) something two or more ways at once. The famous examples of visual ambiguity -- the flipping Necker cube, the vase-face picture, and many others -- substantiate my point. While over time we can easily flip back and forth between top-level points of view, our cognitive systems do not easily support seeing something in two ways at one moment.

This suggests that parallelism becomes less and less relevant at progressively higher levels of activity. This idea is mirrored in the way Jumbo works. As structures coalesce, the system gradually makes a transition from being distributed and parallel to being localized and serial; at the end, there is only one top-level glom left in the cytoplasm. Two gloms are never just different arrangements of the same lower-level structures. Therefore, a certain kind of parallelism is nonexistent in Jumbo -- namely, the parallelism you get when you entertain alternative worlds. Jumbo has but one world (its cytoplasm). However, parallel contemplation of alternative worlds (i.e., possible actions to try in the future) is permissible in Jumbo; in fact, it is of the essence. Jumbo is free to "muse" about many possible rival actions, but once one is chosen and carried out, then that's the way the world is.

Let us thus draw a distinction between two types of codelet. A "musing" codelet contemplates a future possibility without actually creating it; an "action" codelet realizes a contemplated possibility by carrying out a change in the cytoplasm. Musing codelets perform "read-only" operations on the cytoplasm (e.g., evaluating sparks or flashes); action codelets perform "write"s in the cytoplasm (e.g., making, breaking, strengthening, or weakening a bond).

In summary, in the early stages of glom-building, when Jumbo's cytoplasm is composed of many scattered pieces, nonoverlapping actions in it can be viewed as occurring simultaneously (whether the simultaneity is virtual or real); however, as gloms merge and become larger but fewer in number,

parallelism of action inside the cytoplasm gradually gives way to seriality, although the parallel contemplation of possible actions to take continues unabated.

THE HAPPINESS OF GLOMS

At any stage, Jumbo's cytoplasm contains gloms having various degrees of "happiness" -- a numerical measure applicable to all gloms. There are two components to happiness. The INTERNAL component tries to summarize how well the glom, as a specific instance of a general concept, realizes that Platonic ideal. The EXTERNAL component tries to summarize the current situation of this glom in the cytoplasm. Is it part of a larger glom (thus "secure"), or is it unattached (thus "lonely")?

Consider a syllable-level glom. It must ask itself, "How good a syllable am I?" This question is answered by a number taking into account the degree of completeness of the glom, the way its pieces fill the roles they are supposed to play, the suitedness of its parts for each other, and -- recursively -- the internal happinesses of its parts.

Take the syllable-candidate "lik", for instance. This glom, seen as a potential syllable, is complete: it has a filler for the three main roles required for "syllablehood": initial consonant cluster, vowel group, and final consonant cluster. (Such roles are analogous to the roles of "husband" and "wife" in a marriage, although not all three of them have to be present in every syllable, "ing" being a good example of an internally happy syllable lacking an initial consonant cluster.) Being a COMPLETE syllable is not enough to make "lik" an internally HAPPY syllable, because "k" does not finely fill the "final consonant cluster" role. By contrast, "lick" is an internally happy syllable, since "ck" is a fine final consonant cluster.

Internal unhappiness makes a glom "squeak". For instance, the incomplete syllable "li" wishes to be made complete, so it squeaks quite loudly. The imperfect syllable "lik" will squeak, but not quite so loudly.

In real life, George may ask himself, "How good a husband would I make?" without having any particular wife in mind. He may feel that he fills the abstract "husband" role ideally without there being any specific marriage into which he fits. An analogous notion holds for gloms. Take the syllable "lick". Internally, it is great, but if it has not been incorporated into a larger structure, it will still be unhappy. This brings us to the external component of happiness.

The external component of any glom's happiness depends on how the glom fits into the current state of the cytoplasm. Initially, all letters are unhappy but not desperate, because they are all in the same

boat -- unmatedness. As some get mated and others don't, however, a new "emotion" enters the picture: jealousy. The longer a letter goes unmated, the unhappier it becomes. It "squeaks" to express its unhappiness. This goes as well for unmated higher-level gloms.

"Squeaking" translates into likelihood of sparking with other items in the cytoplasm -- that is, the urgency of a codelet that will place the item on a queue of potential "sparkers". As in real life, it's the squeakiest wheel that gets the oil.

"Happiness" is thus a measure that a glom computes, taking into account both its self-estimated internal stability, and its self-perceived "normalcy" relative to the external society. Jumbo's overall goal could be succinctly stated as: producing one happy top-level glom. En route to this goal, it has to be guided by the happinesses of the intermediary structures it makes.

DESTRUCTIVE OPERATIONS IN JUMBO

It would be nice if the randomly-driven building processes that take place from the ground up always gave rise to an excellent candidate for wordhood. However, since this is unlikely, there must be ways of changing around what has been constructed. Since much care has gone into the buildup of the constituents on various lower levels, it would be reckless to disband the thing without trying some reworking first.

Traditionally in artificial intelligence, reaching a decision point where one sees that one has to go back and undo something is a critical moment. The standard solution is "intelligent backtracking" [Bobrow 1974], in which you either take a different fork at that decision point or backtrack further. The latter amounts to undoing one decision after another until you find one you can live with, and then starting there from scratch.

Taking a different fork requires you to choose at random or by reason. The latter is obviously preferable, but the world does not always give you enough time or knowledge to reason everything out. Even the best-calculated tine of the fork may lead to no success, in which case you would have been better off choosing at random, for you wouldn't have invested so much time attempting to foresee the unforeseeable. Thus the advantage of choosing at random is that you don't worry over matters that you can't possibly anticipate -- you simply plunge forward and see what happens.

This is what Jumbo does. However, this should not be confused with blindness. Although the "choices" at branch points are made at random, there are powerful biases built in. Not everything has an equal chance; indeed, smart ideas will tend to be assigned high urgencies, and dumb ones low urgencies.

There is no guarantee of this, nor is there a guarantee that high-urgency codelets will get run first, since after all, randomness is randomness -- but in the long haul, things will work out favorably. This, at any rate, is the hope on which Jumbo is based.

There is another aspect to the hope placed in randomness: it seems more likely that interesting routes will be discovered by accidental exploration than if the "best" route is always chosen at every junction. Furthermore, having an extremely simple control structure allows the system to do things that a very complex control structure would get thoroughly tangled up in. Complex control structures are highly brittle, whereas probabilistic control structures are rubbery and don't care what happens in what order.

So, what are the possible remedies to choose from, when there is an unhappy pseudo-word? There are two basic types of transformation: "entropy-preserving" and "entropy-increasing". By "entropy", I mean "perceived disorder". A cytoplasm containing no bonded letters has maximal entropy; a cytoplasm containing just one pseudo-word has minimal entropy. Entropy-preserving transformations do not increase disorder: they preserve the number of gloms. By contrast, entropy-increasing transformations destroy gloms, bringing the cytoplasm closer to its initial state of total disorder.

Of the two varieties of entropy-preserving transformation, the milder one is REGROUPING: readjusting internal boundaries so as to yield new substructures without rearranging anything. The other, more drastic, variety is REARRANGEMENT: taking pieces of a glom and shuffling them around to make new gloms on the same level [Holland 1981]. A more radical remedy is the entropy-increasing transformation of DISBANDING: pulling a glom's top-level bonds apart and perhaps even disbanding some of the lower-level structures.

ENTROPY-PRESERVING TRANSFORMATIONS

Suppose that the unhappy pseudo-word is "pangloss", perceived as two syllables -- "pang" and "loss". First consider regrouping. One possibility is to move the "g" from the first syllable to the second, thus making "pan-gloss". On the other hand, it is not possible to detach the entire consonant cluster "ng" from the first syllable and to reattach it to the second consonant cluster, because "ngl" is not an acceptable consonant cluster.

A more complex type of regrouping can occur when two syllables coalesce into one, or one breaks into two, like drops of liquid. A simple example can be given using four letters found inside "pangloss". Suppose the two syllables "ap" and "so" get bonded together in this order: "so-ap". Now "soap" might be bisyllabic (like "react" or "Moab"), but its preferred perception is as a single syllable.

Therefore Jumbo should be able to carry out this type of regrouping fluently. The "fission" process -- moving from one-syllable words like "soap" to two-syllable words "so-op" or "so-ap" -- should be as natural for Jumbo as the "fusion" process.

The next entropy-preserving alternative, rearrangement, offers many possibilities. One would be to perform the operation known as "spoonerism": interchanging the initial consonant clusters of two syllables. This produces a new two-syllable candidate, "lang-poss". The term comes from the name of an Anglican minister, The Reverend Spooner, but it is tempting to ignore that dignified etymology and to invent parallel culinary terms: "forkerism" (the interchange of final consonant clusters) and "kniferism" (the interchange of vowel groups). Operating on "pang-loss", those would produce "pass-long" and "pong-lass" respectively. Spoonerism need not involve entire clusters; sometimes pieces of clusters can be exchanged, as in the transformation of "slag-pons" to "lag-spons".

Rearrangement can involve the EXCHANGE of syllables ("pang-loss" to "loss-pang"). It can also involve the internal REVERSAL of a syllable at the cluster level ("stan" to "nast") or the letter level ("stan" to "nats"). Reversal at either level, unfortunately, poses tricky problems. Obviously, not all syllables can be reversed at the cluster level, since some consonant clusters cannot function initially, while others cannot function finally (try "knots", for example). Reversal at the letter level is equally knotty (try "thump"). Thus one cannot just barge ahead with either type of reversal; musing codelets must prepare the way by checking out the desirability of contemplated actions. This emphasizes once again that, although Jumbo is permeated by randomness, it is also permeated by dynamic tests imbuing it with caution and foresight.

Another type of rearrangement is where one syllable wants to trade its final consonant cluster for another one's initial consonant cluster. For instance, in "plag-noss", "plag" might want to trade its "g" for "noss"'s "n", to yield "plan-goss". Again, the feasibility of such a maneuver depends on whether the given clusters can function in the desired new positions. A natural nomenclature for such trades suggests itself: "sporkerism" and "foonerism".

A musing codelet proposing a trade resembles an offer by one baseball team to trade its pitcher for another team's left-fielder. Such offers are usually made out of recognition of weaknesses, and the feeling that both can benefit from the switch. The weaker a team is in a certain position, the more disposed it is to give away something in order to fill that role well. Likewise in Jumbo. The more valuable a trade is perceived to be by the gloms involved, the more urgent will be the corresponding action codelet manufactured by the musing codelet.

Thus the principle of the terraced scan is at work at this stage of the game no less than in the earlier, glomming-together stages.

FLUID DATA STRUCTURES: A PRINCIPAL AIM OF JUMBO

The combination of regrouping and rearrangement provides a surprisingly powerful tool. It preserves a great deal of structure at all times, and yet achieves a significant reshuffling of high-level pieces. Here is a random walk through pseudo-word space, beginning at "pang-loss":

pang-loss	start
pong-loss	kniferism
long-pass	spoonerism
pass-long	switch syllables
pas-slong	regroup "s"
sap-slong	reverse syllable
slap-song	spoonerism
slang-sop	forkerism
sop-slang	switch syllables
slop-sang	spoonerism
slos-pang	trade "p" for "s"
los-spang	spoonerism
loss-pang	regroup "s"
pang-loss	switch syllables
pan-gloss	regroup "g"

You can continue with this process for a long time.

One might suppose this process to be idiosyncratic to the domain, an obscure type of transformation having little relevance to general intelligence. Such an impression would, however, be quite wrong. Recall that Jumbo's deep aim is to aid in demonstrating the thesis "cognition equals recognition". Now any study of recognition reveals the indescribably fluid boundaries of percepts and concepts. There are thousands of ways of drawing the letter "A" (see HOF-6), yet we recognize any of them at a glance. For any mental category (concrete ones like "dog" and abstract ones like "metaphor"), we immediately recognize thousands of instances, despite vast differences.

To recognize -- i.e., to match an item with a Platonic abstraction -- there must be room for flexing both the Platonic abstraction and the mental representation of the thing itself, to make them meet. What kinds of distortion are sensible? Readjustment of internal boundaries, reassessment of connection strengths, and rearrangement of parts are at the core of the recognition process. Thought's fluidity emanates from the nonrigidity -- the fluidity -- of representational structures. They have an ability to adjust, to alter themselves effortlessly, to assume many conformations. For fluidity to emerge at a high level, the underlying data structures must be permeated by an ability to slip back and forth at the drop of a hat.

It is important that this capacity reside inherently inside the structure itself: it must be ready and willing to be internally reconformed if you just "tap" it the right way. The term "change of representation", used in some AI projects, usually refers to a "reasoning engine" converting one representation into another representation. With Jumbo, by contrast, reconformability is inherent in the structures themselves. Perhaps the major part of the intelligence of the system as a whole resides in the hinge-points of its data structures. The import of that key idea goes far beyond the confines of this one domain.

It is my belief that a nearly constant background activity of creative minds is a playful twisting-around of mental structures of all sorts -- turning them inside out or backwards, regrouping parts, erasing or inserting new levels of structure, moving things back and forth between levels, and so on. In short, taking full advantage of the fluidity of their mental representations, and simply being sensitive to unanticipated side effects.

ENTROPY-INCREASING TRANSFORMATIONS

If Jumbo deems its single top-level glom unsatisfactory, its first attempted remedy will be to create regrouping and rearrangement codelets and to give them high urgencies. Thus, a bit of shuffling-about as illustrated above will take place. During this, Jumbo will monitor the overall happiness of the cytoplasm. If after a while that happiness has not reached a satisfactory level, Jumbo will alter its priorities and pour disbanding-codelets into the coderack.

An example of disbanding would be: smash the word candidate into its syllables, then smash those syllables further into their cluster-level constituents. In the case of "pang-loss", this yields: "p", "a", "ng", "l", "o", and "ss". Now one may hope that some fresh gloms will bubble up. If not, one can disband further!

Action codelets for disbanding are called "dissolvers". A disbanding operation is not undertaken lightly, since it risks destroying a happy glom. Therefore, a musing codelet must check the idea out first, and if it approves of the disbanding, it creates a dissolver -- but if there is hope that some gentler remedy might make the cytoplasm happier, it will assign the dissolver a low urgency.

There are two sorts of dissolvers: targeted and nontargeted. The former is aimed at a specific glom that has given trouble for a while. The latter is a beast whose mission is to find ANY glom and tear it into its immediate constituents.

Why nontargeted dissolvers? Well, if you have been troubled by a glom that has been unfruitful no matter how you shuffled it about, one way to restart

quickly would be to load up the coderack with a bunch of nontargeted dissolvers. In a flash, they, not unlike a horde of hungry piranhas, will tear that pesky glom from limb to limb, leaving you to start out again with a totally disordered cytoplasm.

There is a problem here. Suppose you put ten nontargeted dissolvers in the coderack, and after just five have run, the cytoplasm has been reduced to raw letters. Five nontargeted dissolvers remain in the coderack. What's to prevent them from later being called upon to run, and at that time totally ruining the new emerging order?

TEMPERATURE AND SELF-WATCHING

One way is to have a global variable called the "temperature" of the cytoplasm, low when the top-level gloms are happy, and high when they are unhappy. The happiest state is when there is just one single happy glom. Since we want to preserve such a state, this state is assigned the "freezing" temperature. When the cytoplasm is freezing, dissolving is prevented.

When the cytoplasm is boiling, nontargeted dissolvers have permission to tear apart any glom, whereas when it is merely warm, they are inhibited (i.e., have a low probability of carrying out their mission) in proportion to the happiness of the gloms they are attacking. The cytoplasmic temperature is thus Jumbo's equivalent to the "mood" of society regarding divorce. High temperature encourages split-ups, low temperature discourages them.

Cytoplasmic temperature is a very general concept. It provides quick global feedback about the system, helping to dynamically govern the probabilities of various kinds of activity. The temperature regulates the degree of chaos in the system, something like the rods in a nuclear reactor. When things start to get too wild, you lower the temperature a bit, and when things seem to be going nowhere, you raise it and pump in some craziness [Smolensky 1983].

Temperature will be useful only if kept updated. If not assessed frequently enough, its outdatedness could be harmful. Now temperature-measurement codelets must vie for recognition in the coderack along with everybody else. This brings up the tricky question of how the system keeps itself balanced, with so many complex things happening at once, all mingling in the same coderack.

How can loop-like behavior be obviated? One way is for the system to watch its own cytoplasm and coderack [Hofstadter 1982a], and to make sure that if any loopiness is detected, preventive actions are taken. Such simple remedies as raising or lowering the temperature may suffice, as would changing the values of other important global variables.

But how does self-monitoring really take place? This is as yet unexplored territory in Jumbo. The general idea is that there should be a way of summarizing the contents of the cytoplasm and coderack in simple terms, so that no detailed self-scrutiny is needed. For the coderack, a good start might involve a "census" detailing what kinds of codelets are present, and for each type, giving its percentage of the total urgency. For the cytoplasm, listing the top-level gloms' appearances (letter sequences) might be a good beginning. That way, periodic and quasi-periodic assembly and disassembly of structures would be easily detectable. But again, the tricky problem of self-reference arises: cytoplasm-watchers and coderack-watchers must themselves be codelets, vying like all codelets for time!

These twisty ideas are not yet implemented. Clearly they will be significant in determining how Jumbo tunes itself and thus controls itself dynamically. Such feedback mechanisms offer the potential for great flexibility and power, but they are also very tricky. They are reminiscent, in ways, of human consciousness, an alluring mystery towards which AI is slowly moving [Hofstadter 1979, 1981a, 1982d, Smith 1982]. Mechanisms for self-watching are therefore something to look forward to in the developing architecture of Jumbo.

A SELF-SENSITIVE, SELF-DRIVEN SYSTEM

In Jumbo, there is no top-level "driver" making decisions about what to do next. The flow is a function of weights and random numbers, not of careful consideration and reflection at each microscopic branch-point. This makes Jumbo sound stupider than it is. In life, you often must make a decision randomly in order not to get bogged down in an infinite regress. At some level, reflection must give way to reflex. But this need not be disastrous, provided your reflexes are well-tailored ones. That is the purpose of the weights -- urgencies -- attached to codelets, and of the measures of happiness.

Together, urgencies and happinesses guide Jumbo's processing. Clearly, the flow of control, to the extent that it is determinate, is determined by urgency values. What determines those values? Some processes, such as spark-generating codelets, have fixed urgencies. Others, such as codelets for spark-evaluating, flash-evaluating, and bonding, have variable urgencies. The variation comes from the perceived mutual suitedness of the gloms involved. This in turn is determined by the INTRINSIC AFFINITY of the structures involved -- a fixed feature of English -- and their HAPPINESS. Thus happinesses and affinities determine the urgency values for bonding- and disbanding-codelets.

In this way, happinesses and urgencies are intertwined, and they combine to yield a total outcome that is the overall, observable behavior of the system. Jumbo thus emerges as an exquisitely self-sensitive collection of processes: a coherent whole composed of many independent interwoven strands. (See the "Prelude...Ant Fugue" and comments in [Hofstadter 1979] or [Hofstadter 81a]. See also [Rumelhart 1982], [Hinton 1981], especially Chapter 1, and [Feldman 1982].)

EPILOGUE: JUMBO'S (EPI)PHENOMENAL INTELLIGENCE

Jumbo's intelligence, if indeed it has any, clearly has not been directly programmed; rather, it emerges as a statistical consequence of the way that many small things that one HAS designed interact with each other. It is like the chess program that "likes to get its queen out early", surprising its programmer, who never explicitly put such a strategic concept into the program. Daniel Dennett calls this an "innocently emergent" quality [Dennett 1977]; I call it an "epiphenomenon" [Hofstadter 1979, 1982c, 1982d]. One could therefore say that Jumbo involves a third metaphor: that of statistical mechanics, which explains how macroscopic order can emerge from microscopic chaos.

This view of intelligence goes against the grain of traditional AI work, which seeks to find explicit (nonstatistical) rules governing the flow of thoughts. The hope is to describe thoughts at their own level without having to resort to describing biological underpinnings of them. It is an admirable dream, but it is not a priori obvious that such laws exist.

I call this the "Boolean Dream" after George Boole who set forth an elegant calculus called "The Laws of Thought" [Boole 1855], which we know as Boolean algebra. Is the Boolean dream realizable? Most AI people seem to hope so; some even feel that a claim to the contrary is an antiscientific claim. I see things differently. My view is that AI and the Boolean dream are closely related but not synonymous. Thus the perishing of the Boolean dream would not be a setback to AI, but simply a recognition that some high-level mental phenomena are innocently emergent. The goal of AI would then shift from trying to capture the "laws of thought" in an elegant mathematical formalism to trying to understand the laws governing subcognitive events (let us call such events "mentals") that collectively go to make up cognition, or the dynamics of thinking.

To sum matters up, then, we could say that the shifted burden of AI should eventually become the clarification of the meaning of the following corny but hopefully catchy "equation":

thinkodynamics = statistical mentalics.

O, a noble dream!

REFERENCES

- Arnold, Henri and Bob Lee. "Jumble #21". Signet (New American Library), 1982.
- Bergerson, Howard. "Palindromes and Anagrams". Dover, 1973.
- Bobrow, Daniel G. and Bertram Raphael. "New Programming Languages for AI Research". ACM Computing Surveys, Vol. 6, No. 3, September 1974.
- Bongard, Mikhail. "Pattern Recognition". Hayden Book Co. (Spartan Books), 1970.
- Boole, George. "The Laws of Thought" (1855). Dover.
- Dennett, Daniel C. "Brainstorms". Bradford, 1977.
- Erman, Lee D., Frederick Hayes-Roth, Victor R. Lesser, and D. Raj Reddy. "The Hearsay-II Speech-Understanding System: Integrating Knowledge to Resolve Uncertainty". ACM Computing Surveys, Vol. 12, No. 2, June 1980, pp. 213-253.
- Feldman, J. A. and D. H. Ballard. "Connectionist Models and Their Properties". Cognitive Science, Vol. 6, No. 3, July-September, 1982, pp. 205-254.
- Hanson, A. and E. Riseman, eds. "Computer Vision Systems". Academic Press, 1978.
- Hinton, Geoffrey E. and James A. Anderson, eds. "Parallel Models of Associative Memory". Lawrence Erlbaum Associates, 1981.
- Hofstadter, Douglas R. "Gödel, Escher, Bach: an Eternal Golden Braid". Basic Books, 1979.
- Hofstadter, Douglas R. and Daniel C. Dennett, eds. "The Mind's I". Basic Books, 1981.
- Hofstadter, Douglas R. "Metamagical Themas: Roles and Analogies in Human and Machine Thought". Scientific American, Vol. 245, No. 3, September, 1981, pp. 18-30.
- Hofstadter, Douglas R. "Metamagical Themas: On Spheishness, 'Jootsing', and Creativity". Scientific American, Vol. 247, No. 3, September, 1982, pp. 18-34.
- Hofstadter, Douglas R. "Metamagical Themas: On Variations on a Theme, Slippability, and Creativity". Scientific American, Vol. 247, No. 4, October, 1982, pp. 20-29.
- Hofstadter, Douglas R. "Artificial Intelligence: Subcognition as Computation", in F. Machlup and Una Mansfield (eds.), "The Study of Information", to be published in 1983 by John Wiley and Sons. Also Indiana University Computer Science Department Technical Report #132, November, 1982.

- Hofstadter, Douglas R. "Who Shoves Whom Around Inside the Careenium?" *Synthese*, Vol. 53, No. 2, November, 1982, pp. 189-218. Also Indiana University Computer Science Department Technical Report #130, July 1982.
- Hofstadter, Douglas R. "Metafont, Metamathematics, and Metaphysics: Comments on Donald Knuth's Article 'The Concept of a Meta-Font'". *Visible Language*, Vol. XVI, No. 4, Autumn, 1982, pp. 309-338. Also Indiana University Computer Science Department Technical Report #136, December, 1982.
- Hofstadter, Douglas R. "On Seeking Whence", forthcoming Indiana University Computer Science Department Technical Report, 1983.
- Holland, John H. "Genetic Algorithms and Adaptation". University of Michigan Department of Behavioral Science Technical Report #34, December 1981.
- Lea, W. A., ed. "Trends in Speech Recognition". Prentice-Hall, 1980.
- Lehninger, Albert. "Biochemistry" (2nd edition). Worth Publishers, 1975.
- Reddy, D. Raj, et al. "Working Papers in Speech Recognition, IV: The HEARSAY II System". Carnegie-Mellon University Computer Science Department Technical Report, February, 1976.
- Rumelhart, David E. and Donald A. Norman. "Simulating a Skilled Typist: A Study of Skilled Cognitive-Motor Performance". *Cognitive Science*, Vol. 6, No. 1, January-March, 1982, pp. 1-36.
- Schank, Roger C. "Language and Memory". *Cognitive Science*, Vol. 4, No. 3, July-September, 1980, pp. 243-284.
- Simon, Herbert A. "Studying Human Intelligence by Creating Artificial Intelligence" (1980 Procter Lecture). *American Scientist*, Vol. 69, No. 3, May-June, 1981, pp. 300-309.
- Smith, Brian C. "Reflection and Semantics in a Procedural Language". Massachusetts Institute of Technology, Laboratory for Computer Science Technical Report #272, 1982.
- Smolensky, Paul. "Harmony Theory: A Mathematical Framework for Learning and Parallel Computation". Unpublished manuscript, Center for Human Information Processing, University of California at San Diego, 1983.
- Waterman, D. A. and Frederick Hayes-Roth, eds. "Pattern-Directed Inference Systems". Academic Press, 1978.

AN APPROACH TO LEARNING FROM OBSERVATION

Gerald DeJong
 Coordinated Science Laboratory
 University of Illinois at Urbana-Champaign
 1101 W. Springfield Avenue
 Urbana, IL 61801

ABSTRACT

Learning from observation requires a system to appreciate the significance of an event (or set of events) which are either fortuitous or part of another's planning process. Then the new events must be generalized into a new concept. We present a framework in which these processes can occur. The approach is called explanatory schema acquisition. It involves knowledge-based generalization which can construct a first-pass generalized concept from just one input example.

Key terms: Schema learning, machine learning, knowledge acquisition, artificial intelligence, natural language processing, induction.

I. INTRODUCTION

There are two problems that the generalization process must face. The first is to know when it should be applied. Clearly, every input text ought not to cause the system to construct a new schema. Only "interesting" inputs should invoke the schema acquisition system. The second problem is how to perform the generalization. There are a number of subproblems here, for example, selecting which events and objects should be generalized, imposing limits on the extent of generalization, and actually carrying out the schema construction.

Briefly, the approach can be summarized as 1) understanding the input, 2) evaluating the five generalization criteria outlined above, and 3) possibly generalizing the input to a new schema based on the evaluations of (2). The understanding phase is not significantly different from most planning-type understanders [Schmidt & Sridharan, 1977; Cullingford, 1978; DeJong, 1979; Kolodner, 1980; Lebowitz, 1980] with one exception. The understander maintains data dependency supports [Fikes, 1975; Doyle, 1979] for all of the inferences necessary to make sense of the input. This dependency network drives the generalization process. The particular input representation is generalized as far as possible without violating the dependency support structure. Violating the dependency support structure would undermine the reasons the system had in believing the story

makes sense. Usually, input can be generalized in a number of different ways. The particular generalization selected is determined by fitting the input into a taxonomy of generalization techniques.

II. TO GENERALIZE OR NOT TO GENERALIZE

There are five aspects to be considered when deciding whether or not to generalize an input into a new schema. By hypothesis we will assume that the input did not match an existing schema (if it had then the system already possesses the desired schema and indeed that schema would have been used to process the story). If any of these five conditions does not hold, constructing a new schema from this input is inappropriate.

The criteria are:

- 1) Is the main goal of a character achieved?
- 2) Is the goal a general one?
- 3) Are the resources required by the goal achiever generally available?
- 4) Is this new method of achieving the goal at least as effective as the other known volitional schemata to achieve this goal?
- 5) Does the input match one of the known generalizable patterns?

These criteria are tested for all goals in the story. The first criterion "Was the goal achieved?" is self explanatory and easily judged. The second "Is it a general goal?" and the third "Are the resources generally available?" require some discussion.

Novelty alone in an approach to achieving a goal is not sufficient to warrant constructing a new schema. Consider, for example, a plot from the "Mission Impossible" television series. These plots are noteworthy in that they are very novel. They all use bizarre methods to achieve rather peculiar goals. Furthermore, they are always successful. However, the goals achieved are not the type that arise in ordinary life and the resources and skills needed are so specialized and uncommon that the same solution would never be applicable again.

Clearly a new schema should be constructed only if there is a reasonable expectation that it will be helpful in future processing. If a schema will never be used again, it should not be constructed in the first place.

How can the utility of a particular goal be judged? The answer to this is closely tied to where goals come from. Achieving a goal which arises from general conditions important to an individual's well-being and using readily available resources is likely to result in an interesting new schema, one which will arise again and again. For the solution, we use an aspect of Schank & Abelson's theory of planning [Schank & Abelson, 1977]. In their view themes give rise to the highest level goals (goals which are not simply subgoals in the achievement of other goals). Interpersonal and Life themes are what we are interested in. An example of the former is a husband offering (and therefore, at some level, wanting) to type a term paper for his over-worked student wife. This exemplifies the theme of taking on goals of others that one cares for. It is not necessary to go beyond this theme in explaining the husband's actions. Examples of life themes are attempting to satisfy one's hunger, to gain money, or to relieve boredom. Themes give rise to goals that require no further justification.

The fourth criterion is self-explanatory. The idea is that the system should not bother constructing schemata that are much less efficient than similar already-known schemata. In a natural language input this would occur only if a character were using a highly sub-optimal plan.

The fifth criterion is that the input must be identified within the taxonomy of generalization techniques. This is discussed in the next section.

III. TAXONOMY OF GENERALIZATIONS

There are four situations which when recognized in the text either individually or in combination ought to invoke the generalization routines. They are:

- Schema Composition
- Secondary Effect Elevation
- Schema Alteration
- Volitionalization

In the first part of this section we will illustrate each of these situations with an example.

3.1. Schema composition

The first situation we will discuss is called schema composition. Basically, it involves composing known schemata in a novel way. Typically, this will involve a primary schema, essentially unchanged, with one or more of its preconditions satisfied in a novel way by other known schemata.

To clarify the procedure, consider an example. This example is a story about a kidnapping. Let us assume that we, the readers of this example, do not yet have a schema for kidnapping or extortion or any similar notion. We do, however, assume the knowledge of a considerable quantity of background information about stealing, bargaining, the use of normal physical objects, and goals of people and institutions.

Paris police disclosed Tuesday that a man who identified himself as Jean Maraneaux

abducted the 12 year old daughter of wealthy Parisian businessman Michel Boullard late last week. Boullard received a letter containing a snapshot of the kidnapped girl. The next day he received a telegram demanding that 1 million francs be left in a lobby waste basket of the crowded Pompidou Center in exchange for the girl. Asking that the police not intervene, Boullard arranged for the delivery of the money. His daughter was found wandering blindfolded with her hands bound near his downtown office on Monday.

In this story the primary schema is BARGAIN, a schema which we assumed the system already knew. One of the preconditions specified in the BARGAIN schema is that each party to the bargain must convince the other that he can indeed deliver his side of the bargain. For Maraneaux, this corresponds to making Boullard believe that he (Maraneaux) has control of Boullard's daughter and can, therefore, relinquish the girl to him. Maraneaux achieves this by actually establishing control over the daughter (via an instance of the STEAL schema) and then sending Boullard a photograph. To the system, this is a novel way to satisfy BARGAIN's preconditions.

3.2. Secondary effect elevation

Consider the following scenario:

Fred wanted to date only Sue, but Sue steadfastly refused his overtures. Fred was on the verge of giving up when he saw what happened to his friend, John: John wanted to date Mary but she also refused. John started seeing Wilma. Mary became jealous and the next time he asked her, Mary eagerly accepted. Fred told Sue that he was going to make a date with Lisa.

Here Fred has not acquired a new schema; he has used an existing schema (DATE) in a new way. This is called secondary effect elevation. Fred's DATE schema already contains all of the knowledge necessary for resolving his dilemma. The problem is that the normal DATE schema is organized in the wrong way. In secondary effect elevation situations an existing schema is annotated indicating that the schema may be used to achieve a result which is normally neutral or negative.

The main purpose of the DATE schema is to satisfy certain recurring social goals (like companionship, sex, etc.). DATE contains secondary effects as well. These are often undesirable effects accompanying the main, planned effects. For example, one is usually monetarily poorer after a date. Another secondary effect is that if one has an old girlfriend, she may become jealous of a new date.

What Fred learned from John's experience is that it is occasionally useful to invoke the DATE schema in order to cause one of its secondary effects (jealousy) while completely ignoring the usual main goal.

Just as with schema composition, the existing schema is changed to reflect a generalization made from a specific instance. In this case, the

specific instance is John's interactions with Mary. Notice, however, that Fred did not simply copy John's actions. John actually made a date with Wilma while Fred only expressed an intention to date Lisa. This is not an earth-shaking difference, but in the context of dating it is extremely significant. In the normal DATE situation expressing an intention to date someone is not nearly so satisfying as an actual date. Once modified for the purpose of causing jealousy, however, expressing an intention for a date and actually carrying it out can be equally effective.

One might argue that the distinction between main and secondary effects of a schema is otiose and, in situations such as this, even deleterious. After all, DATE already had all of the information necessary for solving Fred's problem. If a system simply treats all of the effects of a schema the same, then any effect can be singled out during the planning process to be used as the main goal. There is, however, a strong argument against this position. The possible desired effects of a schema do not exist only within the schema itself. They are used to organize and select among schemata in both understanding and planning applications. Many effects (like feeling more tired after a date than before) will not be used in the normal planning or understanding process. If they are treated the same as legitimate main goals the system will be swamped in a combinatorial quagmire of undifferentiated possibilities, most of which are wildly implausible. For example, we do not want our understanding process to predict that John will take a nap when it is told that John dated Mary. Given the input "John took a nap" the system ought to be able to justify it. However, it ought not actively predict it. Given the multiplicity of individual actions making up the DATE schema (each with its own set of effects) the vast majority of the effects from this schema (and any other schema) are simply irrelevant to overall planning and understanding processes. Instead, we would like our system to single out the plausible volitional effects of its schemata and use only those for schema organization and selection. Thus, in our example, Fred has constructed, via secondary effect elevation, a new use of the DATE schema.

3.3. Schema alteration

Schema alteration involves modifying a nearly correct schema so that it fits the requirements of a new situation. The alteration process is guided by the system's world model. This is illustrated by the following brief anecdote:

I had occasion to replace temporarily a broken window in my back door with a plywood panel. The plywood sheet from which the panel was to be cut had a "good" side and a "bad" side (as does most raw lumber). The good side was reasonably smooth while the bad side had several ruts and knot holes. I automatically examined both sides of the sheet (presumably as part of my SAWING or CUTTING-A-BOARD-TO-FIT schema) and selected the good side to face into the house with the bad side to be exposed to the elements. After I had

cut the panel and fitted it in place I noticed that several splinters had been torn out leaving ruts in the "good" side. I immediately saw the problem. Hand saws only cut in one direction. With hand saws, the downward motion does the cutting while the upward motion only repositions the cutting blade for another downward motion. I had cut the wood panel with the "good" side facing down. The downward cutting action has a tendency to tear splinters of wood out of the lower surface of the board. Since the good side was the lower surface, it suffered the loss of splinters. If I had to perform the same action again, I would not make the same mistake. I would cut the board with the good side facing up. However, what I learned was not just a simple specialized patch to handle this particular instance of splintering. Since I knew the cause of the splintering, I knew that it would not always be a problem: it is only a problem when 1) the lumber is prone to splintering, 2) there is a "good" side of the board that is to be preserved, and 3) one is making a crosscut (across the wood's grain) rather than a rip cut (along the grain). Moreover, the solution is not always to position the wood with the good side up. My electric saber saw (also a reciprocating saw) cuts during the upward blade motion rather than the downward motion. Clearly, the solution when using the saber saw is the opposite: to position the board with the good side down. Now, these are not hard and fast rules: with a sufficiently poor quality sheet of plywood splintering would likely always be a problem. Rather, these are useful heuristics that lead to a refinement of the SAWING schema.

Note that this refinement to the SAWING schema is far more general than required to handle the particular problem that gave rise to it. The refinement contains contingencies relevant to the use of saber saws even though no saber saw was used in the immediate problem. This is possible because the refinement is driven by world model, not just the problem. The SAWING schema was altered by identifying and eliminating the offending cause in the underlying knowledge-based explanation of the phenomena.

3.4. Volitionalization

This situation involves transforming a schema for which there is no planner (like VEHICLE-ACCIDENT, ROULETTE, etc.) into a schema which can be used by a planner to attain a specific goal. Consider the following story:

Herman, who was married to Joy for 15 years had fallen in love with his secretary, Heather. When Joy refused to divorce him, Herman cut a hydraulic brake hose in Joy's car. The next time she drove down the winding road to town she lost control of her car and struck a tree. The car burst into flames.

This story describes a vehicle accident. However, unlike most vehicle accidents, this one has an active agent. The VEHICLE-ACCIDENT schema is normally non-volitional. That is, it dictates what to expect in a vehicle accident situation, but it does not allow planned invocation. This schema cannot be involved in problem-solving planning.

The story illustrates that an event which was previously only attributable to accidental causes can, in fact, be controlled. Thus, a volitional counterpart can be constructed for the previously non-volitional schema. This new volitional schema has certain effects that can be labeled as its main goals, new constraints on the actors and objects, etc. It is, in fact, a new schema in its own right. Furthermore, using the underlying explanation of the story to drive the generalization process, the schema can encompass situations significantly different from the one described in the story. For example, it should handle a story about a student weakening the steering rod of a professor's car after receiving a failing grade.

IV. AN EXAMPLE

Here, we give an example of the first-pass implementation of the system. It illustrates a new schema acquired by volitionalization. The system does not have a natural language front end. The inputs are given in conceptual form. These are denoted by lines starting with "Processing---...". There are just four inputs. English versions of the inputs are given below. For readability, this output is a slightly altered form of the program's output. The system generates many more GENSYM atoms and refers to them by their GENSYM names. The GENSYMs have been replaced with mnemonic symbols.

ENGLISHIFIED INPUT:

- 1) CLAUDIUS OWNED AN ISLAND ESTATE.
- 2) AGRIPPINA FED CLAUDIUS POISONED MUSHROOMS.
- 3) CLAUDIUS DIED.
- 4) AGRIPPINA INHERITED THE ISLAND ESTATE.

The first input

Processing---(\$POSS (ACTOR *CLAUDIUS*)
(OBJECT *ESTATE*))

New Schema---S00001 (\$POSS (ACTOR *CLAUDIUS*)
(OBJECT *ESTATE*))

Setting bindings and links

New Schema---S0000 (\$DESIRE-FOR-MATERIAL-THINGS
(ACTOR *CLAUDIUS*) (OBJECT *ESTATE*))
Setting bindings and links

The second input

Processing---(\$FEED (ACTOR *AGRIPPINA*)
(OBJECT *POISON* TYPE *MUSHROOM*))

New Schema---S00003 (\$FEED (ACTOR *AGRIPPINA*)
(OBJECT *POISON*) (TO *CLAUDIUS*))

Setting bindings and links

New Schema---S00005 (\$EAT (ACTOR *CLAUDIUS*)
(OBJECT *POISON*))

Activated by S00003

Setting bindings and links

\$EATING poison brings in the poison schema. We call it NAIVE POISON to remind ourselves that it is an incomplete schema containing little more than what is needed for this story.

New Schema---S00006 (\$NAIVE-POISON (ACTOR
AGRIPPINA) (INSTRUMENT *MUSHROOM*)
(OBJECT *CLAUDIUS*))

Activated by S00003

Setting bindings and links

New Schema---S00007 (\$MOVE (ACTOR *AGRIPPINA*)
(OBJECT *MUSHROOM*) (FROM (OUTSIDE *CLAUDIUS*)
(TO (INSIDE *CLAUDIUS*))))

Activated by S00006

Setting bindings and links

New Schema---S00008 (\$BODILY-HARM (OBJECT
CLAUDIUS))

Activated by S00006

Setting bindings and links

New Schema---S00009 (\$HEALTH (CREATURE
CLAUDIUS) (STATE (*VAR* S1)))

Activated by S00008

Setting bindings and links

New Schema---S00010 (\$INGEST (ACTOR *AGRIPPINA*)
(OBJECT *POISON*) (TO *CLAUDIUS*))

Activated by S00006

Setting bindings and links

This \$DEATH is predicted by the poisoning

New Schema---S00011 (\$DEATH (OBJECT *CLAUDIUS*))

Activated by S00006

Setting bindings and links

Old Schema---S00009 (\$HEALTH (CREATURE
CLAUDIUS) (STATE *ALIVE*))

Activated by S00011

Setting bindings and links

New Schema---S00012 (\$HEALTH (CREATURE
CLAUDIUS) (STATE *DEAD*))

Activated by S00011

Setting bindings and links

Alternatively, Claudius might only become ill.

This is also predicted.

New Schema---S00013 (\$NAUSEA (ACTOR *CLAUDIUS*))

Activated by S00006

Setting bindings and links

If Claudius dies, Agrippina might have planned his death.

New Schema---S00014 (\$NAIVE-REMEDITATED-MURDER
(ACTOR *AGRIPPINA*) (OBJECT *CLAUDIUS*))

Activated by S00006

Setting bindings and links

Old Schema---S00011 (\$DEATH (OBJECT *CLAUDIUS*))

Activated by S00014

Setting bindings and links

Old Schema---S00006 (\$NAIVE-POISON (ACTOR
AGRIPPINA) (INSTRUMENT *MUSHROOM*) (OBJECT
CLAUDIUS))

Activated by S00014

Setting bindings and links

New Conjunction Node---S00015 (AND (\$DEATH
(ACTOR *CLAUDIUS*)) (NOT (\$PUNISH (OBJECT
AGRIPPINA))))

Old Schema---S00011 (\$DEATH (OBJECT *CLAUDIUS*))

The third input

Processing---(\$DEATH (OBJECT *CLAUDIUS*))

Old Schema---S00011 (\$DEATH (OBJECT *CLAUDIUS*))

This matches the dying prediction. Thus, very little processing must be done.

The fourth input

Processing---(\$NAIVE-INDIVIDUAL-INHERIT
(HEIR *AGRIPPINA*) (BEQUESTS *ESTATE*))

New Schema---S00018 (\$NAIVE-INDIVIDUAL-INHERIT
(HEIR *AGRIPPINA*) (BEQUESTS *ESTATE*))

Setting bindings and links

Old Schema---S00001 (\$POSS (ACTOR *CLAUDIUS*)
(OBJECT *ESTATE*))

Activated by S00018

Setting bindings and links

Old Schema---S00011 (\$DEATH (OBJECT *CLAUDIUS*))

Activated by S00018

Setting bindings and links

New Schema---S00021 (\$HEALTH (CREATURE
AGRIPPINA) (STATE *ALIVE*))

Activated by S00018

Setting bindings and links

Old Schema---S00012 (\$HEALTH (CREATURE
CLAUDIUS) (STATE *DEAD*))

Activated by S00018

Setting bindings and links

New Schema---S00022 (\$TRANSFER-OF-POSSESSION
(FROM *CLAUDIUS*) (TO *AGRIPPINA*) (OBJECT
ESTATE))

Activated by S00018

Setting bindings and links

Old Schema---S00001 (\$POSS (ACTOR *CLAUDIUS*)
(OBJECT *ESTATE*))

Activated by S00022

Setting bindings and links

New Schema---S00025 (\$POSS (ACTOR *AGRIPPINA*)
(OBJECT *ESTATE*))

Activated by S00022

Setting bindings and links

Finally, the system has a motivation for the murder.

New Schema---S00026 (\$MATERIAL-DESIRE (ACTOR
AGRIPPINA) (OBJECT *ESTATE*))

Activated by S00025

Setting bindings and links

VOLITIONALIZATION TRIGGERED

NEW SCHEMA V00035:

(FORM SCHEMA

(VARS (BENEFACITOR (*VAR* V00031)

PRIOR-HEIRS (*VAR* V00033)

BEQUESTS (*VAR* V00029)

HEIR (*VAR* V00030)

ACTOR (*VAR* V00030)
OBJECT (*VAR* V00031))
(POINT-OF-VIEW (*VAR* V00030))
(GOAL V00032)

(ACTIVATE

(V00032 (\$POSS (ACTOR (*VAR* V00030))
(OBJECT (*VAR* V00029)))

V00033

(\$NAIVE-INDIVIDUAL-INHERIT

(BENEFACITOR (*VAR* V00031))

PRIOR-HEIRS (*VAR* V00033)

BEQUESTS (*VAR* V00029)

HEIR (*VAR* V00030))

V00034

(\$NAIVE-PREMEDITATED-MURDER ACTOR

(*VAR* V00030)

OBJECT (*VAR* V00031))

(COMPLEX (V00033 V00034))))

V. CONCLUSION

The example shows a new schema acquired by the system. The form of this schema is identical to those pre-programmed into the system. The only difference is in the mnemonic naming of variables. There are several problems with this implementation which are the subject of current research. First, there is an inadequate treatment of time. The system assumes (whenever it cares) that the inputs are presented in the same order as the events they report. The system performs (through schema expansion) a good deal of causal analysis. This also implies some temporal orderings. However, by and large, difficult time problems do not surface for this simple story. A second problem is with variable constraints. This implementation does very little reasoning about objects. Most of its generalization is performed on constituent events rather than the objects participating in those events. These generalizations (such as generalizing POISON to MURDER) seemed less obvious and therefore more interesting. Furthermore, there has been previous work on generalizing objects [Fikes, 1972]. Finally, there is a good deal of work to be done on goal manipulation of story characters and on schema representation. The representations used here are ad hoc and cannot easily handle the next task we have chosen (a sequence of kidnapping stories).

Unlike more correlational approaches [Michalski, 1977; Fox & Reddy, 1977; Langley, 1981], the procedure is capable of constructing a schema from one input example. Nor is the procedure "failure-driven" [Kolodner, 1980; Lebowitz, 1980; Schank, 1982]. Nor is it primarily analogical [Winston, 1980; Carbonell, 1982]. The LEX system [Mitchell, Utgoff & Banerji, 1983] is also a significantly different approach. In that system concepts are acquired that are judged to be useful through search and experimentation not through an analysis of why an operator was useful. In explanatory schema acquisition concepts are formed entirely on the basis of this kind of analysis. The knowledge-based concept approach owes much to Soloway [1977] and Lenat [1976].

ACKNOWLEDGMENT

I am indebted to the members of the CSL learning group: Chris Debrunner, Paul Harrington,

Paul O'Rourke, and Alberto Segre. Paul O'Rourke implemented the system that performed the example.

REFERENCES

- Carbonell, J., "Experiential Learning in Analogical Problem Solving," Proc. Nat'l. Conf. on Artificial Intelligence, Pittsburgh, PA, 1982.
- Cullingford, R., "Script Application: Computer Understanding of Newspaper Stories," Research Rept. 116, Yale Computer Science Dept., New Haven, CT, 1978.
- DeJong, G., "Prediction and Substantiation: A New Approach to Natural Language Processing," Cognitive Science, vol. 3, pp. 251-273, 1979.
- Doyle, J., "A Truth Maintenance System," Artificial Intelligence, vol. 12, no. 2, pp. 231-272, 1979.
- Fikes, R., "Deductive Retrieval Mechanisms for State Description Models," Proc. 4th IJCAI, pp. 99-106, 1975.
- Fikes, R., Hart, P. and Nilsson, N., "Learning and Executing Generalized Robot Plans," Artificial Intelligence, vol. 3 no. 2, pp. 251-288, 1972.
- Fox, M. and Reddy, R., "Knowledge-Guided Learning of Structural Descriptions," Proc. 5th IJCAI, Cambridge, MA, 1977.
- Kolodner, J., "Retrieval and Organizational Strategies in Conceptual Memory: A Computer Model," Computer Science Research Rept. 187, Ph.D. dissertation, Yale Univ., New Haven, CT, 1980.
- Langley, P., "Data-Driven Discovery of Physical Laws," Cognitive Science, vol. 5, pp. 31-54, 1981.
- Lebowitz, M., "Generalization and Memory in an Integrated Understanding System," Computer Science Research Rept. 186, Ph.D. dissertation, Yale Univ., New Haven, CT, 1980.
- Lenat, D., "AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search," AIM-286, AI Laboratory, Stanford Univ., Stanford, CA, 1976.
- Michalski, R., "A system of Programs for Computer Aided Induction: A Summary," Proc. 5th IJCAI, Cambridge, MA, pp. 319-320, 1977.
- Mitchell, T., Utgoff, P. and Banerji, R., "Learning by Experimentation: Acquiring and Refining Problem Solving Heuristics," in Machine Learning, R. Michalski, J. Carbonell, & T. Mitchell (Eds.), Tioga, Palo Alto, CA, 1983.
- Schank, R., Dynamic Memory, Cambridge Univ. Press, Cambridge, 1982.
- Schank, R. and Abelson, R., Scripts Plans Goals and Understanding, Lawrence Erlbaum Associates, Hillsdale, NJ, 1977.
- Schmidt, C. and Sridharan, N., "Plan Recognition Using a Hypothesize and Revise Paradigm: An Example," Proc. 5th IJCAI, Cambridge, MA, pp. 480-486, 1977.
- Soloway, E., "Knowledge Directed Learning Using Multiple Levels of Description," Ph.D. dissertation, Computer Science Dept., Univ. of Massachusetts, Amherst, MA, 1977.
- Winston, P., "Learning and Reasoning by Analogy," Commun. ACM, vol. 23, no. 12, 1980.

CONCEPT LEARNING IN A RICH INPUT DOMAIN¹

Michael Lebowitz

Columbia University, Department of Computer Science
New York, NY 10027

ABSTRACT

Automatic concept learning from large amounts of complex input data is an interesting and difficult process. In this paper we discuss how the use of a permanent, generalization-based, memory can serve as an important tool in developing programs that learn in rich input domains. Its use allows programs to determine what concepts to learn, as well as definitions of the concepts. We present in this paper a characterization of our research, and then describe our use of Generalization-Based Memory in two programs under development at Columbia, UNIMEM and RESEARCHER, and describe how they perform concept evaluation and generalization of complex structural descriptions, problems typical of those we are concerned with.

Key Terms: Learning, automatic concept formation, generalization, Generalization-Based Memory, intelligent information systems, artificial intelligence, cognitive science.

1 INTRODUCTION

Automatic concept learning in the form of generalization has been shown to be useful in interpreting and organizing large amounts of information about a domain [Lebowitz 80; Schank 82; Lebowitz 83a], as well as being an interesting task in its own right. Recently, we have been concerned with the development of new methods of concept formation that make use of permanent memory and include the determination of what concepts to learn, as well as the definitions of the concepts. In particular, we have concentrated on the problems of concept formation from a stream of input that is complex in any of several different ways. In this paper, we will detail the class of problems we are addressing, present the basic learning technique that we use, known as Generalization-Based Memory (GBM), and indicate solutions to some of the specific problems that are involved.

Much of the concept learning research that has been done in Artificial Intelligence has consisted of either supplying programs with examples, and possibly counter-examples, of specified concepts and having these

programs determine definitions of those concepts ([Winston 72; Mitchell 78], for example) or of using largely analytic techniques to classify input (e.g., [Michalski 80; Langley 81]). In "real-world" settings, the crucial concepts to be learned -- those that best help explain and organize information about a domain -- are not pre-supplied; rather, it is necessary to determine these concepts from a stream of very complex input data. Consequently, our research concentrates not just on how to compare examples, but also on methods for determining what examples to compare, which largely determines what concepts to create.

To take examples from various programs we have worked on, we will look here at how intelligent systems could extract from complex input streams generalizations such as: "States that have high school expenditures have high per capita incomes" (from information about the states of the United States); "A large class of disk drives use flexible (floppy) discs" (from patent abstracts about disk drives); or "Terrorist attacks in Northern Ireland are frequently carried out by the IRA" (from news stories about terrorism), to the same extent that people do.

We describe here a powerful memory organization and concept learning technique, Generalization-Based Memory. GBM was developed for the computer program IPP [Lebowitz 80; Lebowitz 83a], based on intuitions about how complex human episodes might be stored in memory in a manner analogous to Schank's MOPs [Schank 82; Kolodner 80]. We will discuss the advantages gained by using the same technique in a more traditional concept learning environment or for intelligence information systems that involve complex streams of input. Our presentation of the problems of concept learning from complex input is to be done in the context of two intelligent information systems being developed at Columbia, UNIMEM and RESEARCHER, both of which use GBM.

UNIMEM is a program that can accept a large quantity of relatively unstructured facts about a domain, use generalization techniques to determine important concepts, and use these concepts to organize the information in a fashion that allows further generalization and intelligent question answering. For example, if information about the states in the U.S. was given to such a program (a domain used in prototype testing), the program might determine that New England states, or states with large education budgets were useful concepts. UNIMEM is being used to study problems that can arise when the individual items used for learning are not highly structured, each consisting simply of a set of descriptive features.

¹This research was supported in part by the Defense Advanced Research Projects Agency under contract N00039-82-C-0427. Comments by Kathy McKeown on an earlier draft of this paper were most helpful.

The problems in forming concepts from complex input data involved in our research with UNIMEM include: the impact of domain-dependent knowledge on concept learning; categorizing numeric input information so that generalization is possible; concept evaluation and refinement from further examples; using concepts that very slightly contradict new input items; dealing with concepts that change over time; and question answering based on Generalization-Based Memory. In this paper we present the basic techniques for using GBM and for evaluating concepts in the context of UNIMEM.

RESEARCHER [Lebowitz 83b], in contrast with UNIMEM, deals with highly structured, physical descriptions of devices. RESEARCHER reads patent abstracts in natural language form, and then remembers and generalizes information from these texts. Understanding patents tends to require complex physical descriptions of objects. In this paper, we use RESEARCHER as a context in which to discuss the problems of comparing complex, highly structured representations.

Figure 1 shows some typical concepts generalized by each of the Generalization-Based Memory programs mentioned here. The IPP and UNIMEM generalizations were actually made by the programs (although the English was generated by hand), and the RESEARCHER examples show what we are aiming for.

IPP Concepts:

Bombings in El Salvador cause damage, but do not often hurt anyone.
Urban terrorists in Italy frequently use silencer equipped pistols.

UNIMEM Concepts:

State class -- High urban percentage, low minority percentage, moderate income, low taxes, manufacturing important [RI, NJ, TX, MI, FLA, OH]

State class -- High value of farmland, fairly high population, manufacturing, agriculture, tourism important [NC, ARK, TENN, MINN, WISC, VA, MO]

RESEARCHER Concepts:

Floppy disk drive

Double density disk drive

Fully enclosed disk drive

Figure 1: GBM Concept Examples

In the remainder of this paper, we describe how our research relates to other work in concept formation, and present an overview of our concept learning methods, concentrating on our use of Generalization-Based Memory. Finally, we describe the way we handle concept evaluation and generalization of complex structural descriptions, problems typical of those we are concerned with.

2 COMPLEX INPUT DOMAINS

The intelligent information systems we are developing basically engage in what [Michalski 83] calls *multiple concept learning from observation (descriptive generalization)*, in that these programs are given large number of examples, with no pre-specification of the concepts to generalize, and they acquire sets of concepts by deciding what instances to compare and how such examples are similar. The concepts derived are often overlapping, in that many concepts can describe the same example.

The tasks of our programs also involve aspects of Michalski's *concept acquisition*. In addition to determining the properties of instances in the classes that they create they fit objects to those classes. There are elements in our programs of both observing patterns in data and developing discriminant descriptions of the classes thereby derived.

Our research is characterized by several other properties, all somewhat novel for working systems, but, we feel, crucial to the development of useful, dynamic, information systems. We look at learning that is *incremental*. It is not possible in systems that are continually receiving input to wait for all examples to be available for inspection before creating concepts. We require that, after every example is processed, our systems have made the best possible generalizations based on the input that has been processed so far.

We expect that our systems will ultimately deal with *large numbers of examples*. It is the ability to deal with many examples and many concepts simultaneously that gives human learning the power we would like our systems to have. No method that requires comparison of a new instance with all, or a large portion of, previous examples will be acceptable, for computational reasons. Even comparison with generalized concepts must be done in a principled way. We are, in addition, sometimes concerned with cases where the individual items to be generalized are themselves complex, as in RESEARCHER.

The final parameter that characterizes all our work is that we are dealing with *"pragmatic" generalizations*. That is, we are concerned with concepts that describe what is usually, but not necessarily always, true. This means, crucially, that methods that invalidate a generalized concepts on the basis of a single example are not acceptable. In the same vein, we do not require that every concept that could legitimately be generalized be found. The class of pragmatic generalizations provides more power and flexibility in representing what it is possible to learn about a rich domain.

We feel that methods for dealing with the type of input described here will be necessary in developing systems that take full advantage of the large quantities of complex information we are now able to provide them.

3 GBM

In this section, we provide an overview of the techniques used to form concepts as part of maintaining a

Generalization-Based Memory. For clarity, we describe the way the process works in UNIMEM, but the main techniques are identical in IPP and RESEARCHER.

The basic idea of Generalization-Based Memory is that a generalization begins to create a hierarchy of concepts that describe a situation from a small number of examples, and then records in memory specific items, both those that the concepts are generalized from and further examples, in terms of the generalized concepts. More specific generalizations are recorded along with specific examples under the more general cases. GBM involves identifying and defining multiple concepts, as opposed to maintaining a single model of a concept.

In order to standardize our terminology, we refer to the objects stored in memory, i.e. the input examples, which are used to build generalizations, as *instances*. In UNIMEM these are descriptions of objects in a domain. An instance is described in UNIMEM terms of a set of *features* (essentially property/value pairs). As we will see, RESEARCHER uses more complex descriptions of instances. The combinations of generalizations, themselves sets of features, and the events and sub-generalizations they organize are called *GEN-NODEs*.² GEN-NODEs form the basis of GBM. The structure of a typical GEN-NODE is shown in Figure 2.

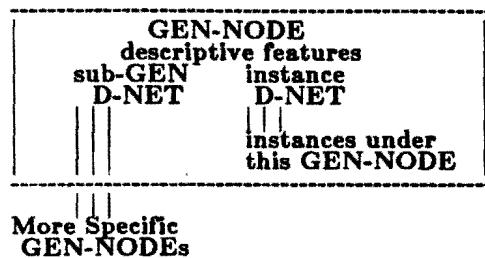


Figure 2: Schematic GEN-NODE

Generalization-Based Memory consists basically of one or more hierarchies of GEN-NODEs that describe concepts of increasing specificity. As shown in Figure 2, instances and sub-GEN-NODEs are stored under each GEN-NODE using discrimination networks (D-NETs) [Charniak, et al. 80]. (Note that a GEN-NODE can organize both instances and more specific GEN-NODEs.) D-NETs provide an efficient way to retrieve any object stored with a given set of indices. In the GBM model, every feature of an instance or sub-GEN-NODE is initially used as an index, resulting in shallow, bushy D-NETs that allow retrieval of an object given any one of its features. The resulting plethora of indices is pruned by ceasing to use as indices features that pertain to a large number of objects in a given D-NET.

The use of a hierarchy of GEN-NODEs with D-NETs as a method of memory organization allows efficient

storage of information, since information in a generalization does not have to be repeated for each instance that it describes. In addition, it allows relevant generalizations and instances -- and *only* relevant generalizations and instances -- to be efficiently found in memory during processing, allowing further generalizations. This property of GBM is largely independent of the specific knowledge representation being used.

The process of maintaining GBM, which is the learning process we are considering here, is a relatively simple one, once the memory organization method has been defined. As each new instance is processed, the most specific GEN-NODE that describes it is found. This is done easily and efficiently, using the discrimination nets that index the GEN-NODEs, starting with a very general node that covers the whole range of instances in the domain. Then, before the instance is actually indexed under that GEN-NODE, a check is made for instances already stored there that have additional features in common with the new instance, which can be found using the instance D-NET. If there are enough such features (one of many adjustable parameters of the system), a new concept is generalized, and the contributing instances indexed there. Otherwise, the new instance³ is simply stored under the existing GEN-NODE.

Two further important features characterize GBM. Since concepts are generalized on the basis of few instances, they must be evaluated to eliminate over-generalization (including the elimination of whole concepts). This is discussed in Section 4. The second feature is the use of an idea known as *predictability*. While space does not permit a discussion of predictability here (see [Lebowitz 83a]), the basic idea is that only the presence of some features of a concept in an instance indicate the relevance of the concept, and that these features can be identified quite easily using GBM.

To summarize, GBM satisfies handles the domain characteristics described in Section 2 as follows:

- 1) GBM is inherently *incremental*. As each instance is added to GBM, the best possible concepts that can be generalized so far are made.
- 2) GBM is ideal for handling *large numbers of examples*. The use of a hierarchy of concepts that organize specific instances allows only instances that might lead to generalizations to be compared to each other. Relevant concepts are easily found. It is also an efficient way to store the concepts.
- 3) All concepts generalized in GBM are "*pragmatic*". No concept is removed by a single counter-example, but

²GEN-NODEs were called S-MOPs in IPP, as they are, in some sense, specialized versions of Schank's Memory Organization Packets [Schank 82].

³The process is actually a bit more complex, as a given instance can be stored in multiple spots in memory for two different reasons. An instance can either be classified initially in several different ways, each of which would indicate a place to store it, or several different "most specific" GEN-NODEs might be found, each of which would lead to the processing described in the text.

instead, the process described in the next section is used to evaluate all concepts. The generalization process is also pragmatic because it can sometimes miss concepts that could be found by comparing instances that were stored in widely different parts of GBM, but this seems a reasonable trade-off to avoid combinatoric numbers of comparisons.

4 CONCEPT EVALUATION

As mentioned in the previous section, the concept learning process we have described inherently leads to over-generalization, particularly in a domain where there is a large amount of information about each instance. Thus, we require each concept learned to be evaluated over time. For each generalization made by UNIMEM, an evaluation process continually looks for later instances for which the generalization might be relevant. This occurs as a normal part of the memory update process, since the generalizations to be evaluated are exactly those that might be used to store the new instances. UNIMEM checks whether a relevant generalization is confirmed or contradicted by each new instance.

A new instance found by UNIMEM is considered to contradict an applicable concept if it possesses a predictable feature indicating that the concept is relevant, but also another feature with the same property as the concept (such as the region of a state), but with a different value (Midwest instead of East, perhaps). When this condition occurs, confidence in the concept is undermined.

Early versions of confidence for generalizations in GBM simply involved adding or subtracting points from a numeric confidence level for each GEN-NODE, resulting in a property much like the confidence in conclusions discussed in [Collins 78], or the confidence in rule application used in some expert systems. In a domain rich in information this technique will not do, as there will almost *always* be extraneous information in each generalized concept, as the result of inevitable coincidences, that will cause confidence in the concept to be undermined.

What we would like to do when a generalization is disconfirmed is to throw away the "bad" (overly specific) parts and keep the "good" parts. The problem then reduces to identifying the components of a generalization that are overly specific, so that they can be deleted, leaving intact a valid generalization. Furthermore, for this to be useful, it must be done at a minimum of cost, hopefully occurring as a natural part of the memory update process, and requiring only a small amount of extra record-keeping.

The solution devised for UNIMEM is straightforward. Instead of keeping a single confidence level as part of each GEN-NODE, UNIMEM tracks how often each feature of a concept is confirmed or contradicted. In effect, a confidence level is maintained for each feature of each concept, rather than a single value for an entire concept.

Specifically a counter is maintained for each feature of each generalization and these counters are incremented

or decremented as their features are confirmed or contradicted, respectively, in a situation where a concept is deemed relevant. The counter modification occurs as UNIMEM determines which GEN-NODEs best describe a new instance, as described in Section 3. If a counter passes a negative threshold, then we can eliminate the feature from the generalization, since the feature has been wrong much more often than right. We sometimes have to eliminate entire generalizations when too many of their features have been eliminated. Details of this process, and an example of its application in the domain of football plays, can be found in [Lebowitz 82].

When this scheme was added to UNIMEM, it proved quite effective in culling extraneous features from generalizations, and only totally disconfirming those concepts that were completely the result of coincidence. We show here a simple example from the states domain, in which we provided UNIMEM with a number of facts about each state in the United States.

Figure 3 illustrates a concept (GND1) generalized by UNIMEM. In it, the third column of each feature (the value) indicates a category derived from numeric data by a method described in [Lebowitz 83c]. For example, the first feature of the generalization is that per capita income in the relevant states is in the third of four categories, i.e., rather high. Roughly, the concept generalized here describes states with moderately high per capita income, rather low taxes, high school expenditure, and fairly low minority population (the last is actually a broad category that covers most states). This concept can be used to describe the seven states listed.

```
GND1:
INCOME      RANGE      INC3:4
TAXES       RANGE      TAX2:5
SCHOOL-EXP  RANGE      SCH3:3
MINORITY-PCT RANGE      MIN1:2
Organizing: IOWA, KANSAS, MICHIGAN, MONTANA,
             NEBRASKA, PENNSYLVANIA, TEXAS
```

Figure 3: Final UNIMEM Generalization

Figure 4 shows how this concept was initially generalized from Iowa and Nebraska (the states were presented to UNIMEM in a random order). Notice that these states are similar in a number of additional ways, e.g. they are both farm states, so UNIMEM initially generalized an over-general, and not widely applicable, concept. These extraneous features were ultimately removed by the evaluation process described in this section, leaving a much more useful concept.

```
GND1:
CRIME-RATE  RANGE      CR13:5
STATE-DEBT  RANGE      DEB2:7
INCOME      RANGE      INC3:4
TAXES       RANGE      TAX2:5
MIGRATION-NET RANGE      MIG1:9
SCHOOL-EXP  RANGE      SCH3:3
STATE       REGION     FARM
MINORITY    RANGE      MIN1:2
Organizing - IOWA NEBRASKA
```

Figure 4: Initial Generalization

Finally, in Figure 5 we show how UNIMEM was able to

generalize more specific concepts when an attempt was made to store additional instances under the GEN-NODE GND1. (Note that this means that the seven states shown in Figure 3 were not all stored under GND1 at any single time.) For example, Kansas and Nebraska are farm states with low migration rates. This typifies the process in GBM that rarely allows large numbers of instances to end up stored under a given GEN-NODE -- further generalization inevitably creates new concepts which are added to the concept hierarchy in GBM and used to help organize instances.

GND15:
STATE SIZE SIZE:6
MIGRATION-NEY RANGE MIG:9
Organizing: IOWA, MICHIGAN, PENNSYLVANIA

GND0:
STATE REGION FARM
MIGRATION-NEY RANGE MIG1:9
Organizing: KANSAS, NEBRASKA

GND25:
STATE SIZE SIZE:6
INDUSTRY TYPE MANUFACTURING
Organizing: MONTANA, TEXAS

Figure 5: Sub-generalizations of GND1

5 RESEARCHER

As mentioned earlier in this paper, RESEARCHER is a program that reads patent abstracts and adds information from them to a Generalization-Based Memory so that it can effectively answer questions [Lebowitz 83b]. In this paper, we look only at the process of taking representations of two objects (or, equivalently, a generalized concept and a concrete object) and extracting a generalized concept. The representations we compare are frame-like and primitive-based, concentrating on the physical relations among the various parts of a complex object. (See [Wasserman and Lebowitz 82] for a complete description of the representation scheme.)

In the disc drive domain, typical concepts the generalization process might identify as being useful would be floppy disc drives or double sided discs. As with all our work, this must be done without specifically providing with examples of these concepts. Instead, instances stored together in Generalization-Based Memory, are recognized as being similar and generalized.

The use of GBM is more complex here than in the UNIMEM. The "features" that two objects have in common can only be determined by comparing two complex object representations. The matching problem is much the same as that faced by Winston in his blocks world learning work [Winston 72]. Our problem is in certain ways both harder and easier than Winston's. It is harder because we are dealing with much more complex representations. It is simplified, however, at least in the long run, by the existence of an entire GBM, rather than a single model. We believe this will simplify the matching process. We look here both at the complexity of matching object descriptions and at how GBM can simplify the process.

The representations for two similar, slightly simplified disc drive patents, used to test the initial version of RESEARCHER's generalization module are shown in Figure 6.

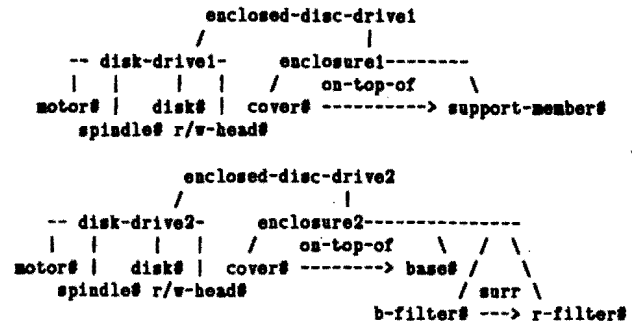


Figure 6: Similar Disc Drives

Clearly the two disc drives in Figure 6 have much in common that can be the source of a new concept derived through generalization -- an "enclosed disc drive". Figure 7 shows the concept created by RESEARCHER's generalization module.

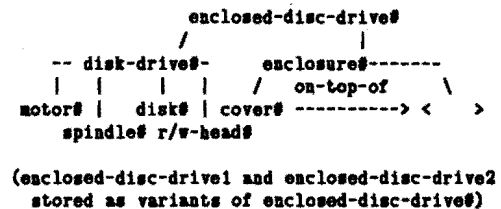


Figure 7: Generalized Enclosed Disc Drive

The idea illustrated in Figure 7 is that RESEARCHER finds the parts of two objects that are similar, and abstracts them out into a generalized concept. In this example, the two devices contained similar disc drives and enclosures. Each had a cover on top of some other object. These similarities form the basis of a generalized enclosed disc drive. Only the additional parts and relations of each instance need be recorded in memory along with the generalization.

Adapting GBM for use on complex structural descriptions has proven to be a difficult problem, even when only considering the assorted relations among the objects in the descriptions. Here we present one of the major problems and suggest the nature of the possible solution.

The central problem in generalizing structural descriptions is the process of matching two representations (either of two objects or an object and a generalized object), thereby determining what parts and relations correspond (as was pointed out for simpler examples in [Winston 72]). Clearly, if we wish to

determine that the disk mounts in two drives are similar, they must be compared with each other. Since one part of the description of complex objects is a set of relations, we must associate the relations in one object with those in the other.

The matching process here is a quite difficult one. Since we are dealing with structured objects, the parts of very similar objects may be aggregated differently in various descriptions. For example, a read/write head might be described as a direct part of a disc drive in one patent, but part of a "read/write assembly" in another. This makes the inherent similarity hard to identify.

At the moment, we deal with this "level problem" with simple heuristics that allow only a limited amount of "level hopping" during the comparison process (to avoid the need to consider every possible correspondence among levels), and a bit of combinatoric force.

We feel that the ultimate solution to the level problem lies in more extensive use of Generalization-Based Memory. If a new object can be identified as an instance of a generalized concept, with only a few minor differences (done with a discrimination-net-based search of the sort described in Section 3), then the levels of aggregation will be set. In effect, by using the form of aggregation that was generalized, we will have a canonical framework for describing new objects. In addition, by using Generalization-Based Memory, we need compare only a small number of *differences* between objects, rather than complex descriptions. This should allow RESEARCHER to meet all our performance constraints (i.e. be incremental, handle large numbers of objects, and generalize pragmatically), even using the complex representations needed to describe real world objects.

6 CONCLUSION

We believe that our work with Generalization-Based Memory has several important morals. The first is that the development of a dynamic set of concepts is a powerful approach to take when learning from a rich input domain. We can never hope to find the "right" set of concepts, so it is crucial that we constantly update the concepts that we have and look for new ones. Furthermore, the use of long-term memory, in the form of GBM, allows us to deal with many concepts at once, and still retain efficiency. In fact, as we have shown considering many concepts at once often ends up being easier than one at a time and certainly leads to more powerful systems.

REFERENCES

- [Charniak, et al. 80] Charniak E., Riesbeck, C. K., and McDermott, D. V. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, Hillsdale, New Jersey, 1980.
- [Collins 78] Collins, A. Fragments of a theory of human plausible reasoning. TINLAP-2, Urbana-Champaign, Illinois, 1978.
- [Kolodner 80] Kolodner, J. L. Retrieval and organizational strategies in conceptual memory: A computer model. Technical Report 187, Yale University Department of Computer Science, 1980.
- [Langley 81] Langley, P. "Data-driven discovery of natural laws." *Cognitive Science* 5, 1 (1981), 31 - 54.
- [Lebowitz 80] Lebowitz, M. Generalization and memory in an integrated understanding system. Technical Report 186, Yale University Department of Computer Science, 1980. PhD Thesis
- [Lebowitz 82] Lebowitz, M. "Correcting erroneous generalizations." *Cognition and Brain Theory* 5, 4 (1982), 367 - 381.
- [Lebowitz 83a] Lebowitz, M. "Generalization from natural language text." *Cognitive Science* 7, 1 (1983), 1 - 40.
- [Lebowitz 83b] Lebowitz, M. Intelligent information systems. Proceedings of the Sixth International ACM SIGIR Conference, ACM SIGIR, Washington, DC, 1983.
- [Lebowitz 83c] Lebowitz, M. Classifying Numeric Information. Columbia University Department of Computer Science, 1983.
- [Michalski 80] Michalski, R. S. "Pattern recognition as rule-guided inductive inference." *IEEE Transactions on Pattern Analysis and Machine Intelligence* (1980).
- [Michalski 83] Michalski, R. S. "A theory of methodology of inductive learning." *Artificial Intelligence* 20 (1983), 111 - 161.
- [Mitchell 78] Mitchell, T. M. Version spaces: An approach to concept learning. Technical Report HPP-79-2, Stanford University, Stanford, California, 1978.
- [Schank 82] Schank, R. C. *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*. Cambridge University Press, New York, 1982.
- [Wasserman and Lebowitz 82] Wasserman, K. and Lebowitz, M. Representing complex physical objects in memory. Columbia University Department of Computer Science, 1982.
- [Winston 72] Winston, P. H. Learning structural descriptions from examples. In P. H. Winston, Ed., *The Psychology of Computer Vision*, McGraw-Hill, New York, 1972.

The Chunking of Goal Hierarchies: A Generalized Model of Practice

Paul S. Rosenbloom and Allen Newell
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

This work describes recent advances in the specification and implementation of a model of practice. In previous work we showed that there is a ubiquitous regularity underlying human practice — the *power law of practice*. We also developed an abstract model of practice — the *chunking theory of learning*. This previous work established the feasibility of the chunking theory for a single task — a 1023-choice reaction-time task — but the implementation was specific to that one task. In the current work we develop a modified formulation of the chunking theory that allows a more general implementation. In this formulation, task algorithms are expressed in terms of hierarchical goal structures. These algorithms are simulated within a goal-based production-system architecture designed for this purpose. *Chunking* occurs during task performance in terms of the parameters and results of the goals experienced. It improves the performance of the system by gradually reducing the need to decompose goals into their subgoals. This model has been successfully applied to the task employed in the previous work, and to a set of new tasks.¹

Key Terms: Practice, learning, machine learning, artificial intelligence, cognitive science, chunking, goal processing, production systems.

1. Introduction

How can systems — both natural and artificial — improve their own performance? At least for natural systems (people, for example), we know that *practice* is effective. A system is engaged in practice when it repeatedly performs one, or a set of similar, tasks. Recently, Newell and Rosenbloom [1981] brought together the evidence that there is a ubiquitous law — the *power law of practice* — that characterizes the improvements in human performance during practice. The law states that when human performance is measured in terms of the time to perform a task, it improves as a power-law function of the number of times the task has been performed (called the *trial number*). This result holds over the entire domain of human performance, including both purely perceptual tasks, such as target detection [Neisser, Novick, & Lazar, 1963], and purely cognitive tasks, such as

supplying justifications for geometric proofs [Neves & Anderson, 1981] or playing a game of solitaire [Newell & Rosenbloom, 1981].

The ubiquity of the power law of practice suggests that it may reflect something in the underlying *cognitive architecture*. The nature of the architecture is of fundamental importance for both Artificial Intelligence and Psychology (in fact, for all of Cognitive Science). It provides the control structure within which thought occurs; determining which computations are easy and inexpensive, and what errors will be made and when. Take the recent flux of work on expert systems such as MYCIN [Shortliffe, 1976] and R1 [McDermott, 1980] for example. Two important ingredients in such systems stem from fundamental work on the cognitive architecture; specifically, the development of *production systems* [Newell & Simon, 1972; Newell, 1973] and *goal-structured problem solving* [Ernst & Newell, 1969; Newell & Simon, 1972]. This article discusses recent efforts to take advantage of the power law of practice by using it as a generator for a production-system practice mechanism. This is a highly constrained task because of the paucity of plausible practice models that can produce power-law practice curves [Newell & Rosenbloom, 1981].

As a beginning, Newell and Rosenbloom [1981] developed an abstract model of practice based on the concept of *chunking* — a concept already established to be ubiquitous in human performance — and derived from it a practice equation capable of closely mimicking a power law. We hypothesized that this model formed the basis for the performance improvements brought about by practice. In Sections 2 and 3 we briefly summarize this work on the power law and the abstract formulation of the chunking theory.

Rosenbloom and Newell [1982; In press] took the analysis one step further by showing how the *chunking theory of learning* could be implemented within a highly parallel, activation-based production system — *Xaps2* — for a 1023-choice reaction-time task [Seibel, 1963]. This work established more securely that the theory is a viable model of human practice, by showing how it could actually be applied to a task to produce power-law practice curves. By producing a working system, it also established the theory's viability as a practice mechanism for artificial systems.

The principal weakness of the work done up to that point, was that the implementation was heavily task dependent. Both the representation used for describing the task, and the chunking mechanism itself had built into them knowledge about the specific task and how it should be performed. To produce a generalized model of practice, this flaw must be removed by developing generalized, task-independent models of performance and practice.

The first component of the mission — generalizing the model

¹ This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

We would like to thank John Laird for innumerable helpful discussions about this material.

of task performance — is discussed in detail in Rosenbloom [1983]. It is tackled by examining a set of tasks that fall within a neighborhood around the Seibel [1963] task — *stimulus response compatibility* tasks. The area of compatibility is a subdomain of the set of tasks referred to as *reaction-time tasks*. These tasks, which are used widely in psychological experimentation, involve the presentation to a subject of a stimulus display — such as an array of lights, a string of characters on a computer terminal, or a spoken word — for which a specific "correct" response is expected. The response may be spoken, manual — such as pressing a button, pushing a lever, or typing — or something quite different. From the subject's reaction-time — the time it takes to make the response — and error rate, it is possible to draw conclusions about the nature of his cognitive processing.

We will not pursue the analysis of compatibility in detail here, except to note that models based on the concept of *goal hierarchies* provide excellent fits to the data [Rosenbloom, 1983]. In a goal hierarchy, the root node expresses a desire to do a task. At each level further down in the hierarchy, the goals at the level above are decomposed into a set of smaller goals to be achieved. Decomposition continues until the goals at some level can be achieved directly. This is a common control structure for the kinds of complex problem-solving systems found in artificial intelligence, but this is the first time they have been applied to the domain of reaction time tasks.

These goal structures (discussed in Section 4) provide us with the generalized task representation required as the basis for a task-independent chunking mechanism. Chunking essentially implements a form of store-versus-compute trade-off, in which patterns of parameter values are related to patterns of results. This practice mechanism is described in Section 5.

The chunking and goal processing mechanisms were implemented as part of the architecture of the *Xaps3* production system. The *Xaps3* architecture is an evolutionary development from the *Xaps2* architecture. Only those changes necessitated by the needs of chunking and goal processing were made. This architecture is described in Section 6.

In Section 7 we present some of the new results generated from simulations of the practice and compatibility tasks within the new architecture².

In Section 8 we place this work in perspective by relating the chunking theory to previous work on learning mechanisms. The theory stakes out a position intermediary between three previously disparate mechanisms, bringing out an unexpected commonality among them.

In Section 9 we propose a way in which the scope of the chunking theory can be expanded to more than just speeding up existing algorithms. We describe a way in which chunking could be lead to perform such other varieties of learning as *method acquisition* — the way in which a system first develops a performance algorithm for a task.

Finally, in Section 10, we conclude and summarize.

2. The Power Law of Practice

Performance improves with practice. More precisely, the time to perform a task decreases as a power-law function of the number of times the task has been performed. This basic law —

known as the *power law of practice* or the *log-log linear learning law*³ — has been known since Snoddy [1926]. While this law was originally recognized in the domain of motor skills, it has recently become clear that it holds over a much wider range of human tasks — possibly extending to the full range of human performance. Newell and Rosenbloom [1981] showed this for perceptual-motor skills [Snoddy, 1926; Crossman, 1959], perception [Kolars, 1975; Neisser, Novick, & Lazar, 1963], motor behavior [Card, English & Burr, 1978], elementary decisions [Seibel, 1963], memory [Anderson, 1980], routine cognitive skill [Moran, 1980], and problem solving [Neves & Anderson, 1981; Newell & Rosenbloom, 1981].

Practice curves are generated by plotting task performance against trial number. This cannot be done without assuming some specific *measure* of performance. There are many possibilities for such a measure, including such things as quantity produced per unit time and number of errors per trial. The power law of practice is defined in terms of the *time* to perform the task on a trial. It states that the time to perform the task (T) is a power-law function of the trial number (N):

$$T = BN^{-\alpha} \quad (2.1)$$

If Equation (2.1) is transformed by taking the logarithm of both sides, it becomes clear why power-law functions plot as straight lines on log-log paper:

$$\log(T) = \log(B) + (-\alpha) \log(N) \quad (2.2)$$

Figure 2-1 shows the practice curve from one subject in Kolars [1975] study of reading inverted texts — each line of text on the page was turned upside down — as plotted on log-log paper. The solid line represents the power-law fit to this data. Its linearity is clear ($r^2 = 0.932$).

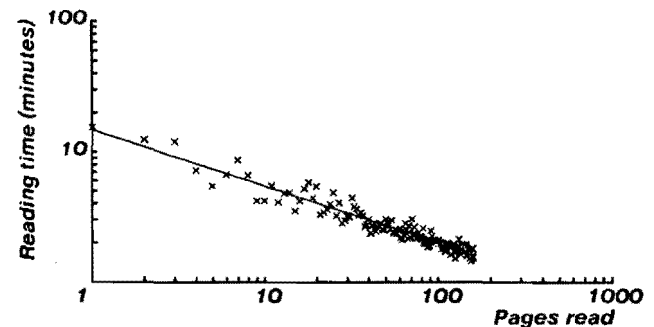


Figure 2-1: Learning to read inverted text (log-log coordinates). Plotted from the original data for Subject HA [Kolars, 1975].

Many practice curves are linear (in log-log coordinates) over much of their range, but show a flattening at their two ends. These deviations can be removed by using a four-parameter *generalized* power-law function. One of the two new parameters (A) takes into account that the asymptote of learning can be greater than zero. In general, there is a non-zero minimum bound on performance time, determined by basic physiological

² A more comprehensive presentation and discussion of these results can be found in Rosenbloom [1983].

³ Power-law curves plot as straight lines on log-log paper.

limitations and/or device limitations — if, for example, the subject must operate a machine. The other added parameter (E) is required because power laws are not translation invariant. Practice occurring before the official beginning of the experiment — even if it consists only of transfer of training from everyday experience — will alter the shape of the curve, unless the effect is explicitly allowed for by the inclusion of this parameter. Augmenting the power-law function by these two parameters yields the following generalized function:

$$T = A + B(N + E)^{-\alpha} \quad (2.3)$$

A generalized power law plots as a straight line on log-log paper once the effects of the asymptote (A) are removed from the time (T), and the effective number of trials prior to the experiment (E) are added to those performed during the experiment (N):

$$\log(T - A) = \log(B) + (-\alpha) \log(N + E) \quad (2.4)$$

Figure 2-2 shows a practice curve from a 1023-choice reaction-time task [Seibel, 1963], as it is fit by a generalized power-law function (each data point represents the mean reaction time over a block of 1023 trials). This is a perceptual-motor task in which the task environment consists of a horizontal stimulus array of ten lights, and a response array of ten buttons in a highly compatible one-one correspondence with the lights, each button right below a light. On each trial, some of the lights are *On*, and some are *Off*. The subject's task is to respond by pressing the buttons corresponding to the lights that are *On*.

This curve, which shows flattening at both ends when plotted as a simple power law, is now linear over the whole range of trials. As stated earlier, similar fits are found across all

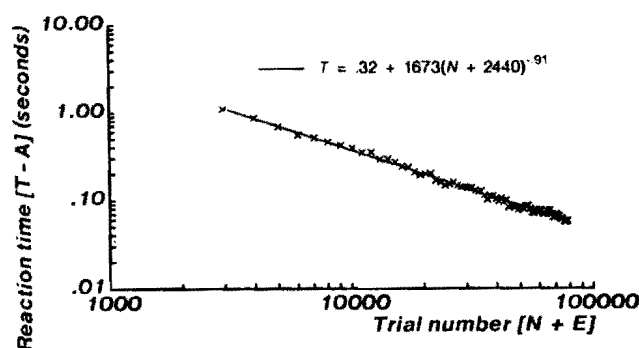


Figure 2-2: Optimal general power law fit to the Seibel data (Log-Log coordinates).

dimensions of human performance. Though these fits are impressive, it must be stressed that the power law of practice is only an *empirical* law. The true underlying law must resemble a power law, but it may have a different analytical form.

3. The Chunking Theory of Learning

The chunking theory of learning proposes that practice improves performance via the acquisition of knowledge about patterns in the task environment. Implicit in this theory is a model of task performance based on this pattern knowledge. These patterns are called *chunks* [Miller, 1956]. The theory thus starts from the *chunking hypothesis*:

- *The Chunking Hypothesis*: A human acquires and organizes knowledge of the environment by forming and storing expressions, called *chunks*, which are structured collections of the chunks existing at the time of learning.

The existence of chunks implies that memory is hierarchically structured as a lattice (tangled hierarchy, acyclic directed graph, etc.), rooted in a set of preexisting *primitives*. A given chunk can be accessed in a top-down fashion, by *decoding* a chunk of which it is a part, or in a bottom-up fashion, by *encoding* from the parts of the chunk. Encoding is a recognition or parsing process.

The existence of chunks does not need to be justified solely on the basis of the practice curves. Chunks stand on their own as a thoroughly documented component of human performance [Miller, 1956; DeGroot, 1965; Bower & Winzenz, 1969; Johnson, 1972; Chase & Simon, 1973; Chase & Ericsson, 1981]. The traditional view of chunks is that they are data structures representing the combination of several items. For example, in one set of classic experiments, Bower and colleagues [Bower, 1972; Bower & Springston, 1970; Bower & Winzenz, 1969] showed that recall of strings of numbers or letters is strongly influenced by the segmentation of the string. If the segmentation corresponds to a previously learned grouping of the items (for example, FBI-PHD-TWA-IBM), performance is better than if it is contradictory (FB-IPH-DTW-AIB-M). These results were interpreted as evidence for segmentation guided chunking of familiar strings. By replacing a string of several letters with a single chunk, the subject's memory load is reduced, allowing more letters to be remembered. At recall time the chunks are decoded to yield the original items to be recalled.

The chunking theory of learning proposes two modifications to this classical view. The first change is to assume that there is not a single symbol (chunk) to which the items are encoded, and from which they can later be decoded. As a simple example, the process of reading the string IBM out loud is more than just the encoding of the three letters into a chunk, followed by the subsequent decoding to the three letters. What needs to be decoded is not the visual representation of IBM, but the articulatory representation — allowing the subject to say IBM.

Based on this consideration, the chunking theory assumes that there are two symbols for each chunk — a *stimulus* symbol and a *response* symbol. The process of using a chunk consists of encoding the stimulus items to the stimulus symbol (a many-one mapping), mapping the stimulus symbol to the response symbol (a one-one mapping), and decoding the response symbol to the response items (a one-many mapping) (Figure 3-1). Encoding and decoding are fast hierarchical processes, while the mapping serves as a (serial) point of control at which the choice of response can be made. Acquisition of a chunk speeds up performance by reducing the number of mappings to be performed. In the example in Figure 3-1, before the chunk is acquired, three mappings (one for each letter) are required. After acquisition of the chunk, one mapping suffices.

The second difference between this proposal and the classical view of chunking is that the chunk consists of the three processes (encoding, mapping, and decoding), not just the symbols. This is really just a shift of emphasis, in which chunks are viewed as the processes, rather than just as the results of the processes.

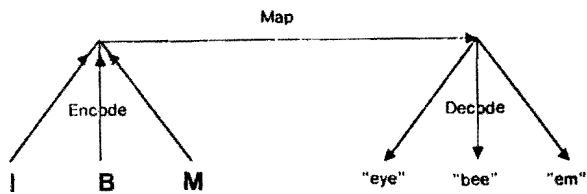


Figure 3-1: A three-part chunk for the articulation of the visual string "IBM".

4. Goal-Structured Performance Models

In its current formulation, chunking explains how performance on a task can be sped up with practice. It does not explain how the organism first learns to do the task (but see the discussion in Section 9). Consequently, each task simulation must be initialized with a performance model for the task. What is needed — and is provided by the notion of a *goal hierarchy* — is a general, task-independent representation for these performance models. Goal hierarchies are found frequently in artificial intelligence systems, but have not previously been employed in the modeling of the kinds of reaction-time tasks with which we are dealing.

Goal hierarchies are built out of *goals*, where each goal is a data structure representing a desired state of affairs. A goal is not a procedure for bringing about that state; it is only a description of the state. In order to bring about the goal state, there must be a *method* associated with the goal. The method could be a rigid algorithm, or it could be one of the more flexible *weak methods* [Newell, 1969] such as means-ends analysis [Ernst & Newell, 1969] or heuristic search [Nilsson, 1971]. Goals take *parameters* — information used in the fulfillment of the goal — and return *results*.

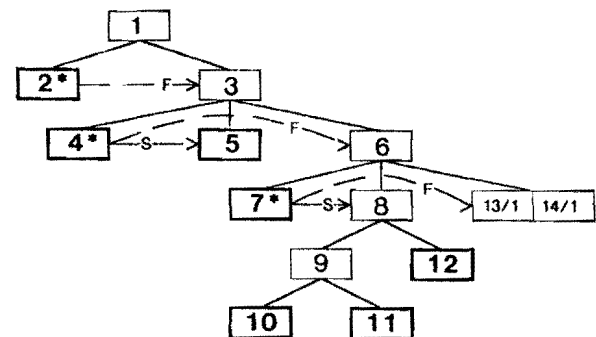
When a goal can be decomposed into a set of simpler goals [Nilsson, 1971], and those goals can be decomposed even further, a goal hierarchy results. In its simplest form, as an AND hierarchy, a goal is successful if all of its subgoals are successful. The structure to be described here, more closely resembles an AND/OR hierarchy, in which some goals succeed only if all of their subgoals succeed, and others succeed if any one of their subgoals succeed. A *terminal* goal is reached when the goal can be fulfilled directly, without the need for further decomposition.

One strategy for processing a goal hierarchy is to attempt the goals in a *depth-first* fashion, in which the most recently generated (the deepest) goal is always selected as the next goal to process. With a depth-first paradigm, there is always exactly one goal being actively worked on at any point in time. We will refer to this goal as the *active* or *current* goal. When a subgoal becomes the active goal, the parent goal of that subgoal is *suspended* until control is returned to it by completion of the subgoal, at which point it again becomes the current goal. On completion, the subgoal will either have *succeeded* or *failed*. The *control stack* specifies the location in the hierarchy at which control currently resides. It does this by maintaining the path from the root goal of the hierarchy to the current goal. This path consists of the active goal, and all of its suspended ancestors.

Figure 4-1 shows two different representations of a goal hierarchy for the Seibel [1963] task. At the top of the figure, the

hierarchy is shown as a tree structure. At the bottom of the figure, the goals are shown in their depth-first processing order. In both representations, the bold-faced goals are the terminals — those goals that can be fulfilled directly. The labeled arrows (either S or F) represent *branch* points in the hierarchy. With simple depth-first processing, the children of a node are processed in a strict left-to-right fashion. We have augmented this style of control by allowing the left-to-right processing to be conditioned on the success or failure of the previous child (actually of any previous goal for which this information is still available). For example, goal number 2 (No-Light-On?) tests whether there are any lights on within a specific region of the display of lights. Only if this goal fails, should goal 3 (Do-Lights) be attempted. If goal 2 succeeds, then there are no lights on that need to be processed, and the parent goal (goal 1: Do-Lights-If-Any) can just be terminated successfully. Those goals that do not return a result are called *predicates* (denoted by a * in Figure 4-1), and are used to test the truth of various conditions. It is only the success or failure of predicates that matters (as the basis for a branch). If the predicate succeeds, then the condition is true. If it fails, it could be either that the condition is false, or that something went wrong during the attempt.

The goal structure in Figure 4-1 is based on a recursive divide-and-conquer algorithm in which the stimulus display is broken up into smaller and smaller horizontal segments, until *manageable* pieces are generated. There are three types of horizontal segments that have been defined as manageable. The first type of manageable segment is one in which no lights are on. Such segments require no explicit processing, so the goal just returns with success. The opposite of the first type of segment — one in which no lights are off — is also manageable. For such a segment, the system generates a single response



1. Do-Lights-If-Any(Min-X, Max-X)
2. No-Light-On?(Min-X, Max-X)
- IF FAILED No-Light-On? THEN
3. Do-Lights(Min-X, Max-X)
4. No-Light-Off?(Min-X, Max-X)
- IF SUCCEEDED No-Light-Off? THEN
5. Do-Press-All-Buttons(Min-X, Max-X)
- IF FAILED No-Light-Off? THEN
6. Do-Off-And-On(Min-X, Max-X)
7. One-Light-On?(Min-X, Max-X)
- IF SUCCEEDED One-Light-On? THEN
8. Press-Button-Under-On-Light(Min-X, Max-X)
9. Get-On Light X(Min-X, Max-X)
10. Get-On-Light-Stimulus(Min-X, Max-X)
11. Get-Stimulus-X
12. Press-Button-At-X
- IF FAILED One-Light-On? THEN
13. Do-Lights-If-Any(Min-X, [Min-X + Max-X]/2)
14. Do-Lights-If-Any([Min-X + Max-X]/2, Max-X)

Figure 4-1: Goal hierarchy for the Seibel [1963] task.

specifying that a *press* action must occur in the entire region

defined by the segment (using the Do-Press-All-Buttons goal). Specifying a single button press is actually a special case of this, in which the region is just large enough to contain the one button. Allowing multi-on-light segments to be manageable, implies that sequences of adjacent on-lights can be pressed simultaneously, even before chunking has begun. Such behavior is seen very early in the trial sequence for some subjects. The remaining manageable segments are those that contain exactly one light on. These segments are processed (using the Press-Button-Under-On-Light goal) by finding the location of that light, and generating a button press at that location. If a generated segment does not meet any of these three criteria, it is unmanageable, and is split into two smaller segments.

The recursive aspect of the algorithm implies that many different instances of each goal will be simultaneously represented in the system, though at most one can actually be active. In order to keep track of which goal instance is relevant to which segment of the stimulus display, the segment (in terms of its minimum and maximum X values) is an *explicit* parameter to the goals. In addition to the explicit parameters, a goal can have *implicitly* defined parameters. Any object existing before the activation of the goal (i.e., part of the goal's *initial state*), that is examined during the processing of the goal — such as a stimulus light — is an implicit parameter of the goal. Implicit parameters are dynamically determined by the actual processing of the goal.

A second implication of the recursion in this hierarchy, is that the system does not start off with a constant built-in goal hierarchy. Instead, it has a generator of goal hierarchies. That chunking works on such a structure, is important for any claims about the potential generality of the mechanism (for more on this, see Section 9).

The recursion occurs at goals 13 and 14 in Figure 4-1. They are repetitions of the topmost goal in the hierarchy (Do-Lights-If-Any), but the scope of each is limited to one half of the display currently being processed. The numeric computation to obtain the middle of the segment (involving an addition and a division), while on the surface too powerful a computation to appear where it does, is only intended as an approximation to a process that divides the stimulus display into two (or three) roughly equal parts.

In addition to the goal hierarchy, the model assumes a *working memory* for the short-term storage of information relevant to the processing that is going on. For each goal, the working memory is logically partitioned into two components — the *initial state* and the *local state*. The initial state consists of the data existing at the time the goal is first activated. The remainder of the working memory — consisting of those pieces of data created during the processing of the goal — make up its local state. Only the local state of a goal can be modified during the processing of that goal; the initial state can be examined, but not modified. The modularity resulting from this scoping rule increases the likelihood that an arbitrary set of goals can be pursued without interfering with each other. This modularity is also important in ensuring correct performance of the chunking mechanism.

Each datum in the working memory is relevant in a particular temporal *processing context*. As long as the datum's creator goal is in the control stack, the system is either working on that goal, or on one of its descendents. The datum may be relevant at any point in this processing. However, once the goal disappears from the control stack, it is no longer being pursued.

The datum will most likely no longer be relevant. The processing context of a datum is that period of processing during which its creator goal is in the control stack. Once a piece of information becomes *out of context* — its creator is no longer part of the control stack — it usually can be safely deleted from the working memory. Data that cannot be deleted — because they are needed outside of the context in which they were created — are called the *results* of the goal. Their continued presence in the working memory is assured by changing their context to be the context of the parent goal.

5. Chunking on Goal Hierarchies

Given a task-independent organization for performance models, we can return to the original objective to describe a task-independent formulation of the chunking theory. In the chunking mechanism described by Rosenbloom and Newell [1982; In press], chunks related patterns of stimuli (lights) to patterns of responses (button presses). The goal-oriented formulation is obtained by altering this definition slightly so that chunks relate patterns of goal parameters to patterns of goal results. Each chunk improves the performance of the system by eliminating the need to fully process a specific instance (a combination of parameter values) of a particular goal. It replaces the normal processing (decomposition into subgoals for non-terminal goals, and direct execution of an action for terminal goals) with a direct connection between the relevant parameter values and results. A goal can (and almost certainly will) have more than one possible chunk; each combination of parameter values requires its own chunk. It should be clear that chunking implements a store-versus-compute trade-off, a form of goal-based caching.

As with the abstract characterization of the chunking theory, each chunk consists of three components: *encoding*, *decoding*, and *connection* (or *mapping*). The goal's parameter values form the basis for the encoding component. Given the presence of those values in the working memory, the encoding component generates a new object representing their combination. Encoding is a parallel, goal-independent, data-driven process. Every encoding component executes as soon as appropriate, irrespective of whatever else is happening in the system. The results of encoding components can themselves become parameter values to other goals, leading to a hierarchical encoding process.

The results of the goal form the basis for the decoding component. Given the presence of an encoded result-object in the working memory, the decoding component generates the actual results returned by the goal. Decoding occurs when the results are needed. As with encoding, the set of decoding components forms a hierarchical structure, in which complex results are decoded to simpler ones, which are then decoded even further.

The connection component of the chunk generates the encoded result from the encoded parameter. Connections provide a locus of control by occurring serially, under the control of the goals. Thus, a connection can be made only when the system is working on the goal for which the chunk was formed (and after the encoding component has executed). This assures that, even though encoding and decoding are uncontrolled, only appropriate results are generated.

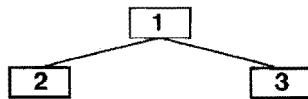
Chunks are created bottom up in the goal hierarchy. A chunk can be created for a goal when the following two conditions are met: (1) the goal has just completed successfully, and (2) all of

the goal's subgoals were themselves processed by chunks. It is this bottom-up aspect of chunking that leads to hierarchical encoding and decoding networks. However, notice that bottom-up chunking does not imply that all low-level chunks are learned before any high-level chunks are learned, or even that all of the chunks must be learned for a subgoal before any can be learned for its parent goal. The second condition on chunk creation merely states that chunks must exist for the goal's subgoals *in the current situation*. Whether other chunks exist or do not exist for the subgoals is irrelevant.

Given enough practice, all of the situations for all of the goals in the hierarchy will be chunked, and asymptotic behavior will be reached for the task. The amount of time this takes depends on the number of goals, the number of situations for each goal, how frequently the different situations arise, and whether chunks are created whenever they can be.

As a simple example of how chunking works, consider the simple three-goal hierarchy in Figure 5-1. This structure computes the average of two numbers. The top-level goal (Compute-Average-Of-Two-Numbers) takes as parameters the two numbers to be averaged, and returns a single result, which is their mean. The first subgoal (Compute-Sum-Of-Two-Numbers) performs the first half of the computation. It takes the two numbers as parameters, and returns their sum as its result. The second subgoal finishes the computation by taking the sum as a parameter, and returning half of it as its result.

Suppose that the first task is to average the numbers 3 and 7. Control would pass from goal number 1 to goal number 2. When goal 2 finishes and returns its result of 10, a chunk of three components is created (bottom left of Figure 5-2). An encoding



1. Compute-Average-Of-Two-Numbers
2. Compute-Sum-Of-Two-Numbers
3. Divide-Sum-By-2

Figure 5-1: A simple three-goal hierarchy for the averaging of two numbers.

component is created that encodes the two parameters (3 and 7), into a new symbol (E1). It executes as soon as it is created, because the parameters are in the working memory. A decoding component is created that, when necessary, decodes from a second new symbol (D1) to the result (10). A connection component (the horizontal line with the goal name above it and goal number below it) is created that generates the result symbol (D1) when it detects both the presence of the encoded parameter (E1) and that goal 2 is the active goal. The connection does not execute immediately, because goal 2 is already complete when the chunk is created.

Following the termination of goal 2, goal 1 is reactivated, but then is suspended in favor of goal 3 (Divide-Sum-By-2). When this goal terminates successfully (returning the number 5), a chunk is created for it (bottom right of Figure 5-2). The encoding component encodes the number 10 into the symbol E2; the decoding component decodes from the symbol D2 to the

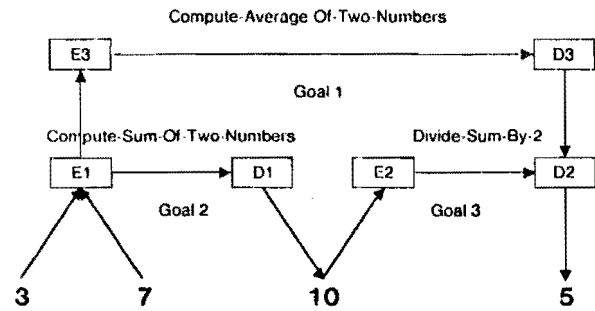


Figure 5-2: Sample chunks created for the hierarchy in Figure 5-1.

number 5; and the connection component connects E2 to D2 (in the presence of an active goal 3). In contrast to the chunk for goal 1, this chunk can be used in more than one task situation. It can be used whenever goal 1 generates a sum of 10, whether it does it by adding 3 and 7, 5 and 5, or any other pair of numbers. This is a form of transfer of training.

Following the termination of goal 3, goal 1 is reactivated, and terminates successfully (returning the number 5). No chunk is created for goal 1 because its subgoals were not processed by chunks. At this point, the task is complete.

Given what was learned during the performance of this task, the next time the same task is performed things will go differently. As soon as the task is restarted (again with the values 3 and 7), the encoding component from the chunk for goal 2 executes, placing E1 in the working memory. Goal 1 is activated, and then suspended in favor of goal 2. At this point, the connection component for goal 2 executes, generating D1 and successfully completing goal 2. D1 is decoded to the number 10, which is then immediately reencoded to E2 by the encoding component for goal 3. Following the subsequent reactivation and suspension of Goal 1, Goal 3 is activated. The connection component for Goal 3 executes, generating D2, and returning D2 as the result to goal 1. This time, when goal 1 terminates, a chunk is created (top of Figure 5-2), because both of the subgoals were processed by chunks.

The encoding component for this chunk builds upon the existing encodings by encoding E1 to a new symbol (E3); it does not go straight from goal 1's primitive parameters (3 and 7). This happens (and causes hierarchical encoding) because, for this instance of goal 1, E1 is the implicit parameter, not 3 and 7. Recall from Section 4 that the implicit parameters of a goal consist of those pieces of the goal's initial state that are examined during the goal's performance. E1 is generated before goal 1 is activated (so it is part of the goal's initial state), and examined by the connection component for goal 2. On the other hand, neither of the objects representing the numbers 3 and 7 are examined during the processing of goal 1. Therefore, E1 is an implicit parameter (and included in the chunk), while the numbers 3 and 7 are not.

The decoding component is created in a similarly hierarchical fashion. It decodes from a new symbol (D3) to D2. This occurs because D2 (and not the number 10) is the result of goal 1. It never became necessary to decode D2, so it was passed directly up as the result of both goals 3 and 1. The connection component of this chunk links E3 to D3 in a straightforward manner.

If the same task is performed yet again, the encoding components immediately generate E1, followed by E3. Goal 1 is activated, and its connection component executes, generating D3 and completing goal 1. If the result is needed by some part of the system outside of the hierarchy, it will be decoded to D1, and then to the number 10.

The example that we have just gone through outlines the basics of how the chunking mechanism works. The next step is to look at the more complex situation of the Seibel task (Figure 4-1). Chunking starts in this structure with the terminal goals (numbers 2, 4, 5, 7, 10, 11, and 12). Take goal 11 (Get-Stimulus-X), for example. Successful completion of this goal requires retrieving from the working memory the representation of a stimulus that has been perceived, and then generating a piece of information representing the horizontal location (X) of that stimulus. The parameter for this goal is the stimulus — an implicit parameter — and the result is the location. In the chunk for this situation, the encoding and decoding components are trivial. They simply recode from the single parameter to a new symbol, and from a second new symbol to the result. The connection component tests for the presence of an active Get-Stimulus-X goal and the encoding symbol, and produces the decoding symbol.

Goal 10 (Get-On-Light-Stimulus) presents a slightly more complicated case than goal 11. The goal has both explicit and implicit parameters. The explicit parameters (Min-X and Max-X) define the region of the stimulus display in which an on-light should be found. The implicit parameter is the actual external representation (in the stimulus display) of the on-light that is found. The result of this goal is the internal representation of that on-light, as it appears in the working memory. Just as before, the implicit parameter and the result form the basis for trivial encoding and decoding components. However, the explicit parameters are different. They do not exist as separate entities in the working memory; instead they act as if they were augmentations to the name of the goal. Therefore, they appear along with the name of the goal, as part of the connection component.

As a final example of the workings of the chunking mechanism, consider how it creates the kinds of multiple light-button chunks produced by the chunking mechanism in Rosenbloom and Newell [In press]. The locus of these chunks can be found at the recursive step in the goal hierarchy. The root (and recursive) goal in the hierarchy (Do-Lights-If-Any) has implicit parameters that represent lights in the stimulus display, and generates results that are the button presses for those lights. The earliest chunks that can be created for this goal are those at the bottom of the recursion; that is, goals to process manageable segments of the display. Each of these chunks will represent a single on-light in a region, a region of solid on-lights, or a region with no on-lights. Once the chunks exist for goals 13 and 14 (and their sibling goal 7: the predicate One-Light-On?) in a single situation, the parent goal (goal 6: Do-Off-And-On) can be chunked. This yields a new chunk for the combined situation in both segments. This process continues up the hierarchy until goal 1 is chunked for that level of recursion. But goal 1 at that level is just goal 13 or 14 at the next level up. Therefore, gradually (since these chunks are acquired one at a time), the level of aggregation of segments covered by chunks increases.

This process always leads to light-button chunks for contiguous light-button pairs. It does not lead to chunks for disjoint patterns such as just the two extreme right and left light-

button pairs. This is not a limitation on the generality of the chunking mechanism. Instead, it is a function of the goal structure employed. A different goal structure (reflecting a different processing strategy) could lead to the creation of such disjoint chunk patterns.

One of the strong task-dependencies present in the Rosenbloom and Newell [In press] chunking mechanism was that encoding productions had to have a condition added to them that assured that no on-light appeared between the two patterns being chunked together. This was so, even though the information that this condition was necessary appeared nowhere in the task algorithm. In the hierarchy in Figure 4-1, such conditions are generated naturally from the goals for processing segments of the display with no on-lights in them. We see that the task dependencies show up in the goal hierarchy, not in the chunking mechanism.

The following list of points summarizes the key aspects of chunking as it applies to goal hierarchies.

- Each chunk represents a specific goal with a specific set of parameter values. It relates the parameter values to the results of the goal.
- Chunks are created through experience with the goals processed.
- Chunks are created bottom up in the goal hierarchy.
- Chunks consist of connection, encoding, and decoding components.
- Chunk encoding and decoding are hierarchical, parallel, goal-asynchronous processes that operate on goal parameters and results (respectively).
- Chunk connection is a serial, goal-synchronous process that generates (possibly encoded) results from (possibly encoded) parameters.
- Chunks improve performance by replacing the normal processing of a goal (and its subgoals) with the faster processes of encoding, connection, and decoding.

6. The Xaps3 Architecture

The *Xaps3* production-system architecture was designed to be used in the modeling of goal hierarchies and chunking. It is a new architecture, but it builds upon the work done in the development of the *Xaps2* architecture. In their presentation of *Xaps2*, Rosenbloom and Newell [In press] began by discussing a set of *constraints* that must be met by any architecture within which the chunking theory of learning is implemented. We don't claim that these constraints are known to be necessary — the arguments are not that tight. But this permits (and encourages) attempts to show how the constraints can be circumvented.

The same constraints still hold — as they must if they are really constraints — for the design of the *Xaps3* architecture. More recently, two new constraints have been formulated from the consequences of the chunking theory. These new constraints rule out *Xaps2*, and have led to the design of the *Xaps3* architecture. Though it is a direct descendent of *Xaps2*, the constraints have forced a number of significant changes in the *Xaps3* design.

The first new constraint is the *crypto-information constraint*.

- The *crypto-information constraint*: any system that learns through experience cannot contain crypto-information if it wants to guarantee correct learning.

Crypto-information — "hidden" information — is information that the architecture accesses while making decisions, but that is not accessible to "programs" running within the architecture. In the *Xaps2* architecture, activation is a form of crypto-information. It is used by the architecture to, among other purposes, decide which of multiple instantiations of a production should execute on a cycle. It is crypto-information because knowledge about activation values cannot be represented in the productions executed within the architecture. Another example of crypto-information is the information about working-memory recency used for conflict resolution in the *Ops* languages [Forgy, 1979].

The use of activation in *Xaps2* is a good example of how crypto-information can lead to incorrect performance. Suppose there is an object *A* that has an activation of 0.5, and an object *B* with an activation 0.3. Suppose also that there is a production that matches both *A* and *B*, generating two instantiations. When this situation first occurs, the instantiation that matches *A* will execute because of its greater activation. From this experience, the system learns what action to perform under these circumstances; that is, the action associated with the *A* instantiation. At some later point both *A* and *B* may again both be represented, but this time with the activations reversed. Even though the action associated with *B* is now the correct one, its previous experience tells the system to do the action associated with *A*, because the information about relative levels of activation could not be represented in the record of the previous experience. Not only would the previous information be incorrect, but there would in fact be no way to ever learn the correct information.

It is important to note that this constraint does not rule out all activation-based production-system architectures. If either the activation is not used as a basis for an architectural decision, or sufficient knowledge about activation levels is representable within productions, then activation is not a problem, because it is no longer a form of crypto-information.

Though activation was a focal point of the *Xaps2* design, to meet the demands of this constraint, *Xaps3* has been designed to work without it. It is thus a more traditional, purely symbolic, production-system architecture.

The second new constraint is the *goal-architecturality constraint*.

- The *goal-architecturality constraint*: the representation and processing of goals must be defined in the architecture itself; it cannot be left up to the discretion of productions.⁴

The reason behind this constraint can be traced to the chunking mechanism's need to understand how goals are represented and processed. This requirement implies that goals must be understood by the architecture, because the chunking mechanism is itself an architectural mechanism.

One way for the architecture to understand the processing of goals, is for the algorithm to be defined within the architecture itself. The obvious alternative — production-defined processing of goals — requires the chunking mechanism to be able to abstract the algorithm and representation from the productions, and tune itself to whatever scheme is being used. This requires considerably more intelligence than we are willing to ascribe to the chunking mechanism, or for that matter, to any mechanism built into the architecture. For a system to exhibit truly adaptive behavior, it must be able to apply its full knowledge and learning capabilities to any task requiring intelligence. This is feasible at the program level, but not at the architectural level.

In response to the goal-architecturality constraint, the goal processing that occurred at the level of productions in *Xaps2*, has been moved down into the architecture of *Xaps3*.

Xaps3 is one architecture sitting within the design space delimited by the constraints so far determined. In the remainder of this section we describe the *Xaps3* architecture (see Rosenbloom [1983] for more details). It is like *Xaps2* in the structure of working-memory objects, and in allowing parallelism at the level of production execution, but differs in its lack of activation, and in that goal processing and chunking are built in at the architectural level⁵. This description is divided into sections on the working memory, the production memory, the cycle of execution, the processing of goals, and the chunking mechanism.

6.1 Working Memory

With respect to symbols, the working memory in *Xaps3* is nearly identical to the one in *Xaps2*. It consists of an unordered set of *objects*, representing all of the types of information required in an information processing system (except for the types of information represented in productions): goals, patterns of stimuli and responses, intermediate computations, and so forth. As was true in *Xaps2*, each object is a token — in the sense of the classical type-token distinction. It is uniquely specified by a general *type* and a unique *identifier*. The *type* is a symbol that can be common among several objects, such as *Goal*. The *identifier* is a unique symbol for the object, allowing multiple objects of the same type to be simultaneously in working memory. A new identifier is generated dynamically by the architecture whenever a new object is created.

Like objects in *Xaps2*, *Xaps3* objects also have an optional set of *attributes*. However, unlike *Xaps2* objects, each attribute has only one value. A successfully completed instance of a *Press-Button-At-Horizontal-Location* goal (with a *type* of *Goal*, and an identifier of *Object127*), looks like the following example⁶.

```
(Goal Object127
  [NAME Press-Button-At-Horizontal-Location]
  [STATUS Succeeded] [RESULT-TYPE Response])
```

The interpretation of two of the attributes (*NAME* and *STATUS*) should be obvious. They tell the system that this goal is an instance of the *Press-Button-At-Horizontal-Location* goal that has completed successfully. The third attribute

⁴ Anderson [1982a] has previously made a similar argument for the placement of goal processing in the architecture of the *Act* production system [Anderson, 1983].

⁵ See Anderson [1983] and Sauers and Farrell [1982] for other attempts at integrating the processing of goals into the architecture of a production system.

⁶ The format has been altered slightly for clarity of presentation.

(RESULT-TYPE) specifies the type of object to be returned as the result of this goal (see Section 6.4 for the details).

During the lifetime of a working memory object, two types of auxiliary tags are kept. The first one marks the identifier of the goal that was active — there being at most one — when the object was created (If there was no goal active at the time, then the special symbol <None> is used). This tag is used for determining the object's status as being either part of the local or initial state of the current goal. The second tag marks the identifier of the goal active when the object was last examined by a production. This tag, in combination with the first one, allows the chunking mechanism to determine the implicit parameters of goals.

6.2 Production Memory

There is a single homogeneous production memory in *Xaps3*. The productions in it are similar to the productions in *Xaps2* and to those in the *Ops* languages. They have three parts: a name, a list of one or more conditions, and a list of one or more actions. Here is an example production taken from the implementation of the Seibel task (Figure 4-1)⁷.

```
(DefProd SubGoal/Do-Lights/Do-Press-All-Buttons
  ((Goal <Exists> [NAME Do-Lights] [STATUS Active]
    [MINIMUM-LOCATION = Min-Loc]
    [MAXIMUM-LOCATION = Max-Loc])
   (Goal {<Local> <Exists>} [NAME No-Light-Off?]
    [STATUS Succeeded]))
  →
  ((Goal <New-Object> [Name Do-Press-All-Buttons]
    [STATUS Warn] [RESULT-TYPE Response]
    [MINIMUM-LOCATION = Min-Loc]
    [MAXIMUM-LOCATION = Max-Loc]))))
```

The name is purely for the convenience of the programmer, it doesn't affect the processing of the system in any way. Each condition is a pattern to be matched against the objects in working memory. A condition pattern contains a *type* field (specifying the type of the object to be matched), an *identifier* field (containing several different types of information), and an optional set of patterns for attributes. Condition patterns are primarily built from *constants* and *variables*. Constants are signified by the presence of the appropriate symbol in the pattern, and only match objects that contain that symbol in that role. Some example constants in production *SubGoal/Do-Lights/Do-Press-All-Buttons* are *Goal*, *MINIMUM-LOCATION*, and *Succeeded*. Variables are signified by a symbol (the name of the variable) preceded by an equal sign. They match anything appearing in their role in an object. An example is *= Min-Loc* in the sample production. All instances of the same variable within a production must be bound to the same value in order for the match to succeed.

The identifiers in production *SubGoal/Do-Lights/Do-Press-All-Buttons* are specified by the special symbol <Exists>. Such a condition succeeds if

there is any object in working memory that matches the condition. If there is more than one such object, the choice is conceptually arbitrary (actually, the first object found is used). Only one instantiation is generated. If the identifier is specified as a variable, the condition still acts like an exists condition, but the identifier can be retrieved, allowing the object to be modified by the actions of the production.

The complete opposite of an exists condition, is a not-exists condition — commonly called a negated condition. When the symbol <Not-Exists> appears in the identifier field, it signifies that the match succeeds only if there is no object in working memory that matches the remainder of the condition pattern.

The second condition in the sample production contains the symbol <Local>, as well as <Exists>, in the identifier field (they are conjoined syntactically by curly-brackets). When this symbol is added to an identifier pattern of any type — constant, variable, exists, and not-exists — it signifies that the condition should only be matched against those objects in working memory that are local to the active goal. This information — provided by the objects' *created-by* tags — is a means by which the productions that work on a goal can determine which objects are part of their local context.

The remainder of the condition specifies a pattern that must be met by the attribute-value pairs of working-memory objects. These patterns are built out of constants, variables, built-in predicates (such as <Greater-Than>), and general *Lisp* computations (via an escape mechanism). In addition, any of the above forms can be negated, denoting that the condition only matches objects that do not match the pattern.

There is only one kind of action in *Xaps3* — modify working memory. As in *Xaps2*, the interface to the outside world is through working memory, rather than through production actions. Actions can create new objects in working memory and, under certain circumstances, modify existing objects. A new object is created when the action has an identifier of <New-Object>, by replacing the identifier with a newly generated symbol, instantiating the variables with their values computed during the match, and replacing calls to *Lisp* functions with their values (via another escape mechanism). An existing object can be modified by passing its identifier as a parameter from a condition. As Discussed in Section 4, only objects local to the current goal can be modified.

There are no production actions that lead to the deletion of values or objects from working memory. A value can be removed only by superceding it by another value. As discussed in Section 4, objects go away when they are no longer part of the current context⁸. No explicit mechanism for deletion has proved necessary, so none has been included.

6.3 The Cycle of Execution

Xaps3 follows the traditional *recognize-act* cycle of production-system architectures, with a few twists thrown in by the need to process goals. The recognition phase begins with the *match*, and finishes up with *conflict resolution*. The *cycle number* — the simulated time — is incremented after the recognition phase, whether any productions are executed or not. During the act phase, productions are executed.

⁷ As with the format of working-memory objects, the format of this production has been slightly altered to increase the clarity of presentation.

⁸ This mechanism bears a relationship to the *dampening* mechanism in the *Act* architecture [Anderson, 1976].

6.3.1 Recognition

The match phase is quite simple. All of the productions in the system are matched against working memory in parallel (conceptually), to yield a set of legal instantiations — at most one per production. Each instantiation consists of a production name and a set of variable bindings for that production that yield a successful match. This set of instantiations is then passed in its entirety to the conflict resolution phase⁹, where they are winnowed down to the set to be executed on the current cycle. This winnowing is accomplished via a pair of conflict-resolution rules.

The first rule is *goal-context refraction*. It states that a production instantiation can fire only once within any particular goal context. It is a form of the standard *Ops* refractory inhibition rule [Forgy, 1979], differing only in how the inhibition on firing is released. With the standard rule, the inhibition is released whenever one of the working-memory objects on which the instantiation is predicated, has been modified. With goal-context refraction, inhibition is released whenever the system leaves the context in which the instantiation fired. If the instantiation could not legally fire before the context was established, and it could fire both during and after the context was established, then the instantiation must be based, at least in part, on a result generated during the context and returned when the context was left. Therefore, the instantiation should be free to fire again to reestablish the still relevant information.

The second rule — the *parameter-passing bottleneck* — states that only one parameter-passing instantiation can execute on a cycle (conceptually selected arbitrarily). This rule first appeared in a slightly different form as an assumption in the *Hpsa77* architecture [Newell, 1980a]. It will be justified in Section 6.5.

6.3.2 Action

All of the instantiations that make it through the conflict resolution phase are fired (conceptually) in parallel. Firing a production instantiation consists of: (1) overwriting the *examined-by* tags of the working-memory objects matched by the instantiation, with the identifier of the active goal; (2) replacing all variables in the actions by the values determined during the match phase; (3) evaluating any *Lisp* forms; and (4) performing the actions. This can result in the addition of new objects to working memory, and the modification of existing local objects. If conflicting values are simultaneously asserted, the winner is selected arbitrarily. This does not violate the crypto-information constraint because the architecture is using *no* information in making the decision. If the performance system works correctly, even when it can't depend on the outcome of the decision, then the learned information will not lead to incorrect performance.

6.4 Goal Processing

In Section 4 we described the processing of goals at an abstract level. In this section, we make that processing concrete, by describing how it is implemented within the *Xaps3* architecture. The goals themselves, unlike chunks, are just data structures in working memory. A typical *Xaps3* goal goes through four phases in its life. The current phase of a goal is represented explicitly at all times by the value associated with

the *STATUS* attribute of the working-memory object representing the goal.

In the first stage of its life, the goal is *desired* — at some point, but not necessarily right then, the goal should be processed. Productions create goal desires by generating a new object of type *Goal*. Each new goal object must have a *NAME* attribute, a *STATUS* attribute with a value of *Want*, and a *RESULT-TYPE* attribute. In addition, it may have any number of other attributes, specifying explicit parameters to the goal. The value of the *RESULT-TYPE* attribute specifies the type of the results that are to be returned on successful completion of the goal. All local objects of that type are considered to be results of the goal. Results are marked explicitly so that they won't be flushed when the context is left, and so that the chunking mechanism will know to include them in the chunks for the goal.

Leaving the expression of goal desires under the control of productions allows goals to be processed by the architecture, while still leaving the structure of the hierarchy under program (that is, production) control. The architecture controls the transition to the second, *active* phase of the goal's life. At most one goal is active at any point in time: The architecture attempts to activate a new goal whenever the system is at a loss as to how to continue with the current goal. This occurs when there is an empty conflict set; that is, there is no production instantiation that can legally fire on the current cycle. When this happens, the system looks in working memory to determine if there are any subgoals of the current goal — those goals created while the current goal was active — that are desired. If such a subgoal is found, it is made the active goal, and the parent goal is *suspended* by replacing its *STATUS* with the identifier of the newly activated subgoal. If more than one desired subgoal is found, one is arbitrarily selected (actually, the last one found is used).

Suspension is the third phase in the life of a goal (it occurs only for non-terminal goals). Replacing the *STATUS* of the parent goal with the subgoal's identifier accomplishes two things: it halts work on the goal, because the productions that process goals all check for a *STATUS* of *Active*; and it maintains the control stack for the goal hierarchy. A suspended goal remains suspended until its active subgoal terminates, at which time it returns to being active. If a goal has more than one subgoal, it oscillates between the active and suspended states.

If no progress can be made on the active goal, and there are no desired subgoals, then the system has no idea how to continue making progress, so it terminates the active goal with a *STATUS* of *Failed*. Following termination, the goal is in its fourth and final phase of life. In addition to a failure termination, goals can be terminated with a *STATUS* of *Succeeded*. There are no uniform criteria for determining when an arbitrary goal has completed successfully, so it has been left to productions to assert that this has happened. This is done via the creation of an object of type *Succeeded*. When the architecture detects the presence of such an object in working memory, it terminates the current goal, and reactivates its parent.

At goal termination time, a number of activities occur, in addition to the obvious one of changing the active goal. The first two activities occur only on the successful termination of a goal. As discussed in the following section, the first step is the possible creation of a chunk. The second step is to return the results of the terminating goal to its parent goal. This is accomplished by altering the *created-by* tags of the results so that it looks as if they were created by the parent goal. The third step is to delete all of the objects from working memory created during the goal to be terminated. The fourth, and final, step is to enable result decoding if it is appropriate.

⁹ For purposes of efficiency, these two phases are actually intermingled. However, this does not change the behavior of the system at the level at which we are interested.

6.5 Chunking

As we saw in Section 5, chunking improves performance by enabling the system to use its experience with previous instances of a goal to avoid expanding the goal tree below it. In this section we detail how this has been implemented within the *Xaps3* architecture — yielding a working, task-independent, production-system practice mechanism. We begin by describing how chunks are used, and conclude with how they are acquired.

6.5.1 The use of chunks

The key to the behavior of a chunk, lies in its connection component. When a goal is proposed in a situation in which a chunk has already been created for it, the connection component of that chunk substitutes for the normal processing of the goal. This is accomplished in *Xaps3*, by hooking the connection to the desired state of the goal. The connection is a production containing a condition testing for the existence of a desired goal and a condition testing for the encoding of the goal's parameters¹⁰. It has two actions; one marks the goal as having succeeded, and the other asserts the goal's encoded result. If there is a desired goal in working memory in a situation for which a connection production exists, the connection will be eligible to fire. Whether (and when) it does fire, depends on conflict resolution. Connection productions are subject to the *parameter-passing bottleneck* conflict-resolution rule, because they pass the identifier of the desired goal as a parameter to the action that marks the goal as having succeeded. Therefore, only one goal can be connected at a time — part of the *bottleneck constraint* [Rosenbloom & Newell, In press] — and only one connection can execute for any particular goal instance (removing a source of possible error).

If the connection does fire, it removes the need to activate and expand the goal, because the goal's results will be generated directly by the connection (and decoding), and the goal will be marked as having succeeded. If instead, no connection production is available for the current situation of a desired goal, then eventually the production system will reach an impasse — no productions eligible to fire — and the goal will be activated and expanded. Therefore, we have just the behavior required of chunks — they replace goal activation and expansion, if they exist.

This behavior is, of course, predicated on the workings of the encoding and decoding components of the chunk¹¹. The encoding component of a chunk must execute before the associated connection can. In fact, it should execute even before the parent goal is activated, because the subgoal's encoded symbol must be part of the parent goal's initial state. Recall, that encodings are built up hierarchically. If the parent goal shares all of the parameters of one of its subgoals, then the encoding of the parent goal's parameters is partly based on the encoding generated for the subgoal. This behavior occurs in *Xaps3* because the encoding components are implemented as goal-free productions that do not pass parameters. They fire (concurrently with whatever else is happening in the system) whenever their parameters are in working memory (subject to conflict resolution). If the parameters exist before the parent

goal becomes active, as they must if they are parameters to it, then the encoded symbol becomes part of the parent goal's initial state.

As stated in Section 4, the decoding component must decode an encoded result when it will be needed. Each decoding component is a production that keys off of the non-primitive result pattern generated by the connection production, and off of an object of type *Decode*. When the architecture determines that decoding should occur, it places a *Decode* object in working memory, with the type of the object to be decoded specified by the *TYPE* attribute¹². The actions of the decoding production generate the component results out of which the non-primitive pattern was composed.

6.5.2 The acquisition of chunks

A complete specification of the chunk acquisition process must include the details of when chunks can be acquired, and from what information they are built. A chunk can be acquired when three conditions are met. The first condition is that some goal must have just completed. The system can't create a chunk for a goal that terminated at some point in the distant past, because the information on which the chunk must be based, is no longer available. Chunks also are not created prior to completion (on partial results). Chunks are simple to create in part because they summarize all of the effects of a goal. If chunks were created part way through the processing of a goal — for partial results of the goal — then a sophisticated analysis might be required in order to determine which parameters affect which results, and how. This is not really a limitation on what can be chunked, because any isolatable portion of the performance can be made into a goal.

The second condition is that the goal must have completed successfully. Part of the essence of goal failure, is that the system does not know why the goal failed. This means that the system does not know which parameter values have lead to the failure; thus, it can't create a chunk that correctly summarizes the situation. Chunking is success-driven learning, as opposed to failure-driven learning (see for example, Winston [1975]).

The third, and final, condition for chunk creation, is that all of the working-memory modifications occurring since the goal was first activated, must be attributable to that goal, rather than to one of its subgoals. This condition is implemented by assuring that no productions were fired while any of the goal's subgoals were active. All of the subgoals must either be processed by a chunk, or fail immediately after activation — failure of a subgoal, particularly of predicates, does not necessarily imply failure of the parent goal.

To summarize, a chunk is created after the system has decided to successfully terminate a goal, but before anything is done about it (such as marking the goal as succeeded, returning the results, and flushing the local objects from working memory). At that point the goal is still active, and all of its information is still available.

Most of the information on which the chunk is based, can be found in working memory (except, see below). The first piece of information needed for the creation of a chunk, is the name (and identifier) of the goal that is being chunked. This information is found by retrieving the object representing the active goal. The goal's explicit parameters are also available as attribute-value

¹⁰ Matters are somewhat more complicated for negated (<Not-Exists>) conditions [Rosenbloom, 1983].

¹¹ For efficiency, the encoding and decoding components are not created if there is only one parameter or result (respectively).

¹² Matters are actually somewhat more complicated [Rosenbloom, 1983].

pairs on the goal object. Given the goal's identifier, its implicit parameters are found by retrieving all of the objects in working memory that were part of the goal's initial state — that is, their *created-by* tag contains an identifier different from that of the active goal — and were examined by a production during the processing of the active goal. This last type of information is contained in the objects' *examined-by* tags. The goal's results are equally easy to find. The system simply retrieves all of the goal's local objects that have a type equal to the goal's *RESULT-TYPE*. Because the parameter and result information is determined from the constant objects in working memory, chunks themselves are not parameterized. Each chunk represents a specific situation for a specific goal.

The one complication to the clean picture of chunk acquisition so far presented, concerns the use of negated conditions during the processing of a goal. When a negated condition successfully matches working memory, there is no working-memory object that can be marked as having been examined. Therefore, some of the information required for chunk creation cannot be represented in working-memory. The current solution for this problem is not elegant, but it works. A temporary auxiliary memory is maintained, into which is placed each non-local negated condition occurring on productions that fire during the processing of the goal (local negated conditions can be ignored, because they do not test the initial state). This memory is reinitialized whenever the goal that is eligible to be chunked changes. Before the conditions are placed in this memory, they are fully instantiated with the values bound to their variables by the other conditions in their production.

7. Results

In this section we present some results from applying this architecture to the Seibel and compatibility tasks. A more complete presentation of these results can be found in Rosenbloom [1983]. For the Seibel task, two different sequences of trials were simulated. The first sequence is the same as the one used in Rosenbloom and Newell [In press]. The simulation completed 268 trials before it was terminated¹³. A total of 682 productions were learned. On the second sequence of trials — from a newly generated random permutation of the 1023 possibilities — 259 trials were completed before termination. For this sequence, 652 productions were learned.

Figure 7-1 shows the first sequence as fit by a general power law. Each point in the figure represents the mean value over five data points (except for the last one, which only includes three)¹⁴. For this curve, the asymptote parameter (*A*) has no effect. Only *E*, the correction for previous practice, is required to straighten out the curve. At first glance, it is nonsensical to talk about previous practice for such a simulation, but a plausible interpretation is possible. In fact, there are two independent explanations — either or both may be responsible.

The first possibility is that the level at which the terminal goals are defined is too high (complex). If the "true" terminals are

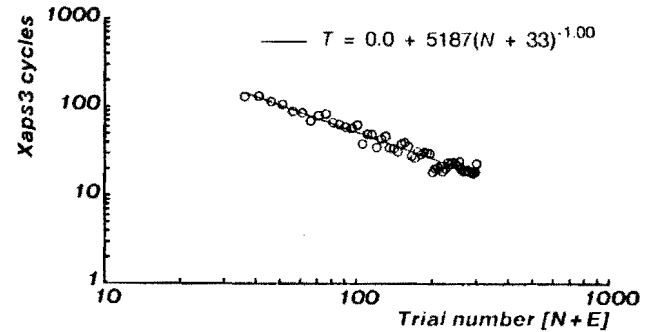


Figure 7-1: General power law fit to 268 simulated trials of the Seibel [1963] task.

more primitive, then chunking starts at a lower level in the hierarchy. One view of what chunks are doing is turning their associated (possibly non-terminal) goals into terminal goals for particular parameter situations. During preliminary bottom-up chunking, the system would eventually reach the lowest level in the current hierarchy. All of the practice prior to that point is effectively previous practice for the current simulation.

The other source of previous practice, is the goal hierarchy itself. At the beginning of the simulation, this structure is already known perfectly. However, there must be a process of *method acquisition* by which the subject goes from the written (or oral) instructions to an internal goal hierarchy. Though method acquisition does not fall within the domain of what we have here defined as "practice", we will propose in Section 9 a scheme whereby chunking may lead to a mechanism for method acquisition.

One problem that cropped up in the simulations of the earlier model of chunking [Rosenbloom & Newell, In press], was that learning was demonstrably too fast. In order to achieve reasonable curves, learning was artificially slowed through the addition of a rate parameter. So far, such a parameter has not been required in the current model, because there is much more to be learned. Chunks are still learned for light-button combinations, as in the earlier model, but these chunks are for only 1 of the 12 goals in the hierarchy in Figure 4-1 (not counting the recursive goals at 13 and 14). Additional chunks are required for the 11 other goals in the structure.

It is encouraging that the current model overcomes some of the minor flaws of the earlier system, but the major focus of this work is on the generality of the model. In addition to the Seibel task, the system so far has been applied to fourteen tasks from three different compatibility experiments [Fitts & Seeger, 1953; Morin & Forrin, 1962; Duncan, 1977]. As a sample of these results, Figure 7-2 shows a pair of simulated practice curves for two tasks from Fitts and Seeger [1953]. These curves contain fifty trials each, aggregated by five trials per data point.

This article is not the appropriate place to discuss the issues surrounding whether the simulated practice curves are truly power laws or something slightly different (such as an exponential). Suffice it to say that we obtain a mixture of exponential, power law, and ambiguous curves. These results roughly follow the predictions of the approximate mathematical analysis of the chunking theory appearing in Rosenbloom [1983]. They also fit with the observation that the human curves tend to be most exponential for the simplest tasks — the

¹³ At this point the *FranzLisp* system — which appeared to not be garbage collecting in the first place — refused to allocate any more memory for the simulation.

¹⁴ This data appear noisier than that in Figure 2-2 showing the human data from Seibel [1963]. This is accounted for by each point in Figure 2-2 being the mean of 1023 data points, rather than the 5 data points of this figure.

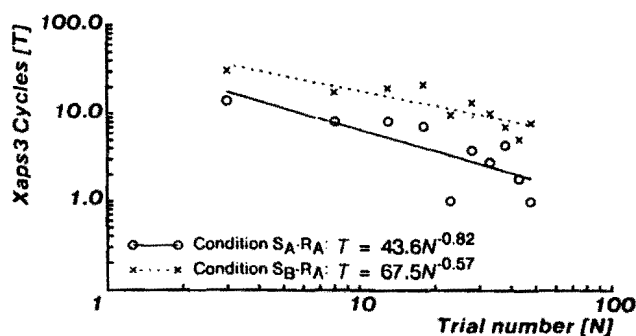


Figure 7-2: Simulated practice curves for conditions S_{A-R_A} and S_{B-R_A} from Fitts and Seeger [1953]. The latter curve is the average over two hierarchy variations.

compatibility tasks are among the simplest tasks for which we have human practice data. For more on this issue, see Newell and Rosenbloom [1981] and Rosenbloom [1983].

8. Relationship to Previous Work

In addition to the already mentioned relationship to caching, the current formulation of the chunking theory of learning provides an interesting point of contact between three previously disparate concepts: (1) classical chunking, (2) memo functions [Michie, 1968], and (3) production composition [Lewis, 1978; Neves & Anderson, 1981; Anderson, 1982b]. Classical chunking has already been described in Section 3. A memo function¹⁵ is a function with an auxiliary table added. Whenever the function is evaluated, the table is first checked to see if there is a result stored with the current set of parameter values. If so, it is retrieved as the value of the function. Otherwise, the function is computed and the arguments and result are stored in the table.

Production composition is a learning scheme whereby new productions are created through the combination of old ones. Given a pair of productions that execute successively, their composition is created from their conditions and actions. The condition side of the new production consists of all of the conditions of the first production, plus those conditions from the second production that do not match actions of the first production. The action side consists of all of the actions from both of the productions. The resulting composition is a single production that accomplishes the combined effects of the older pair of productions.

Though these mechanisms were proposed in quite different contexts, and bear little obvious relationship to each other, the current formulation of the chunking theory of learning has strong ties to all three. It (1) explains how classical chunks can be created and used; (2) caches the results of computations, as do memo functions; and (3) results in productions similar to those generated by goal-directed production composition [Sauers & Farrell, 1982; Anderson, 1983]. The chunking theory differs from these three mechanisms in a number of ways [Rosenbloom, 1983], but at the same time it occupies a common ground

¹⁵ Memo functions themselves are derived from the earlier work by Samuel [1959] on creating a rote memory for the values of board positions in the game of checkers.

between them. This leads us to propose that all four mechanisms are different manifestations of a single underlying store-versus-compute idea. We feel that the chunking theory has a lot going for it, but it is probably too early to know what formulation will work out to be the correct one in the long run for a general practice mechanism.

9. Expanding the Scope of Chunking

In this work we have shown how chunking can provide a model of practice for tasks that can already be accomplished. Performance is sped up, but not qualitatively changed. It is interesting to ask whether chunking can be used to implement any of the other more complex forms of learning — such as method acquisition, concept formation, generalization, learning by being told, and expectation-driven learning. Chunking does not directly accomplish any of these forms of learning, and on first glance, the table look-up nature of chunking would seem to preclude its use in such sophisticated ways. However, we would like to at least raise the possibility that the scope of chunking may extend much further.

Method acquisition is the prototypical case of "hard" learning. Given an arbitrary task, how does the system first construct a method (goal structure) for it? The answer may lie in the integration of chunking with problem-space search. The *problem space hypothesis* [Newell, 1980b; Laird & Newell, 1983] states that intelligent agents are always performing in a problem space. At any instant, the agent will have a *goal* that it is attempting to fulfill. Associated with that goal is a *problem space* in which the goal can be pursued. The problem space consists of a set of *states*, a *problem* (initial and desired states), a set of *operators* that move the agent between states, and *search control* information which assists in guiding the agent efficiently from the initial to the desired state.

One way to look at the goal structures used in the current model is as degenerate problem spaces. The goals are certainly there as needed. The current state can be identified with the contents of working memory, with the set of possible states being defined as all possible working memories. The initial state for a goal is its initial state in the production system. The desired state is defined by the production that tests for goal success. The operators are the subgoals — an operator is simply a goal that the system knows how to accomplish. The search control consists of those productions that specify which subgoals to do when. The one thing missing is the ability to search — the essence of working in a problem space.

Adding a search capability to the current system could be the first step towards method acquisition. It would allow the system to perform tasks that it does not already "know" how to do. The second step must be to show how chunking could lead from an inefficient search, to the development of an efficient goal-structured method.

This may go as follows. Suppose the system is at some state for which it does not know how to make further progress towards the desired state. What should it do? It should create a subgoal in which it can figure out which operator to do next. This subgoal could be pursued in any number of ways. One of the simplest is to ask someone else what to do next (leading to learning by being told). Another possibility is that a look-ahead search can be performed. If a path is found to the desired goal, the first operator on that path is returned as the result of the search. This is of course the paradigm case for a game-tree search, in which the goal is to win the game, but choosing the

next move (operator) may be problematic. Game playing programs effectively go into a choose-operator subgoal in which they simulate future paths in the game tree. They evaluate the alternatives, and return information allowing the selection of an operator in the top-level problem space.

Given such an augmentation to the current system, it would require only a small conceptual leap to imagine how chunking could be applied to these choose-operator subgoals. The resulting chunk would test those aspects of the situation that were examined during the evaluation of the operators, and yield the next operator (subgoal) to pursue. This is a search control chunk for the problem space — part of the backbone of the goal hierarchy.

One limitation of the current chunking mechanism that such a method-acquisition scheme could alleviate, is the inability of chunks to implement parameterized operators. Chunking always creates a totally specified piece of knowledge. As currently formulated, it cannot create the kinds of parameterized operators used as terminal nodes in the goal hierarchies (though it can achieve a form of separation of parameters¹⁶). On the other hand, problem spaces are inherently parameterized by their initial and desired states. Therefore, it may be that it is not necessary for chunks to create parameterized operators. These operators can come from another source (problem spaces). Chunks would only be responsible for making these operators more efficient.

In summary, doing method acquisition via chunking in problem spaces looks feasible, at least at the rather cursory level of detail at which it has been described here. If it does pan out, we hope to be able to formulate a number of the other difficult learning problems within this paradigm. The complications would appear as problem-solving in problem spaces, while the chunking mechanism would remain simple, merely recording the results generated by the problem-solving system.

10. Conclusion

At the beginning of this investigation we set out to develop a generalized, task-independent model of practice, capable of producing power-law practice curves. The model was to be based on the concept of chunking, and to be used as the basis (and a source of constraint) for a production-system practice mechanism. All of this has been accomplished. The generalized model that we have developed is based on a goal-structured representation of reaction-time tasks. Each task has its own goal hierarchy, representing an initial performance algorithm.

When a goal is successfully completed, a three-part chunk can be created for it. The chunk is based on the parameters and results of the goal. The encoding component of the chunk encodes the parameters of the goal, yielding a new symbol representing their combination. The connection component of the chunk ties the encoded parameter symbol to an encoded symbol for the results of the goal. The decoding component of the chunk, decodes the new result symbol to the results out of which it is composed.

¹⁶One form of parameter independence achievable with the current system is the location-independent recognition of patterns. This can be accomplished by having one goal (or a set of goals) to recognize and relate the components of the pattern via their relative locations, and a second goal to determine the absolute location of the entire pattern. Each chunk for the first goal would be a location-independent pattern recognizer.

The chunk improves the performance of the system by removing the need to fully process the goal; the chunk takes care of it. The process of chunking proceeds bottom up in the goal hierarchy. Once chunks are created for all of a goal's subgoals in a specific situation, it is possible to create a chunk for the goal. This process proceeds up in the hierarchy until there is a chunk for the top-level goal for every situation that it could face.

Mechanisms for goal processing and chunking have been built into a new production-system architecture that fits within a set of constraints developed for the architecture of cognition. This architecture has been applied to a number of different reaction-time tasks (though not all of these results are presented here). It is capable of producing power-law practice curves.

As currently formulated, the chunking theory stakes out a position that is intermediary between three previously disparate mechanisms: classical chunking, memo functions, and production composition. These four ideas are different manifestations of a single underlying store-versus-compute idea.

And finally, we have outlined a research path by which the chunking theory, when integrated with a problem-solving system, can potentially be expanded to cover aspects of learning — such as method acquisition — outside of the domain of pure practice.

References

- Anderson, J. R. *Language, Memory, and Thought*. Hillsdale, N.J.: Lawrence Erlbaum Associates, 1976.
- Anderson, J. R. Private communication. 1980.
- Anderson, J. R. Private communication. 1982.
- Anderson, J. R. Acquisition of cognitive skill. *Psychological Review*, 1982, 89, 369-406.
- Anderson, J. R. *The Architecture of Cognition*. Harvard Press, 1983.
- Bower, G. H. Perceptual groups as coding units in immediate memory. *Psychonomic Science*, 1972, 27, 217-219.
- Bower, G. H., & Springston, F. Pauses as recoding points in letter series. *Journal of Experimental Psychology*, 1970, 83, 421-430.
- Bower, G. H. & Winzenz, D. Group structure, coding, and memory for digit series. *Journal of Experimental Psychology Monograph*, 1969, 80, 1-17. (May, Pt. 2).
- Card, S. K., English, W. K. & Burr, B. Evaluation of mouse, rate controlled isometric joystick, step keys, and text keys for text selection on a CRT. *Ergonomics*, 1978, 21, 601-613.
- Chase, W. G., & Ericsson, K. A. Skilled memory. In J. R. Anderson (Ed.), *Cognitive Skills and Their Acquisition*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1981.
- Chase, W. G. & Simon, H. A. Perception in chess. *Cognitive Psychology*, 1973, 4, 55-81.
- Crossman, E. R. F. W. A theory of the acquisition of speed-skill. *Ergonomics*, 1959, 2, 153-166.
- DeGroot, A. D. *Thought and Choice in Chess*. The Hague: Mouton, 1965.
- Duncan, J. Response selection rules in spatial choice reaction tasks. In S. Dornic (Ed.), *Attention and Performance VI*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1977.
- Ernst, G. W. & Newell, A. *GPS: A Case Study in Generality and Problem Solving*. New York: Academic Press, 1969. (ACM Monograph).

- Fitts, P. M., & Seeger, C. M. S-R compatibility: Spatial characteristics of stimulus and response codes. *Journal of Experimental Psychology*, 1953, 46, 199-210.
- Forgy, C. L. *OPS4 User's manual* (Tech. Rep. CMU-CS-79-132). Carnegie-Mellon University Department of Computer Science, July 1979.
- Johnson, N. F. Organization and the concept of a memory code. In Melton, A. W. & Martin, E. (Eds.), *Coding Processes in Human Memory*. Washington, D.C.: Winston, 1972.
- Kolers, P. A. Memorial consequences of automatized encoding. *Journal of Experimental Psychology: Human learning and memory*, 1975, 1(6), 689-701.
- Laird, J. E., and Newell, A. A universal weak method: Summary of results. In *Proceedings of the Eighth IJCAI*, 1983.
- Lewis, C. H. *Production system models of practice effects*. Doctoral dissertation, University of Michigan, 1978.
- McDermott, J. R1: a rule-based configurator of computer systems. (Tech. Rep. 80-119). Carnegie-Mellon University Computer Science Department, 1980.
- Michie, D. "Memo" functions and machine learning. *Nature*, 1968, 218, 19-22.
- Miller, G. A. The magic number seven plus or minus two: Some limits on our capacity for processing information. *Psychological Review*, 1956, 63, 81-97.
- Moran, T. P. *Compiling cognitive skill* (AIP Memo 150). Xerox PARC, 1980.
- Morin, R. E., & Forrin, B. Mixing two types of S-R associations in a choice reaction time task. *Journal of Experimental Psychology*, 1962, 64, 137-141.
- Neisser, U., Novick, R., Lazar, R. Searching for ten targets simultaneously. *Perceptual and Motor Skills*, 1963, 17, 427-432.
- Neves, D. M. & Anderson, J. R. Knowledge compilation: Mechanisms for the automatization of cognitive skills. In Anderson, J. R. (Ed.), *Cognitive Skills and their Acquisition*. Hillsdale, NJ: Erlbaum, 1981.
- Newell, A. Heuristic programming: Ill-structured problems. In Aronofsky, J. (Ed.), *Progress in Operations Research*, III. New York: Wiley, 1969.
- Newell, A. Production systems: Models of control structures. In Chase, W. G. (Ed.), *Visual Information Processing*. New York: Academic Press, 1973.
- Newell, A. Harpy, production systems and human cognition. In Cole, R. (Ed.), *Perception and Production of Fluent Speech*. Hillsdale, NJ: Erlbaum, 1980. (Also available as CMU CSD Technical Report, Sep 1978).
- Newell, A. Reasoning, problem solving and decision processes: The problem space as a fundamental category. In R. Nickerson (Ed.), *Attention and Performance VIII*. Hillsdale, N.J.: Erlbaum, 1980. (Also available as CMU CSD Technical Report, Aug 79).
- Newell, A. & Rosenbloom, P. S. Mechanisms of skill acquisition and the law of practice. In J. R. Anderson (Ed.), *Cognitive Skills and Their Acquisition*. Hillsdale, NJ: Erlbaum, 1981.
- Newell, A. & Simon, H. A. *Human Problem Solving*. Englewood Cliffs: Prentice-Hall, 1972.
- Nilsson, N. J. *Problem-Solving Methods in Artificial Intelligence*. New York: McGraw-Hill, 1971.
- Rosenbloom, P. S. *The Chunking of Goal Hierarchies: A Model of Practice and Stimulus-Response Compatibility*. Doctoral dissertation, Carnegie-Mellon University, 1983.
- Rosenbloom, P. S., & Newell, A. Learning by chunking: Summary of a task and a model. In *Proceedings of AAAI-82, National Conference on Artificial Intelligence*. American Association for Artificial Intelligence, 1982.
- Rosenbloom, P. S. & Newell, A. Learning by chunking: A production-system model of practice. In D. Klahr, P. Langley, & R. Neches (Ed.), *Self-Modifying Production System Models of Learning and Development*. MIT Press, In press. (Also available as Carnegie-Mellon University Computer Science Tech. Rep. # 82-135).
- Samuel, A. L. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 1959, 3, 210-229.
- Sauers, R., and Farrell, R. Grapes user's manual. 1982.
- Seibel, R. Discrimination reaction time for a 1,023-alternative task. *Journal of Experimental Psychology*, 1963, 66(3), 215-226.
- Shortliffe, E. H. *Computer-Based Medical Consultations: MYCIN*. New York: American Elsevier, 1976.
- Snoddy, G. S. Learning and stability. *Journal of Applied Psychology*, 1926, 10, 1-36.
- Winston, P. H. Learning structural descriptions from examples. In Winston, P. H. (Ed.), *The Psychology of Computer Vision*. New York: McGraw Hill, 1975.

Learning Physical Domains: Towards a Theoretical Framework

Kenneth D. Forbus, Dedre Gentner
Bolt, Beranek and Newman, Inc.
10 Moulton Street
Cambridge, Mass. 02139, USA

Abstract

We describe a theoretical framework being developed in an attempt to account for human learning of physical domains. We are using Qualitative Process Theory to model people's physical knowledge and the Structure-Mapping Theory to characterize the computations that move a learner from one representation to another. Here we outline the component theories and propose a learning sequence for physical domains.

1. Introduction

People use and extend their knowledge of the physical world constantly. Understanding how this fluency is achieved would be an important milestone in understanding human learning and intelligence. Our primary purpose is to construct a computational account of human learning.

Below we describe the theoretical framework we are developing in an attempt to account for human learning of physical domains. Any such framework must describe the form of the domain models and the kinds of computations that move the learner from one representation to another. Thus we first briefly summarize Qualitative Process Theory [Forbus, 1981, 1982] and the Structure Mapping Theory [Gentner, 1980, 1983], the component theories that underlie our framework. We then describe a learning sequence, composed of four stages of physical models, and our conjectures about how a learner constructs one type of theory from another.

2. Qualitative Process Theory

Our first requirement is a language in which to describe people's commonsense knowledge about physical situations. People know about a great many kinds of physical changes: things move, collide, bend, break, heat up, cool down, flow and boil. Intuitively we think of these as *processes*. Qualitative Process theory attempts to formalize this notion. An important property for our purposes is that both correct and incorrect physical models can be expressed.

We view a physical situation as composed of objects and relationships among them. The continuous parameters of an object, such as temperature and pressure, are represented by *quantities*. In QP theory the value of a number is defined in terms of its *Quantity Space* - a collection of inequalities which hold between it and other quantities. The Quantity Space is a useful qualitative representation because processes typically start and stop when inequalities between parameters change.

Processes cause changes by *influencing* quantities. There are two kinds of influence which can occur. A quantity can be *directly influenced* by a process or processes, or indirectly influenced because it is a function of some other quantity that is influenced. We will specify functional dependencies between quantities by

$$Q1 \propto Q2$$

(read "Q1 is qualitatively proportional to Q2"), meaning there exists a

function which determines Q1 and is increasing monotonic in its dependence on Q2. $\propto$ expresses partial information, since the implicitly specified function may or may not depend upon other quantities. Figuring out the sign of a quantity's derivative from its influences is called *resolving* its influences.

A process is a thing that acts through time to cause changes. Processes are specified by five parts:

Individuals: Descriptions of the objects which participate in the process.

Quantity Conditions: Inequality statements and status assignments to other conditionalized descriptions which must be true for the process to be active.

Preconditions: Statements other than Quantity Conditions that must be true for the process to be active.

Relations: The relationships between the individuals which hold when the process is active.

Influences: Descriptions of what quantities are directly influenced by the process.

A process acts between any collection of objects which match the individual specifications, whenever both the Preconditions and Quantity Conditions are true. Preconditions, as distinct from Quantity Conditions, express changes that cannot be predicted solely within QP theory - physics alone will not tell us that someone will walk by and turn on a stove, for instance, although it can predict that a result of this action will be that the water sitting in the kettle on top of it will soon boil. The Relations specify the functional dependencies induced by the process as well as any new entities which it introduces by virtue of being active, such as a gas generated by boiling.

The collection of processes acting at any time characterizes "what is happening" then in that situation. A reasoner's theory of dynamics for a domain is characterized in terms of a *Process Vocabulary* that describes the kinds of processes the reasoner believes can occur. The actual changes that occur as a result of processes acting are represented as a *history* [Hayes, 1979].

QP theory sanctions several basic deductions. These include determining what processes might occur, what processes are occurring, and determining how the set of active processes can change over time. These basic deductions can be combined to perform more complex reasoning tasks. Two examples of more complex deductions are qualitative simulation [Forbus, 1982] and measurement interpretation [Forbus, 1983].

3. Comparisons and Structure Mapping

So far we have considered how a Naïve Physics might be represented. We now turn to the question of how such domain models might be learned. A major process in experiential learning is comparing the current situation with stored descriptions. Consider a person who has just moved to a cold climate learning to operate his furnace. By the end of the winter he may learn that he cannot cause the house to get warm any sooner by setting the thermostat higher than the desired temperature. There are at least three different ways

he could reach this conclusion. First, he may compare the current situation with past furnace experiences and notice a regularity. Second, he may match the current situation with a known abstraction, such as a position-action controller (as opposed to a proportional-action controller), to deduce the answer. Third, he may use an analogy, comparing the current situation with a description from another domain, such as fluid flow, to suggest governing principles. Each of these ways of learning relies on some form of comparison, either with a stored record of literally similar events, with a stored abstraction, or with a stored description that can function as an analogy.

The *Structure Mapping Theory* concerns such comparisons (Gentner, 1980, 1981, 1983; Gentner & Gentner, 1983). The theory describes the rules that are used to import a descriptive structure from one domain (the *base* domain) to another (the *target* domain). Literal Similarity, Analogy, and Abstraction Mappings (applications of general laws) are viewed as different kinds of mappings between descriptions. The types of comparisons are defined syntactically, and depend only on the form of the knowledge representation, not its content. In particular:

- (1) A *literal similarity* statement is a comparison in which a large number of predicates is mapped from base to target. There are many more shared predicates than nonshared predicates. The mapped predicates include both object-attributes and relational predicates.¹
- (2) An *analogy* is a comparison in which relational predicates, but few or no object attributes, can be mapped from base to target. The particular relations mapped are determined by systematicity, as defined by the existence of higher-order constraining relations.²
- (3) An *abstraction mapping* is a comparison in which the base domain is an abstract relational structure. Predicates from the abstract base domain are mapped into the target domain. Since there are no object attributes in the base, there are no one-place predicates to be left behind. Applying a rule to a situation is an example of the abstraction mapping. Sometimes we will also refer to the relational structure so mapped as an abstraction.

The *Analogical Shift Hypothesis* concerns how these comparisons are used in learning and reasoning about a domain. In the earliest stages most of the comparisons are literal similarity matches. As the domain becomes familiar, analogies begin to occur. Successful analogies may yield abstractions. Analogy serves as a means of changing mental models, and therefore plays an important role in the middle stages of learning. In the later stages of learning, abstraction mappings and literal similarities predominate.

The role of a comparison in learning depends on at least two things: (1) *accessibility*, the likelihood that the match will be noticed; and (2) *usefulness*, what can be deduced from the match, if it is

accessed. Accessibility, in turn, depends at least on (i) the *familiarity* of the base description and (ii) the overall *similarity* between the base description and the current target. The comparisons discussed above behave very differently with respect to these two criteria.

Literal similarity matches are quite accessible, but of limited usefulness. Psychological evidence indicates that people automatically perform such comparisons to combine perceptually similar experiences into composite prototypes. However, such composites are of limited use in revealing the principles that govern the phenomenon. In general, literal similarity matches are highly accessible, since they can be indexed by object descriptions, by relational structures, or by both. But they are not very useful in deriving causal principles, precisely because there is too much overlap to know what is crucial. Abstractions are extremely useful, but not necessarily accessible: novice learners may not know the appropriate abstraction, or it may be so unfamiliar that its accessibility will be low. Thus abstraction mappings, while ultimately important, are unlikely to play a major role in the early stages of learning.

Analogies lie between the highly accessible literal similarity matches and the highly useful abstraction matches. Potential analogies are less accessible than literal similarity matches, since they require accessing the learner's data base via relational matches; object matches are of no use. However, once found, an analogy should be more useful than a literal similarity match in deriving the key principles, since the shared data structure is sparse enough to permit analysis. Moreover, if we assume the systematicity principle, then the set of overlapping predicates is likely to include higher-order relations, such as causality. Thus analogy can function to reveal principles in a domain that previously lacked the appropriate abstractions (Gick & Holyoak, 1981). Winston's (1981) program, which derives if-then rules by abstracting the predicates common to two analogs, is a case in point.

4. Stages of Understanding

We suspect there are four kinds of mental models that will be generated in the process of understanding. The sequence of models we propose is developmental in that the theories of each stage are generated both by the phenomena being understood and the theories of the stage before it. We do not think every person goes through each stage for every domain, nor that a person is at the same stage in every domain at the same time.

4.1 Stage 1 - ProtoHistories

Suppose we are observing some new physical phenomena. All we can do is observe and remember what is happening. We conjecture that the simplest physical models of a domain are *protohistories* -- prototype histories which serve as summaries of experience.¹ Like object prototypes, protohistories are the "most typical instances" of phenomena (see Rosch, 1973, 1975, 1976; Smith & Medin, 1981). The terms in these descriptions are observables, and their deductive import can be roughly expressed as "If I see X, then Y

1. Object attributes are predicates which take one object as an argument, such as RED(x). Relations are predicates which take two or more arguments, such as COLLIDE(x, y).

2. We define the *order* of a proposition as follows: Constants have order zero, as do functions on them. The order of a proposition is one plus the maximum of the orders of its arguments. Thus COLLIDE(x, y) would be first order if x and y are domain objects, and CAUSE(COLLIDE(x, y), BREAK(x)) would be order two. Examples of higher-order relations are CAUSE and IMPLIES.

1. Some of diSessa's *Phenomenological Primitives* appear to be representable as protohistories [diSessa, 1983].

will(has) happen(ed)."

Consider a balance beam or see-saw. If we place a weight on each side of the fulcrum, it will either tilt counter-clockwise, clockwise, or not at all. Most people have had enough experiences with see-saws to have formed protohistories concerning their behavior. By our conjecture, a protohistory is automatically available whenever we encounter a see-saw. From it, we can often predict which way the particular see-saw will move. For example, we may have a protohistory that describes what happens if a small person gets on opposite a large person.

However, the predictive power of protohistories is quite limited. There is no guarantee that the features matched actually correspond to relevant factors. For example, we will be fooled when a large person sits close to the fulcrum if we have formed our see-saw protohistory from watching people sitting at equal distances. Massive overlap in features is needed for reliable use, which means protohistories will yield conclusions in fewer situations than a true theory would. Furthermore, there is no certain way to decide between conflicting results if more than one protohistory matches a situation.¹

4.1.1 Learning Protohistories

The process of constructing protohistories requires dividing up experience into classes according to literal similarity and abstracting a summary for each class. There has been little direct research on this process. However, investigations into the process of constructing object prototypes provides some hints. First, people seem to be able implicitly to compute a kind of component match. Second, this intersection is not merely a simple feature intersection; rather, it appears that configurations among features are important in the prototype. Third, once this prototype is computed, it has powerful effects on subsequent processing of experience. Once people abstract a prototype from a set of patterns, they may be more confident of having seen the prototype (which was never presented) than they are of having seen the actually presented patterns (Posner & Mitchell, 1967). Finally, people may not be aware of forming prototypes, except as a general sense of increased familiarity with a category.

To summarize, if protohistories behave like object prototypes, then we should find that they are computed implicitly; that they are sensitive to configurations among events; that, once computed, they show the recognition strength and other psychological privileges of prototypes.

4.2 Stage 2 - The Causal Corpus

While protohistories summarize the phenomena, they are not a theory of it. Jumping directly to a detailed theory can be quite difficult. For example, the space of possible equations connecting all observable parameters in a typical situation can be quite large. Instead, we conjecture that weaker theories are formed first, theories that characterize which parts of the situation are relevant to desired

conclusions. In particular, we conjecture that a collection of CAUSE statements, the *Causal Corpus*, is computed from prototype objects and protohistories.

We view CAUSE as an approximation concept, in particular a weak form of ontological commitment. Saying

CAUSE(A, B)

means that there is a mechanism, specified by some theory T, such that

$A \wedge T \Rightarrow B$

Processes provide the notion of mechanism for physical domains. The statement

CAUSE(Triangle(F), Has-Three-Sides(F))

is not a legitimate use of CAUSE by this definition, since the required axioms of geometry do not specify a mechanism.

The further distillation of experience from protohistories to the Causal Corpus serves two purposes (aside from its potential as a means of data reduction). First, it provides a collection of heuristics that can be used directly to draw inferences. While the learner doesn't have firm grounds to consider the CAUSE statements complete or correct, often they may suffice for the desired class of inferences. Second, the collection of CAUSE relations can be used to guide the search for a deeper theory of the domain. The CAUSE statements suggest connections between various aspects of the domain which a deeper theory must either explain or explain away.

Returning to our see-saw example, we might attempt to apply our Causal Corpus to a balance beam built out of blocks. Suppose we call the two blocks on it A and B. Our Causal Corpus might be:

Cause(Bigger(A, B), Tilt-Towards(A))
Cause(Farther(A, B), Tilt-Towards(A))

These statements can be interpreted as rules in several ways: if we see that block A is bigger than block B we can predict tilt, and if we see tilt we may hypothesize that one block is farther out than another. These statements are more broadly applicable than protohistories since they refer to fewer properties. Unlike protohistories, the Causal Corpus can be debugged to some degree.

However, the approximate nature of the CAUSE relation limits the ability to discriminate between conflicting predictions. With the Causal Corpus above, for instance, if block A is bigger and block B is farther out we will have two predictions. As we shall discuss, in order to make more precise inferences the learner must eventually uncover the mechanisms whose behavior the Causal Corpus describes.

4.2.1 Learning the Causal Corpus

We suspect there are three techniques for computing and debugging a Causal Corpus. The first technique is to hypothesize causality from co-occurrence:

If you always see A before B,
then hypothesize CAUSE(A, B)

and

If A is true whenever B is true,

1. There are of course heuristic criteria, such as taking the one that matches "best" or the one which has worked most often. The problem with such heuristics is that little is learned from mistakes.

then hypothesize CAUSE(A, B)

These rules make certain assumptions on the form of memory, namely that some number of circumstances can be remembered, and in sufficient detail so that A is either explicitly stored or computable from what is stored. Protohistories should serve as a means of data reduction, reducing the complexity of constructing a Causal Corpus with these rules.

We also conjecture a class of rules which are used to debug a Causal Corpus in the face of new information (c.f. [Sussman, 1976]). Each rule corresponds to a hypothesis about a bug in the structure of the Causal Corpus, such as a missing precondition. We have not addressed the issue of judging a Causal Corpus for consistency and completeness; such criteria will play a major role in controlling the debugging rules and in the mixture of generation and debugging that occurs.

Analogy provides the third technique for extending a Causal Corpus. The CAUSE relations from one domain can be mapped into another, since CAUSE qualifies as a higher-order constraining relation (see [Winston, 1981]).

4.3 Stage 3 - Naive Physics

The Naive Physics models replace CAUSE statements with theories about the specific mechanisms of change. The ontology is extended by adding processes to explain observed changes. The ontology also includes properties and objects that are not directly observable (for example, heat and heat flow) and the new relationships (such as fluid path and heat path) required to reason about them.

An important advantage of these models is the ability to reason by exclusion. Unlike the previous stages, predictions that fail still yield information about the situation. For instance, if fluid is seen flowing into a container and the level is not rising, then it is reasonable to assume that fluid is flowing out of it through some unknown path.

Returning to our balance beam, we might use a process SWING to describe rotation around a contact point. The Precondition would be that the object is free to rotate, and the Quantity Condition would be a non-zero angular velocity. SWING directly influences the angular position of the beam. Further, its influence is qualitatively proportional to the angular velocity. Thus a prediction concerning tilt becomes a prediction about which instance, if any, of the SWING process will occur. Net force becomes the crucial parameter; to deduce which way something swings requires comparing the forces exerted on each side of the beam. The factors illuminated in the Causal Corpus of "Bigger" and "Farther" become the quantities "Mass" and "Distance". Examining the description of SWING and the CAUSE statements suggests we write force as a function of the two parameters, and in particular:

$$\begin{aligned}\text{Force(A)} &\propto_Q \text{Distance(A, pivot)} \\ \text{Force(A)} &\propto_Q \text{Mass(A, pivot)}\end{aligned}$$

which brings us one step closer to a model that can always determine which way something will tilt.

4.3.1 Learning Naive Physics

The major problem in learning a naive physics is constructing a vocabulary of processes which consistently describes experience. The learner must strip away the irrelevant predicates that are part of his protohistories and Causal Corpus; and often, he must construct new objects and properties that are not observable.

The Causal Corpus provides a search space for potential Process Vocabularies. Each statement in the CC must be elaborated into a consequence of a Process Vocabulary. There appear to be only a small number of distinct ways to perform the elaboration depending on the particular form of the arguments. For example, the statement

The decrease in amount of Q
causes the level to fall.

indicates that some active process (or Individual View) in the situation contains the statement

$$\text{Level(Q)} \propto_Q \text{Amount-of(Q)}$$

in its Relations.

While hypothesizing a Process Vocabulary on the basis of the Causal Corpus should be much simpler than working directly from protohistories or direct observation, it still appears difficult. We conjecture several constraints that make the problem more tractable. First, we think people are *conservative* in the introduction of unobserved properties. For example, we have interviewed subjects whose model of a domain appears to be organized around one parameter - a "generalized strength" attribute - which is used in all Quantity Conditions and comparisons. In reasoning about fluids, for instance, pressure, flow rate, and velocity appear to be used as different names for the same thing. In electricity, voltage, current, power, potential, and velocity of electrons are used interchangeably by some people. The advantage of this generation strategy is of course that simpler models will be explored first, with further distinctions made only when necessary. Second, some physical laws can be viewed as constraints on what Process Vocabularies are possible. Conservation of Energy, for example, demands that if a process directly influences a quantity that represents some kind of energy it must also directly influence some other quantity which represents energy in the opposite direction.

Once again, Analogy can provide a constructive mechanism. It can be used to import candidate processes from previously understood domains, such as understanding heat flow in terms of fluid flow.

4.4 Stage 4 - Expert Models

The models generated so far still contain fundamental ambiguities.¹ The final stage of learning consists of discovering ways to resolve those ambiguities. We suspect several kinds of knowledge are involved. (Obviously, the potential complexity of the models is open-ended.) These kinds of knowledge include equations to describe relationships between parameters and "rules of thumb"

1. [de Kleer, 1979] discusses the nature of ambiguity in qualitative descriptions.

which specify useful default resolutions for the ambiguities inherent in qualitative descriptions. Examples of these ambiguities include knowing circumstances in which certain processes can be ignored (such as evaporation when pouring water from one glass to another) and what their net effect is (such as Black's law for the temperature of mixtures).

Returning to the balance beam example, we know that the force of a block on the beam is qualitatively proportional to the mass of the block and to the distance from the fulcrum. If we also know that the force is the product of distance and weight, then providing numerical values for these quantities will allow an unambiguous prediction about tilt.

4.4.1 Learning Expert Models

One part of refining our commonsense models is to elaborate α_Q statements into constraint equations. [Langley, 1979] describes techniques that should be useful.

Developing rules of thumb requires discovering which outcomes raised by qualitative reasoning are unlikely and which potential interactions can be ignored. We suspect the possible behaviors raised by Naive Physics are compared against known protohistories. Hypothesized outcomes that have no corresponding protohistory are considered unlikely, while those corresponding to a very familiar protohistory are considered very likely.

5. Summary

We have described our progress in weaving together the Structure Mapping theory and Qualitative Process theory into a framework that aims to account for learning in physical domains. We are still at the stage of refining the questions, rather than providing answers.

6. Acknowledgements

This project is carried out at Bolt, Beranek, and Newman, Inc., under contract to Schlumberger-Doll. We thank Allan Collins, Albert Stevens, and Bob Berwick for their insightful comments.

7. References

- de Kleer, J. Causal and Teleological Reasoning in Circuit Recognition TR-529, MIT AI Lab, Cambridge, Massachusetts, September 1979
- diSessa, A. Phenomenology and the Evolution of Intuition. in Mental Models. D. Gentner and A. Stevens, LEA, Inc, April 1983
- Forbus, K. Qualitative Reasoning about Physical Processes. Proceedings of IJCAI-7, 1981
- Forbus, K. Qualitative Process Theory. MIT AI Lab Memo No. 664, 1982
- Gentner, D. The structure of analogical models in science (BBN Report No. 4451). Cambridge, Mass.: Bolt Beranek and Newman Inc., 1980.
- Gentner, D. Structure-mapping: A theoretical framework for analogy. Cognitive Science, 1983, 7, No. 2.

Gick, M. L., & Holyoak, K. J. Analogical problem solving. Cognitive Psychology, 1980, 12, 306-355.

Hayes, P. J. The Naive Physics Manifesto in Expert Systems in the Micro-Electronic Age, edited by D. Michie, Edinburgh University press, May 1979

Hayes, P. J. Naive Physics 1 - Ontology for Liquids Memo, Centre pour les etudes Semantiques et Cognitives, Geneva, 1979

Langley, P. Rediscovering Physics with BACON.3 Proceedings of IJCAI-6, 1979

Posner, M. I., & Gentner, D. R. Chronometric analysis of classification. τ BP_{Psychological Review} τ E, 1967, τ B1 τ E, 392-409.

Sussman, G. A Computational Model of Skill Acquisition, Springer-Verlag, New York, 1976

Winston, P. Learning new principles from precedents and exercises. MIT AI Lab Memo No. 632, May 1981

Knowledge Compilation: The General Learning Mechanism

John R. Anderson
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

The ACT learning mechanisms of knowledge compilation are discussed. Knowledge compilation consists of the mechanisms of composition and proceduralization. Composition operates by collapsing multiple productions into a single production that has the effect of the set. Proceduralization operates by building into productions information that previously had to be retrieved from long-term memory. It is shown how these two mechanisms can simulate the initial stages of skill acquisition in the domain of learning how to program in LISP. It is also shown that these mechanisms can reproduce the effects of the generalization and discrimination learning mechanisms. Generalizations and discriminations emerge as the consequence of compiling the processes of analogy formation and error correction.

Key terms: learning, knowledge compilation, composition, proceduralization, skill acquisition, simulation of LISP programming, analogy, generalization, discrimination, cognitive modelling.

Introduction

For a number of years (Anderson, 1982, 1983a; Anderson, Sauer, & Farrell, 1982; Neves & Anderson, 1981) we have been working on a general set of learning principles that come under the rubric of *knowledge compilation*. These principles are concerned with how a new skill is acquired such as generating a proof in geometry (Anderson, 1983b). The general framework is one in which a learner is viewed as beginning with declarative information relevant to the execution of a skill. For instance, in the case of geometry, the student might learn about the properties of two column proofs and various theorems and postulates. This information is stored in declarative form--i.e., as facts about the domain. For this knowledge to be used, general *interpretive* procedures must apply to use these facts. Two common types of interpretive procedures are general problem-solving procedures and general analogy procedures. Knowledge compilation operates on the traces of such procedures. It will create new procedures which are specific to that domain and which are more efficient.

This paper will discuss knowledge compilation and give an example of its use to simulate learning of initial programming skills. The rest of the paper will be devoted to discussing a recent discovery we have made about the knowledge compilation mechanism--that it can be used to produce learning effects that had been thought to be the result of independent learning mechanisms of discrimination and generalization (Anderson, Kline, and Beasley, 1979; Hayes-Roth and McDermott, 1976; Langley, 1983; Michalski and Stepp, 1983; Mitchell, 1978; Vere, 1975). Further I will argue that there is evidence in the human case that knowledge compilation is the process that underlies generalization and discrimination.

Introduction to Knowledge Compilation

The knowledge compilation mechanisms are defined with respect to a production system like ACT (Anderson, 1983a) which has a separate long-term declarative memory to represent facts and a production memory to represent procedures. The knowledge compilation mechanisms operate on the traces of production application to create new productions. Before getting into the details of a full example or the details of the implementation, it would be worthwhile to have a brief overview.

The knowledge compilation processes in ACT can be divided into two subprocesses. One, called *composition*, takes a sequence of productions that follow each other in solving a particular problem and collapses them into a single production that has the effect of the sequence. The idea of composition was first developed by Lewis (1978). Composition speeds up the process considerably by creating new productions that embody the sequence of steps used in a particular problem domain. The second process, *proceduralization*, builds versions of the productions that no longer require the domain-specific declarative information to be retrieved into working memory so the information can be matched by the general interpretive productions. Rather, the essential products of these retrieval operations are built into the new productions.

The basic processes of compilation can be illustrated with respect to dialing telephone numbers. It has been noted (Anderson, 1976) that with a frequently dialed telephone number, one develops a special procedure for dialing it; sometimes declarative access to the number is lost, and the only access one has to the number is through a procedure for dialing it.

Consider the following two productions that might serve to dial a telephone number:

P1

IF the goal is to dial = telephone-number
and = digit is the first digit of = telephone-number
THEN dial = digit.

P2

IF the goal is to dial = telephone-number
and = digit1 has just been dialed
and = digit2 is after = digit1 in = telephone-number
THEN dial = digit2.

Composition creates "macroproductions" that do the operation of a pair of productions that occurred in sequence. Applied to the sequence of P1 followed by P2, composition would create:

P1&P2

```
IF the goal is to dial = telephone-number
  and = digit1 is the first digit of = telephone-number
  and = digit2 is after = digit1 in = telephone-number
THEN dial = digit1 and then = digit2.
```

Compositions like this will reduce the number of production applications to perform the task.

A composed production like P1&P2 still requires that the information (in this case, the telephone number) be retrieved from long-term memory, held in working memory, and matched to the second and third clauses in P1&P2. Proceduralization eliminates clauses in the condition of a production that require information to be retrieved from long-term memory and held in working memory. In P1&P2, the second and third condition clauses would be eliminated. The variables that would have been bound in matching these clauses are replaced by the values that they are bound to in the special case. If this production is applied in dialing Mary's telephone number, which is 432-2815, the variables in P1&P2 would be bound as follows:

```
= telephone-number = Mary's number
  = digit1 = 4
  = digit2 = 3
```

Substituting these values for the variables and eliminating the second and third condition clauses transforms the production into:

P1&P2*

```
IF the goal is to dial Mary's number
THEN dial 4 and then 3.
```

By further composition and proceduralization, a production can be built that dials the full number:

P*

```
IF the goal is to dial Mary's number
THEN dial 4-3-2-8-1-5.
```

It should be emphasized that forming this production does not necessarily imply the loss of the declarative representation nor of the ability to use it interpretively. In the few reported cases of loss of declarative access to a telephone number, the declarative knowledge has ceased to be used and has simply been forgotten.

Elsewhere (Anderson, 1982, 1983a; Neves & Anderson, 1981) we have discussed the evidence for knowledge compilation from the literature of experimental psychology. The major issue to be considered here is how these mechanisms are implemented and used in relatively complex problem-solving domains. Our work on this topic has been done in the context of the GRAPES simulation of the ACT theory (Sauers and Farrell, 1982). GRAPES is a system devoted to simulating ACT in the context of novice LISP programming. It simulates the problem solving and programming of novices in writing LISP functions and how novices learn from their problem-solving episodes. I will describe here first a typical GRAPES simulation of a subject and then focus on the mechanisms of knowledge compilation underlying that simulation.

A Simulation of LISP Programming

One of our consistent observations about novices is that they are not able to read instructions, of even modest complexity, and then generate, without error, the behavior instructed. This is not surprising given the ACT theory. According to that theory, instructions are stored initially in a declarative form while behavior requires procedures which are represented as productions. Instructions cannot directly set up procedures to perform the skill. To get behavior, general interpretive productions must convert this knowledge into behavior. Many of the problems arise because of the indirection through these interpretive productions.

I will discuss one of the early learning episodes that we have simulated. If the reader is interested in the simulation of more complex programming behavior he is referred to Anderson, Farrell, and Sauers (1982). The problem-solving episode to be discussed, from one of our subjects BR, is typical of the difficulties people have in making the transition from instruction to experience. She had read the instruction on pages 33-37 of Winston and Horn (1981) on function definition and had turned to the first problem on page 37 to define the function FIRST. This function was to return the first element of a list. This was her first attempt at defining a function.

She extracted virtually nothing from the text instruction. What she did extract was a template for how to write a function definition:

```
(DEFUN <function name>
  (<parameter 1> <parameter 2>...<parameter n>)
  <process description>)
```

Winston and Horn assert "angle brackets delineate descriptions of things". She also studied three examples of function definitions to which she referred. The first converted fahrenheit to centigrade:

```
(DEFUN F-TO-C (TEMP)
  (QUOTIENT (DIFFERENCE TEMP 32) 1.8))
```

The second exchanged the first two members of a list:

```
(DEFUN EXCHANGE (PAIR)
  (LIST (CADR PAIR) (CAR PAIR)))
```

The third returned the percentage by which the second argument is larger than the first:

```
(DEFUN INCREASE (X Y)
  (QUOTIENT (TIMES 100.0
    (DIFFERENCE Y X))
    X))
```

The one other relevant thing BR knew was the function CAR and how to use it when interacting with the monitor in LISP. CAR returns the first element of the list which is its argument. She knew, for instance, that if she types (CAR '(A B C)), the monitor would return the answer A. Thus, this problem is really an exercise in using the syntax of function definition rather than defining a novel function.

BR's method of solving this problem relied heavily on trying to use the structure of the template and the examples to guide her function writing. We refer to this process as *structural analogy*. We created a production system in GRAPES that would simulate this protocol. The only productions we required for this simulation were

ones to do structural analogy and ones that could use the LISP functions CAR and CDR at the top level. The first type of productions represent a general prior skill evoked in many contexts (for instance, in filling out income tax forms). The second type was acquired from work with earlier chapters in Winston and Horn.

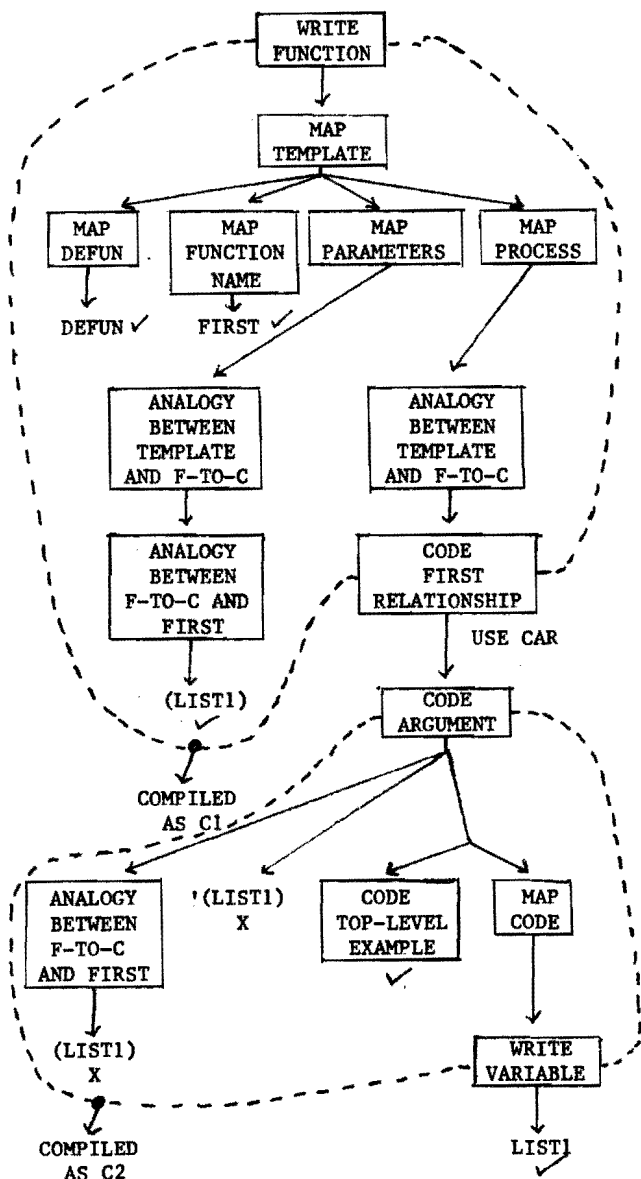


Figure 1.

Figure 1 illustrates the goal tree generated in simulating this example. Each box in Figure 1 represents a goal and each arrow emanating from the box represents a GRAPES production trying to achieve the goal. If the production has subgoals, it is connected to goal boxes below. The simulation starts with the goal of writing the function and chooses as its method to use the template for function definition as model. We refer to this as *mapping* the template. The first subgoals that GRAPES processed in Figure 1 were concerned with mapping DEFUN and <function name> in the template. Productions responding to these goals wrote out "(DEFUN FIRST" without difficulty.

Like our subject, GRAPES was not able to directly write out the parameter part of the function template because GRAPES did not know what a "parameter" was. In cases like this, GRAPES' analogy productions will resort to a concrete example. The concrete example retrieved by GRAPES is the definition of F-TO-C given earlier. The subject reviewed two examples in Winston and Horn—F-TO-C and EXCHANGE. In both cases she noted the single argument which was the parameter. GRAPES solved the analogy: X is to F-TO-C as the parameter list (i.e., (<parameter 1><parameter 2>...<parameter n>)) is to the abstract template, and retrieved (TEMP) as the value for X. Thus, it decided (TEMP) was serving the parameter role in F-TO-C. Then it solved the analogy X is to FIRST as (TEMP) is to F-TO-C and came up with the answer (LIST1) which it put into the function definition. That is, GRAPES decided (LIST1) served the comparable role in the function it was defining as (TEMP) was serving in F-TO-C.

Then GRAPES turned to the process description. Being unable to directly interpret what is meant by <process description>, it looked to its concrete example F-TO-C and saw that LISP code filled this slot which performed the function operations. In analogy, GRAPES set its goal to write code that would perform the operations required by FIRST. The subject at this point looked to a different example, INCREASE, for the same purpose of analogy. A production for using CAR at the top level applied next in GRAPES, but there is no production to specify how to write the argument to the function to CAR in this context. GRAPES and the subject know that CAR will operate on LIST1, but they do not know the syntax for specifying it. GRAPES again turns to its concrete example, F-TO-C, and solves the analogy (CAR ARG) is to (QUOTIENT X) and retrieves (DIFFERENCE TEMP 32) which is the first argument to QUOTIENT. It then solves the analogy problem of what it must do to LIST1 to make it like (DIFFERENCE TEMP 32) and decides it should embed LIST1 in parentheses. Similar to GRAPES, the subject looks at a function EXCHANGE which has the same argument structure as F-TO-C. We assume she makes the same erroneous analogy because she writes (LIST1). The first argument to QUOTIENT in F-TO-C is embedded in parentheses (as is the first argument to LIST in EXCHANGE) because a function, DIFFERENCE, must be called to calculate the argument for QUOTIENT whereas no function call is required for the argument to CAR in the FIRST example. The function definition at this point by both subject and GRAPES is:

```
(DEFUN FIRST (LIST1)
  (CAR (LIST1)))
```

There are two things to note at this point. First, the subject has read, in the text, information that would have enabled her to know not to embed LIST1 in parentheses, but this has no impact on her behavior. Second, on previous occasions she had correctly specified variable arguments when evaluating functions at the top-level. Eventually, the tutor used this second fact, that the subject could do it correctly at the top level, to guide her to a correct solution. Both of these observations illustrate the relative isolation of knowledge. That is, knowledge studied or used in one context is not available in another context.

When the subject tries her function definition, an error message is generated—"LIST1: undefined function object". GRAPES received the same error message when it tried out the same function definition. The error occurred because LISP treats the first thing inside a parenthesized expression to be evaluated as a function and there is no

function corresponding to LIST1. GRAPES associated this error with the failure to correctly specify the argument to CAR. On previous occasions BR had encountered the same error at the top level typing in commands like (CAR (A)) where the argument to CAR is to be taken literally rather than evaluated. Always in the past she had repaired these errors by *quoting* the argument. This is done by preceding the argument with a single quote, i.e. (CAR '(A)). We assume that both the subject and GRAPES have compiled from previous experience a rule that the way to repress this error is by using quote, which stops LISP from evaluating. After applying this rule the function becomes:

```
(DEFUN FIRST (LIST1)
  (CAR '(LIST1)))
```

When this new function is tried on an example, LISP returns the CAR of '(LIST1) which is LIST1--rather than the first element of the value of LIST1. It is at this point that the tutor intervenes and reminds the subject of how she would solve the problem at the top level. In this context the subject would have used (CAR LIST1) rather than (CAR (LIST1)) or (CAR '(LIST1)). We simulated this intervention in GRAPES by refocusing it on the code-argument goal in Figure 1, and putting (CAR LIST1) as a top-level example in working memory. Then GRAPES, as the subject, maps this code to its current function definition and comes up with the correct code:

```
(DEFUN FIRST (LIST1)
  (CAR LIST1))
```

Knowledge Compilation

After finally solving this problem, The knowledge compilation mechanisms formed two productions which aided its solution of the second problem. These productions summarized much of the problem solving that took place.

```
C1
IF the goal is to write a function of one variable
THEN write (DEFUN function (variable)
  and set as a subgoal to code the relation calculated by
  this function
  and then write ).
```

```
C2
IF the goal is to code an argument
  and that argument corresponds to a variable of the function
THEN write the variable name.
```

We have encircled in Figure 1 the portions of the goal tree that are summarized by each of these productions. The first production captures the top level syntax of a function while the second summarized the search involved in finding out how to specify an argument to a function. With these productions, GRAPES was able to write a second function much easier, as was the subject. This function, called SECOND, was to return the second element of a list.

Conclusions

There are a number of conclusions that we draw from this example. The first is the importance of structural analogy to bridging the gap between current knowledge and the needed behavior. We see two sources for the structure from which the analogy is being made.

One is templates provided in the text and worked-out problems. The other is structures that the subject can generate--for instance, the subject generated (CAR LIST1) as top-level solution and then used this in her function definition.

A second conclusion concerns the hierarchical structure of the problem-solving episode as illustrated in Figure 1. Note also that the goals in this tree are expanded depth-first and left-to-right. Jeffries, Turner, Atwood and Polson (1981) also note this hierarchical structure in the programming behavior of experts--although their subjects use breadth-first expansion in correspondence with the edicts of structured programming.

The third conclusion is the importance of knowledge compilation in extracting new production rules from an example problem. These rules streamline the solution of later problems. As the protocol shows, the learning can be on the basis of a single example. It needs to be stressed that the lessons of this example "stuck" which is to say, BR did not have, on later days, the same difficulty with the basic syntax of function definition nor argument specification. It should also be stressed that compilation depends critically on the structure of goal trees. That is, the structure of the goal tree identifies what parts of the problem-solving episode belong together and can be collapsed into a single rule.

In these three features--structural analogy, hierarchical goal trees, and knowledge compilation--we have one complete solution to the issue of how the subject is able to make the transition to a new cognitive behavior.

Further Discussion of Compilation

It would be useful to consider some further examples of proceduralization and composition, the two subprocess of compilation, for the domain of LISP programming.

Proceduralization

Proceduralization can be illustrated in its pure form by the following example: In GRAPES there is a production that will retrieve function definitions from long-term memory and apply them:

```
IF the goal is to code a relation on an argument
  and there is a LISP function that codes this relation
THEN use this function with the argument
  and set as a subgoal to code the argument
```

In this production, *relation* and *function* are variables which allow the production to match different data. The second line of the condition might match, for instance, "CAR codes the first member of a list." If this rule is proceduralized to eliminate the retrieval of the CAR definition, it becomes

```
IF the goal is to code the first member of a list
THEN use CAR of the list
  and set as a subgoal to code the list
```

This is achieved by deleting the second clause in the first production that required memory retrieval. Rather, the rest of the production is made specific to the relation first element and the function CAR. Now a production has been created which can directly recognize the

application of CAR. This will result in a reduction in the amount of long-term memory information that needs to be maintained in working memory.

Composition

As an example of pure composition, suppose one wanted to add the first member of List1 to List2. Then the following two operators would apply in sequence:

```
IF the goal is to add an element to a list
THEN CONS the element to the list
    and set as subgoals to code the element
    and to code the list
```

```
IF the goal is code the first member of a list
THEN use CAR of the list
    and set as a subgoal to code the list
```

The first rule above would apply binding *an element* to "the first member of List1" and *a list* to "List2". The second production would apply binding *a list* to "List1". A simple case of composition would involve combining these two productions together to produce

```
C4
IF the goal is to add the first member of one list to another list
THEN CONS the CAR of the first list to the second list
    and set as subgoals to code the first list
    and to code the second list
```

Such compositions collapse repeated sequences of coding operations to create macro-operators. The result is a speed-up in coding. A major issue concerns what productions to compose together. The above example is a fairly simple case of collapsing two levels of a goal tree into one. However, in some cases, such as when Figure 1 was collapsed into two productions, many productions can be collapsed. GRAPES determines what productions to collapse by inspecting the goal tree. There are two types of goals for purposes of composition. There are *inherent goals* and there are *planning goals*. Inherent goals are intrinsic parts of the problem domain. For current purposes inherent goals are all variants of writing code. The important feature of inherent goals is that in achieving them one achieves part of the original task. On the other hand, planning goals produce results that are used to guide solution of the original problem but the results themselves are not part of the original problem. In Figure 1 the inherent goals are "CODE THE FUNCTION", "CODE THE RELATIONSHIP", "CODE THE ARGUMENT", and "WRITE THE VARIABLE"; all the rest are planning goals.

Composition collapses productions in one of two ways. One possibility is to eliminate the planning goals that are intermediate between two inherent goals. This is what happened when C1, given earlier, was formed for Figure 1. Composition formed a rule that went from the goal of "CODE THE FUNCTION" to the goal of "CODE THE RELATIONSHIP". In doing this it compiled out the planning process and simply left in the products of that planning. The same process underlies the formation of C2 from the goal tree in Figure 1.

The second possibility for composition is illustrated in the case of C4 given earlier under the discussion of pure composition. Here it starts with the goal of adding the first element of one list to another, skips over the intermediate goal of coding the first element of the list, and sets the goals of coding the two lists. In so doing this it is basically creating macro operators somewhat similar to STRIPS (Fikes and Nilsson, 1971).

This learning scheme requires that the learner be able to identify what subgoals were essential to the problem solutions and which are only intermediate to the final solution.

It needs to be emphasized that neither proceduralization nor composition eliminate the original production rules from which they were built. Rather the new compiled rules just serve as additional supplemental rules to produce better performance in certain circumstances.

The effect of the knowledge compilation process is to create a set of productions that mirror the structure of LISP. They may explicitly involve LISP functions like CAR or COND or LISP programming techniques like tail-recursion. These productions will preserve the inherent goals which are specific to LISP and will delete the planning goals involved in domain-general processes like structural analogy. Thus representative productions become (see Anderson, Sauters and Farrell, 1982):

```
IF the goal is to code the second member of a list
THEN use CADDR to set a subgoal
    to code the list.

IF the goal is to obtain all the elements which have
    a relation to any member of a list
THEN use MAPCONC and set as subgoals
    To code a function that will return all the elements that
        have relation to the argument.
    To code the list.
```

Generalization

Generalization is another learning mechanism which can be defined on productions. As it is typically formulated (eg., Anderson, Kline, and Beasley, 1980), it takes a pair of productions and generates what is called the *maximal common generalization*. This is the most specific production that will apply in all the circumstances of either original production and have the same effect as the original productions. So, for a simple example, if we had the following pair of productions:

```
S1
IF Fred is rich
    and Fred is ugly
    and Fred is smart
    and Fred is of medium height
THEN Fred is in club 1.
```

S2

IF Gail is rich
 and Gail is ugly
 and Gail is stupid
 and Gail is of medium height
 THEN Gail is in club 1.

Then the following production would be formed as the generalization of the two:

G1

IF =person is rich
 and =person is ugly
 and =person is of medium height
 THEN =person is in club 1.

where =person is a variable. The first productions can be viewed as specific observations and G1 is a generalization formed from these observations. Note G1 is formed both by deleting condition clauses and by replacing constants by variables. These are the two transformations that occur when one moves to generalizations.

The interesting observation is that compilation results in clause deletion and replacement of constants with variables. Compilation deletes clauses associated with omitted goals and with planning. On the other hand variables can remain from the planning process in the compiled productions. Thus it appears we might be able to get the effect of generalization through compilation.

Specifically, it appears that generalizations can be formed through the process of compiling analogies. The basic framework is as follows: The system has instances committed to memory as declarative facts. When a new instance is encountered, the system compares it to a memorized instance and uses the structure of the memorized instance to guide response. Then compiling this compare-and-respond behavior produces a production which is a generalization from the two instances. This is basically what happened in the case of forming C1 and C2 from Figure 1. I would like to go through a simpler example here that will make this process more transparent. Then, I will discuss the psychological evidence that indicates this is the correct conception of generalization in humans.

Consider a very simple production set that will classify new instances according to their similarity to studied instances:

P1

IF the goal is to classify =object
 and =reference has been studied
 THEN set as subgoals to compare =object to =reference
 and to determine if =object is in the same category,
 as =reference.

P2

IF the goal is to compare =object to =reference
 THEN initialize the measure of overlap
 and set as subgoals to match all the features of =object
 to =reference.

P3

IF the goal is to match =feature to =reference
 and =reference has =feature
 THEN increment the measure of overlap.

P4

IF the goal is to match =feature to =reference
 and =reference does not have =feature
 THEN decrement the measure of overlap.

P5

IF the goal is to determine if =object is in the same category
 as =reference
 and the measure of overlap is above threshold
 and =reference is in =category
 THEN =object is in =category.

P6

IF the goal is to determine if =object is in the same category
 as =reference
 and the measure of overlap is below threshold
 THEN POP failure.

This production set will keep comparing the to-be-classified object to candidate references from memory until a high similarity reference is found. Then it will categorize the object as in the same category as the high similarity reference.

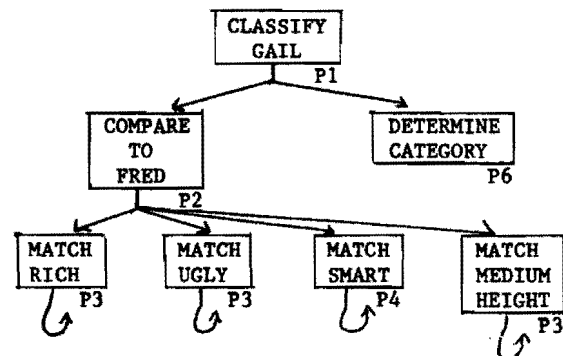


Figure 2.

Suppose the system has committed to memory that Fred is in category 1 and he was rich, ugly, smart, and of medium height. Then the system is asked to categorize Gail who is rich, ugly, stupid, and of medium height. Figure 2 illustrates the goal tree generated by GRAPES in applying this production set to the classification problem. It found matches on three features and mismatch on one--which was sufficient to exceed threshold. The compilation process regarded all the similarity comparison as planning goals and compiled all of this out and formed the following single production:

IF the goal is to classify =object
 and =object is rich
 and =object is ugly
 and =object is of medium height
 THEN =object is in club 1.

This production is essentially identical to the generalization G1 given earlier. Thus, compiling the process of make an analogy will result in a generalization.

What has been described is a procedure for getting the same effect as generalization. What reason is there to believe that it is the process underlying generalization rather than the more direct generalization mechanism? First, it is more plausible within the ACT production system framework. To produce a generalization in ACT it is necessary to start out with very specific productions like S1 and S2 given earlier. This is because the generalization mechanism only works on productions. These highly specific productions are patently non-intuitive and moreover the ACT theory would claim it is not possible to directly form such productions. So, an ACT theory that produces generalization through compilation is more plausible than one that uses the direct generalization mechanism.

Second, independent of the ACT framework, the evidence is against an automatic generalization mechanism that has no strategic component. Both Elio and Anderson(1983) and Kline(1983) have shown that the generalizations a subjects emerge with will depend on both their problem-solving set and the order of the examples. Thus, it does not seem that humans extract always and only the maximal common generalizations. The generalizations they do extract are determined by what they are looking for. This is better modelled by a system like the preceding that leaves similarity detection at the strategic level.

Third, notice that the system can classify on the basis of a single studied instance. A generalization scheme requires at least two instances to form a generalization before novel instances can be classified. Elio & Anderson (1981) showed that subjects can classify novel instances on the basis of similarity to a single studied instance when there is no possibility of generalization. This is consistent with the current conception. In addition, Elio and Anderson showed that subjects could better classify a novel instance when there was a relevant generalization that could be formed from two studied instances--even when overall similarity of studied to novel instances was kept constant. This shows that generalizations also do occur and all classification is not a matter of analogy--again consistent with the current scheme.

Also, experiments have been done where subjects can classify novel instances after studying instances unaware of their classification structure(Brooks, 1978). For instance, subjects learn to associate animals and cities to stimuli in a simple paired-associate experiment. After doing the paired-associate task, they are told that all the stimuli associated to new-world animals and cities form one category and all stimuli associated to old-world responses form another category. Subjects were unaware of the old-world versus new-world dimension during study. They can now reliably classify new instances as to whether they are old or new world stimuli. Since subjects were not aware of the classification structure at study there was no opportunity to form generalizations. The classification must be on the basis of comparing specific test stimuli to specific study stimuli.

Carbonell(1983) provides an elaborate discussion of how analogy might be used to guide problem-solving which goes beyond the discussion here. He also speculated on how generalizations might emerge from the analogy process. He proposed that the subject would store analogical solutions and generalize from these in just the way

typical generalization from examples works. This discussion shows that generalizations can emerge as a byproduct of the analogy process without a separate generalization phase.

Discrimination

Less work has been done on discrimination in the knowledge acquisition literature and there is less consensus about how to do discrimination. However, the general situation for discrimination is one in which there is an overly general rule and there is a need to restrict its range. I personally doubt that there is a single way to form discriminations and this probably accounts for the lack of consensus in the literature. I certainly do not think there is a uniform discrimination process in the human. However, for purposes of discussion I will focus on a kind of discrimination called *action discrimination* in the ACT theory(Anderson, 1982, 1983). Again for clarity I will illustrate it with respect to simple classifications but Anderson(1982) can be consulted for more complex applications.

Suppose we have the following general production:

G2

IF = person is rich
 = person is of medium height
 THEN = person is in club 1.

Suppose this rule is applied to David who is smart, good-looking, rich, and of medium height. It would classify David as being in club 1. However, suppose it is subsequently learned that David is in club 2. Thus, an error has been made and the system sets out to form a discrimination that will correct this error.

In forming a discrimination the system looks for some past instance where the rule correctly applied. Suppose, in this case, Fred is retrieved as someone correctly classified by the rule who is smart, ugly, rich, and of medium height. The discrimination mechanism looks for some feature which was true of the successful instance but was not true of the unsuccessful instance. In this case, the only difference is that David is good-looking while Fred is ugly. Therefore, a new production is formed that has this added feature:

D1

IF = person is rich
 and = person is of medium height
 and = person is good-looking
 THEN = person is in club 2.

This is called an action discrimination because it has a different action than the production(G2) from which it was formed. D1 does not replace G2 but because of the specificity ordering of productions D1 will take precedence over G2 whenever both match. G2 will only apply when the person is not good-looking. Note in forming a action production the system both adds to the condition of a production and changes the action.

The interesting thing to note is that compilation can have the effect of adding to the condition of a production and changing the action. When we compose two productions, P1 followed by P2, we can add to the conditions of P1 some of the conditions of P2 and we can change the action of P1 to the action of P2. Thus, it appears that we might be able to get the effect of discrimination through compilation.

The basic scheme for getting discrimination through compilation is to have the system consciously and deliberately go through the steps involved in forming a discrimination. This means doing discrimination by the computation of productions rather than having it be an automatic process sitting above the production system. Thus, a sequence of productions will compute a discrimination. Compiling that sequence will result in a discriminated production. As we will see, doing this is not as technically straightforward as the generalization case. This is because it would be necessary to inspect the contents of productions to perfectly mimic the automatic discrimination process whereas this is not necessary to mimic generalization. In the ACT theory one production cannot inspect the contents of another. Still we can get nearly the effect of automatic discrimination in a fairly plausible way.

As noted earlier, there are multiple different ways of forming discriminations. This section will focus on the action discrimination as just sketched out. However, I think this general scheme will extend to other types of generalization. This scheme involves having the productions compute the discrimination and then compiling that computation. After illustrating compilation of an action discrimination, I will consider the evidence that this view of discrimination is psychologically correct.

This example will be developed within the framework of productions P1-P6 given earlier for classifying objects by analogy. The learning situation I have in mind is one in which the subject makes a classification of an object, receives feedback, and if the classification is incorrect the subject goes about trying to find a discriminating feature. We will assume the subject has compiled the following too-general production:

G2*
 IF the goal is to classify =object
 and =object is rich
 and =object is of medium height
 THEN =object is in club 1.

This is just a variant of G2 given earlier but now set up for the current framework. As earlier we will assume it misapplies to classify David who is smart, good-looking, rich, and medium height as in club 1. An error is detected and a goal is set to reclassify David.

The following three productions are relevant to the correction of a misclassification:

P7
 IF the goal is to classify =object
 and =object is in =category1
 but =object was classified as in =category2
 and =reference is in =category2
 THEN remove the classification of =object
 and set as subgoals to find a difference between =object
 and =reference
 and then to reclassify =object
 and then to correct the classification of =object.

P8

IF the goal is to find a difference between =object
 and =reference
 and =object is in =category1
 and =object has =feature1 on =dimension
 and =reference has =feature2 on =dimension
 and =feature1 is different from =feature2
 THEN conclude =object is in =category1 because of =feature1.

P9

IF the goal is to correct the classification of =object
 and =object has been classified as in =category2
 but =object is in =category1 because of =feature1
 THEN change the classified category of =object to =category1.

Production P7 will apply to correct the situation. It selects some instance that was in the incorrect category. Suppose again it selects Fred who is rich, ugly, smart, and of medium height. This is an instance the overgeneral production G2* would have correctly classified. There is no guarantee that the instance selected by P7 will be one that fits the overgeneral production, because P7 cannot inspect G2*. However, the spreading activation retrieval mechanisms in ACT (see Anderson, 1983) would tend to select an instance that overlaps highly with the current instance and hence is matched by the rule that classified the current instance. P7 sets goals to find a difference between David and Fred and then to reclassify David.

Production P8 will note that David is good-looking while Fred is ugly and so enter into memory the fact that "David is in club 2 because he is good-looking". Then the system will return to the goal of classifying David once again. Again G2* would apply to misclassify David as in club 1, but now production P9 can apply to correct the classification of David. P7 followed by P8 followed by G2* followed by P9 constitute a goal tree that can be composed together and proceduralized. The result will be the following production:

D1*

IF the goal is to classify =object
 and =object is rich
 and =object is of medium height
 and =object is good-looking
 THEN =object is in club 2.

This is essentially D1 formed earlier by the automatic discrimination mechanism.

There are two essential features that distinguish this means to discrimination from the automatic discrimination of the current ACT theory. First, it requires that the learner initially make a classification of the object and then correct that classification if in error. Second, it requires that the learner make a conscious hypothesis about what distinguishes the current instance from prior instances that were in the hypothesized category. Both of these features have been confirmed in two series of experiments by Lewis and Anderson (in preparation). One series of experiments involved rules for proving triangles congruent in geometry while the other series involved rules for travelling through a maze. In both series subjects were given overgeneral rules to start with and had to discriminate these rules.

Subjects who passively studied instances failed to learn the discriminating features. Subjects only learned when they made active hypotheses about the correct rule which could then be disconfirmed. Second, the only subjects to learn were those who had some conscious awareness of what the discriminating features were. Thus, we found that subjects only discriminated when both conditions were satisfied for the current scheme of discriminating through compilation.

General Conclusions

There is ample evidence for something close to knowledge compilation as a basic process of human learning (Anderson, 1982; Anderson, Farrell, and Sauer, 1983; Neves and Anderson, 1981). The evidence had always been rather scarce for the ACT mechanisms of generalization and discrimination and more recently we have been gathering some rather negative evidence. However, it seemed clear that human learning required generalization and discrimination in some form even if not the ACT form. It now appears that the ACT knowledge compilation mechanisms can produce discrimination and generalization in a more plausible form.

It is something of an embarrassment that we worked so long with the ACT mechanisms before realizing how the knowledge compilation mechanisms could be recruited to provide the effect of generalization and discrimination. This is because we thought of knowledge compilation as simply making more efficient existing paths of processing rather than enabling novel paths of processing. To get novel behavior it seemed that we needed inductive mechanisms of learning such as generalization and discrimination. What we failed to recognize was that if we compiled the results of acting on the basis of similarity detection and difference detection we could produce productions that enabled novel behavior.

References

- Anderson, J.R. Language, Memory, and Thought. Hillsdale, NJ: Erlbaum, 1976.
- Anderson, J.R. Acquisition of cognitive skill, Psychological Review, 1982, 89, 369-406.
- Anderson, J.R. The Architecture of Cognition. Harvard University Press, 1983a.
- Anderson, J.R. Acquisition of proof skills in geometry. In R. S. Michalski, J.G. Carbonell, and T.M. Mitchell (Eds.), Machine Learning. Palo Alto, CA: Tioga press, 1983b.
- Anderson, J.R., Sauer, R., and Farrell, R. Learning to Plan in LISP. ONR Technical Report 82-1. Carnegie-Mellon University, Pittsburgh, PA., 1982.
- Anderson, J.R., Kline, P.J., and Beasley, C.M. A general learning theory and its application to schema abstraction. In G.H. Bower (Ed.) The Psychology of Learning and Motivation. Vol 13, New York: Academic Press, 1979.
- Anderson, J.R., Kline, P.J., and Beasley, C.M. Complex Learning Processes. In R.E. Snow, P.A. Frederico, and W.E. Montague(Eds.) Aptitude, Learning, and Instruction, Vol II. Hillsdale, NJ: Erlbaum, 1980.
- Brooks, L. Nonanalytic concept formation and memory for instances. In E. Rosch and B.B. Lloyd(Eds.), Cognition and Categorization, Hillsdale, NJ: Erlbaum, 1978.
- Carbonell, J.G. Learning from analogy: Formulating and generalizing plans from past experience. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell(Eds.) Machine Learning. Palo Alto, CA: Tioga Press, 1983.
- Elio, R. and Anderson, J.R. The effects of category generalization and instance similarity on schema abstraction. Journal of Experimental Psychology: Human Learning and Memory, 1981, 7,397-417.
- Elio, R. and Anderson, J.R. The effects of information order and learning mode on schema abstraction. Memory and Cognition, 1983, in press.
- Fikes, R.E. and Nilsson, N.J. STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence, 1971, 2, 189-208.
- Hayes-Roth, F. and McDermott, J. Learning structured patterns from examples. Proceedings of the Third International Joint Conference on Pattern Recognition, 1976, 419-423.
- Jeffries, R., Turner, A.A., Polson, P.G., and Atwood, M.E. The processes involved in designing software. In J.R. Anderson(Ed.) Cognitive Skills and their Acquisition. Hillsdale, N.J.: Erlbaum, 1981.
- Langley, P. A general theory of discrimination learning. In D. Klahr, P. Langley, and R.T. Neches (Eds.) Self-modifying Production System Models of Learning and Development. Cambridge, Mass.: Bradford Books, 1983.
- Lewis, C.H. Production System Models of Practice Effects. Ph. D. dissertation, University of Michigan, 1978.
- Lewis, M. and Anderson, J.R. The role of feedback in discriminating problem-solving operators, In preparation, 1983.
- Michalski, R.S. Learning from observation: Conceptual clustering. In R. S. Michalski, J.G. Carbonell, and T.M. Mitchell(Eds.) Machine Learning Palo Alto: Tioga Press, 1983.
- Mitchell, T.M. Version Spaces: An approach to concept learning. Ph.D. dissertation, Stanford University, 1978.

Neves, D.M. and Anderson, J.R. Knowledge compilation: Mechanisms for the automatization of cognitive skills. In J. R. Anderson (Ed.) Cognitive Skills and their Acquisition. Hillsdale, N.J.: Erlbaum, 1981.

Sauers, R. and Farrell, R. GRAPES User's Manual. Technical Report ONR-82-3, 1983.

Vere, S.A. Induction of concepts in the predicate calculus. Proceedings of the Fourth International Joint Conference of Artificial Intelligence, 1975, 281-287.

Winston, P.H. and Horn, B.K.P. LISP. Reading, MA: Addison-Wesley, 1981.

LINEAR SEPARABILITY AND CONCEPT NATURALNESS

Douglas L. Medin
Psychology Department
University of Illinois
Champaign, IL 61801

ABSTRACT

Many theories of human concept learning and some models for machine pattern recognition imply that linearly separable categories should be easier to master than categories which are not linearly separable. When categories are linearly separable, classification can be thought of as analogous to a summing of evidence against some criterion. An accumulating body of evidence suggests that such a summing of probabilistic evidence is not very natural for learners and that linear separability is not an important constraint on human classification performance. When learners perceive the various attributes or sources of evidence to be related to some superordinate theory, however, summing of evidence becomes more natural and linearly separable categories are easier to master than nonlinearly separable categories. These results are consistent with the idea that concept naturalness cannot be described solely in terms of abstract structural properties (such as the presence or absence of linear separability) and that people's theories about categories may determine which operations or descriptors are natural.

Key terms: Ill-defined categories, classification learning, linear discriminant functions, prototype models, machine pattern recognition, concept naturalness.

I. INTRODUCTION

1.1. Main focus

Classification is a fundamental cognitive task. Our entire system of naming objects and events, talking about them and interacting with them presupposes the ability to group experiences into appropriate categories. Since experiences might be partitioned in a limitless variety of ways, an important question is why we have the categories we have and not others. That is, what structural principles or constraints characterize natural concepts? We will assume that natural concepts should be easy for people to learn although it would be incorrect to make the converse assumption that easily learned concepts necessarily are natural. As will be seen, alternative theories imply differing constraints on category structure.

1.2 Concept structure

There is general agreement that natural categories such as bird, fruit, furniture, and tools are structured so that exemplars within a category are more similar to one another than to exemplars from alternative categories. There is considerable disagreement, however, concerning the rigidity of this structure.

1.2.1 The classical view

One view is that all natural concepts are characterized by simple sets of defining properties or attributes that are singly necessary and jointly sufficient to determine category membership (e.g., Katz & Postal, 1964). A candidate exemplar either does or does not possess these defining properties and in this sense classical view concepts are well-defined.

1.2.2 The probabilistic view

For a variety of reasons the classical view of concepts has fallen into disfavor (see Smith & Medin, 1981, for a detailed review). The consensus now has it that most natural concepts are not well-defined but rather are based on relationships that are typical or only on the average true. Members of a category may vary in the number of characteristic properties they possess and correspondingly vary in the degree to which they are judged to be good examples (typical) of the concept (Rosch, 1973; Smith, Shoben & Rips, 1974; Mervis & Rosch, 1981). For example, robins have the characteristic properties of flying, building nests in trees, and singing and are rated to be better examples of the category bird than are ostriches who manifest none of these properties (McCloskey & Glucksberg, 1978). Not only is it the case that category membership judgments appear to be based on characteristic properties, in many cases laypeople and experts alike cannot specify the defining properties of concepts (e.g., furniture) that are required by the classical view. The idea that many natural concepts are ill-defined will be referred to as the probabilistic view.

1.3 Learning models for ill-defined categories

1.3.1 Models and constraints

If many natural categories do not have defining properties, how do people acquire and use them? Candidate models and theories exist in

both cognitive psychology and machine learning. There are certain formal similarities in models across these domains that suggest some constraints on concept naturalness.

1.3.2 Independent attribute models

Posner and Keele (1968) proposed that people form an impression of the central tendency of a category as a result of experience with exemplars and that categorical judgments come to be based on this central tendency or prototype. An ostrich, for example, would be judged to be a bird and not a mammal because it is more similar to the bird prototype than to the mammal prototype.

What does prototype theory imply about the structure of categories? The main constraint is that the process has to work in the sense that all category members will be accepted and all nonmembers rejected. If it happened that canaries were mammals and not birds, a prototype process would not work because canaries have many characteristic properties of birds and few characteristic properties of mammals.

There are corresponding models in the literature on machine pattern recognition (e.g., Nilsson, 1965). Of the many elegant and powerful pattern recognition algorithms that have been proposed (see, for example, Chen, 1973), particular interest attaches to a special case of discriminant analysis, namely, linear discriminant functions. If A is a pattern having n properties $A = (a_1, a_2, \dots, a_n)$, then $g(A) = w_1 a_1 + w_2 a_2 + \dots + w_n a_n + w_{n+1}$ is a linear discriminant function where $w_1, w_2, \dots, w_n$ are property weights (and w_{n+1} is an additional constant included for computational convenience) associated with the set of properties.

If exemplars (each having n properties) from two different categories are to be classified, then one can attempt to find a set of weights on these properties such that the linear discriminant function yields higher values to all instances of one class than to those of the other. If such a function can be found, then the categories can be partitioned on the basis of the discriminant function and the categories are said to be linearly separable. This constraint can also be described in geometric terms. Two categories are linearly separable when a boundary exists between the two sets of instances, this boundary being a straight line if the instances are described in two-space, a plane if the instances are described in three-space, and so on.

There is a formal similarity between linear discriminant functions and prototype models. One way of thinking about classifying stimuli on the basis of similarity to prototype is that it involves a summing of evidence against a criterion. If an instance has a sufficient number of bird properties then it will be classified as a bird. Whether one thinks of the process as a global similarity judgment or a summing of evidence, for a prototype process to work perfectly the categories must be linearly separable. Whichever formal language one adopts, models based directly

or indirectly on a weighted function of properties, considered independently, imply that linearly separable categories should be easier to master than nonlinearly separable categories. Because this class of models assumes that properties are processed independently, they will be referred to as independent attribute models.

1.3.3 Relational coding models

There is a set of classification models that do not imply that linear separability is an important constraint on categories. These alternative models have in common the assumption that classification involves in one way or another combinations or interactions of properties or attributes (e.g., Anderson, Kline, & Beasley, 1979; Hayes-Roth & Hayes-Roth, 1977; Medin & Schaffer, 1978; Michalski, 1980). We refer to this class of models as relational coding models.

1.3.4 Contrasts between independent attribute and relational coding models

Since it is difficult to specify the attributes comprising natural concepts, researchers interested in developing classification theories have often turned to artificially constructed categories. The structure of these categories is designed to mirror some property of interest, allowing one to examine learning without the additional burden of having to deduce the underlying attributes. This is the strategy that we have adopted to evaluate the importance of linear separability in human categorization. Specifically, two categorization tasks are set up to be similar in major respects, except in one case the categories are linearly separable and in the other case they are not. Independent attribute models imply that the task involving linearly separable categories should be easier to learn than the task not conforming to this constraint.

II. EVIDENCE BEARING ON LINEAR SEPARABILITY

2.1. General experimental design

Categories that either are or are not linearly separable are shown in Figure 1 (taken from Experiment 1 of Medin & Schwanenflugel, 1981). In this example, stimuli are comprised of values on four component dimensions, described in terms of a binary notation. The value 1 represents the typical or characteristic value for members of category A and the value 0 is typical for category B members. No defining features exist to partition categories A and B, but the linearly separable categories in the top half of Figure 1 can be separated on the basis of characteristic features. One simple way to see this is to focus on the first three dimensions comprising category members. Every category A member and no category B member has two of the three characteristic values (value 1) for category A. Therefore, every exemplar could be unambiguously classified by checking to see if a majority of its values on the first three dimensions were consistent with category A or with category B.

The categories shown in the bottom of Figure 1 have the same overall distribution of values

but are not linearly separable. The difficulty is that exemplar A₁ has more values characteristic of category B than of A. Only its value on the first dimension is typical for category A, and it would not help to give the first dimension a large weight because then exemplar B₂ would erroneously be categorized as an A. One can see that A₁ and A₂ are both placed in category A, and B₁ and B₂ are placed into category B. The categories are not linearly separable. Thus, if linear separability is as important in human classification as independent attribute models imply, then it should be easier for people to learn to categorize the stimuli in the top of Figure 1 than to master the task in the bottom of Figure 1.

LINEARLY SEPARABLE CATEGORIES

<u>Category A</u>					<u>Category B</u>				
<u>Exemplar</u>	<u>Dimension</u>				<u>Exemplar</u>	<u>Dimension</u>			
	D ₁	D ₂	D ₃	D ₄		D ₁	D ₂	D ₃	D ₄
A ₁	1	0	1	1	B ₁	1	0	0	1
A ₂	1	0	1	0	B ₂	0	0	1	0
A ₃	1	1	0	1	B ₃	0	1	0	0
A ₄	0	1	1	0	B ₄	0	0	0	1

CATEGORIES NOT LINEARLY SEPARABLE

<u>Category A</u>					<u>Category B</u>				
<u>Exemplar</u>	<u>Dimension</u>				<u>Exemplar</u>	<u>Dimension</u>			
	D ₁	D ₂	D ₃	D ₄		D ₁	D ₂	D ₃	D ₄
A ₁	1	0	0	0	B ₁	0	1	1	0
A ₂	0	1	1	1	B ₂	1	0	0	1
A ₃	1	1	1	0	B ₃	0	0	0	0
A ₄	1	0	1	1	B ₄	0	0	0	1

Figure 1. Abstract representation of the alternative categorization tasks. (Each task involves eight stimuli varying along four dimensions.)

2.2. Summary of results

The results are easy to summarize: there is no evidence that linear separability provides an important constraint on human categorization. Medin and Schwanenflugel (1981) reported a series of four studies which varied the type of stimulus material (geometric shapes or photographs of faces), instruction (focus on abstraction versus memorize exemplars), and category size (from four exemplars per category to an unlimited number of exemplars per category). In no case were linearly separable categories easier to learn than nonlinearly separable categories. In experiments where linear separability was pitted against number of cases of high between-category similarity (a factor which relational coding models imply is

important), the nonlinearly separable categories actually were easier to learn.

There is also much earlier work bearing on linear separability. In their classic monograph examining the relation between identification learning (learning exemplars) and classification, Shepard, Hovland, and Jenkins (1961) employed procedures where in some cases the categories were linearly separable and in other cases they were not. Since this contrast was not the object of their study, between-category similarity was not fully controlled and should have favored the linearly separable categories. Nonetheless, across variations in stimulus properties and in learning conditions Shepard et al. found no evidence that linearly separable categories were easier than nonlinearly separable categories.

More recently we have conducted additional studies using still further variations on stimuli, number of categories, and learning procedures. In no case has linear separability mattered. Overall, these findings are more compatible with relational coding models than with independent attribute models.

III. FURTHER CONSIDERATIONS

3.1. Typical values and summing evidence

The failure of linear separability to provide an important constraint on categorization might arise either because people have difficulty in learning the typical (or modal) values for a category or because they do not find it natural to employ a summing of typical values as a decision rule for categorization. Additional observations suggest that summing probabilistic evidence is the key factor.

First of all, when people are asked to provide that characteristic or typical values at the end of training they are quite accurate. Furthermore, procedures that should facilitate the learning of modal values do not differentially facilitate the learning of linearly separable versus nonlinearly separable categories. For example, Murphy (1983) gave participants two runs through the category exemplars prior to learning during which the exemplars were either blocked by category or randomly intermixed. Blocking facilitated learning but this facilitation was as large for nonlinearly separable categories as for linearly separable categories. Finally, we often ask participants at the end of the experiment how they learned the task, and only rarely do they mention anything like a summing of characteristic values. Instead, their descriptions more commonly are in terms of rules plus exceptions or a disjunction of conjunctions, descriptions that are more consistent with relational coding models.

3.2. Summing evidence and naturalness

The above observations suggest that participants in our experiments do not find it natural to sum up evidence to make categorizations, even when a two out of three rule would work perfectly.

Strategy might be to assume that this is a fixed constraint and that results that hold for one realization of abstract category structure will hold for all realizations of that same abstract structure. The apparent generality of our results supports this approach. An alternative strategy, which we have followed, is to consider the possibility that what is natural may interact with the particular knowledge structures or theories that might be brought to bear on a problem.

IV. THEORIES AND LINEAR SEPARABILITY

4.1. Theories and summing evidence

Our basic idea is that if learners perceive the various attributes or sources of evidence to be related to some superordinate theory, then the idea of summing evidence may become more natural. In terms of induction models based on descriptors such as Michalski's (1980) rule-guided system, the theory might suggest that a new descriptor based on a summing of attribute values be constructed and evaluated.

4.2. Additional linear separability experiments

4.2.1. Linear separability

The logic of our recent experiments is quite simple. Participants are tested on category structures that either are or are not linearly separable and by means of assignment of dimensional values to the abstract category structure or by instructions we either do or do not provide the basis for a theory about the categories to influence performance.

4.2.2. Experiments

In one recent experiment (conducted in collaboration with Tim Murphy) the stimulus properties were descriptions of behaviors of people. The people belonged to one of two categories which were given arbitrary names. The behaviors were relevant to the trait dimensions of talkative, honest, cooperative, or cautious. For the theory condition, the typical values for one category all corresponded to a single trait and the typical values for the other category corresponded to its polar opposite. For example, if the superordinate trait were talkative then the typical behaviors for one category would be things like "told a lot of funny stories at a party" and "went out of his way to talk to a neighbor," and the corresponding behaviors in the other category would be things like "didn't interrupt others when they were talking at the party" and "remained silent when others were gossiping about their neighbor." In the control condition each of the four behaviors associated with the exemplars was related to a different one of the four traits. In both conditions the abstract description or notation would be analogous to that shown in Figure 1 but in the theory condition the dimensions corresponded to locations in which the behaviors occurred (park, home, party, store) and in the control condition the dimensions corresponded to the four different traits. We expected that in the theory condition participants would

perceive the behaviors as relevant to a unifying trait and would find it natural to use a summation of typical values as a decision rule.

The results fit with this expectation.

There was a significant interaction between category structure and theory versus control condition. Participants in the theory condition learned the linearly separable tasks more readily than the nonlinearly separable task (mean errors during learning 27.4 versus 38.1) whereas the opposite result obtained in the control condition (mean errors for linearly separable group = 49.9, for nonlinearly separable group = 38.9).

In a related followup experiment (conducted in collaboration with Gerald Dewey and Glenn Nakamura), the attributes were descriptions of objects. The description pairs were "made of metal" versus "made of rubber," "medium weight" versus "light weight," "flat surface" versus "irregular surface," and "easy to grasp" versus "difficult to grasp." The modal or typical values for one category were the first mentioned value of each pair and the modal properties for the other category were the opposite values. In the control condition standard instructions were used. In the theory condition an additional sentence was added: "It may help you to know that the objects in one of the categories might serve as reasonable substitutes for a hammer."

The results were straightforward. In the control condition the nonlinearly separable task was easier (mean errors 52.2 versus 57.6) but in the experimental condition the linearly separable task was easier (mean errors of 31.2 versus 43.0).

Finally, we have evidence that the theory and the attribute structure must be compatible if a summing of evidence is to be evoked. William Wattenmaker and I performed an experiment where the stimuli were properties of animals and participants either were or were not informed that the categories would map onto the categories predator and prey. We did not observe an interaction of category structure and instructions. We think the absence of an interaction derives from the fact that summing of evidence was not natural. For example, it's a good adaptation for prey to be armored or to live in a tree, but it might not have been natural for participants to think that both being armored and living in a tree is a still better adaptation.

V. SUMMARY AND IMPLICATIONS

The overall pattern of results provides a relatively clear but mixed picture. A preponderance of evidence suggests that linear separability is not a major constraint on human classification. This evidence is consistent with relational coding models and inconsistent with independent attribute models. When learners perceive the component attributes or sources of evidence to be related to a compatible superordinate theory, however, then summing of evidence becomes more natural and linearly separable categories are easier to learn than nonlinearly separable categories.

The results imply that constraints provided by different types of category structure are not independent of the types of theories or knowledge structures brought to bear on them. So while the overall pattern of results favors relational coding models, these models will need to provide an account of how these knowledge structures modify the types of operators or descriptors that are natural.

VI. REFERENCES

- Anderson, J.R., Kline, P.J., & Beasley, C.M. "A general learning theory and its application to schema abstraction." In G.H. Bower (Ed.), The Psychology of Learning and Motivation (Vol. 12). New York: Academic Press, 1975.
- Chen, H.C. Statistical Pattern Recognition. Rochelle Park, NJ: Hayden Books, 1973.
- Hayes-Roth, B., & Hayes-Roth, F. "Concept learning and the recognition and classification of exemplars." Journal of Verbal Learning and Verbal Behavior, 1977, Vol. 16, pp. 321-338.
- Katz, J.J., & Postal, P.M. An Integrated Theory of Linguistic Descriptions. Cambridge, MA: MIT Press, 1964.
- McCloskey, M.E., & Glucksberg, S. "Natural categories: Well-defined or fuzzy sets?" Memory & Cognition, 1978, Vol. 6, pp. 462-472.
- Medin, D.L., & Schaffer, M.M. "Context theory of classification learning." Psychological Review, 1978, Vol. 85, pp. 207-238.
- Medin, D.L., & Schwanenflugel, P.L. "Linear separability in classification learning." Journal of Experimental Psychology: Human Learning and Memory, 1981, Vol. 7, pp. 355-368.
- Mervis, C.B., & Rosch, E. "Categorization of natural objects." In M.R. Rosenzweig & L.W. Potter (Eds.), Annual Review of Psychology, 1981, Vol. 32, pp. 89-116.
- Michalski, R.S. "Pattern recognition as a rule-guided inference system." IEEE Transactions on Pattern Analysis and Machine Intelligence, 1980, Vol. 2, pp. 349-361.
- Murphy, T.D. Stimulus Presentation Effects and the Processing of Ill-defined Categories. Unpublished M.A. Thesis, University of Illinois, 1983.
- Nilsson, N.J. Learning Machines. New York: McGraw-Hill, 1965.
- Posner, M.I., & Keele, S.W. "On the genesis of abstract ideas." Journal of Experimental Psychology, 1968, Vol. 77, pp. 353-363.
- Rosch, E. "On the internal structure of perceptual and semantic categories." In T.E. Moore (Ed.), Cognitive Development and the Acquisition of Language. New York: Academic Press, 1973.
- Shepard, R.N., Hovland, C.I., & Jenkins, H.M. "Learning and memorization of classifications." Psychological Monographs, 1961, Vol. 75, (13, Whole No. 517).
- Smith, E.E., & Medin, D.L. Categories and Concepts. Cambridge, MA: Harvard University Press, 1981.
- Smith, E.E., Shoben, E.J., & Rips, L.J. "Structure and process in semantic memory: A featural model for semantic decision." Psychological Review, 1974, Vol. 81, pp. 214-241.

Footnote

This research was supported in part by National Institute of Mental Health Grant MH32370 to Douglas L. Medin.

How Can CHILD learn about Agreement? Explorations of CHILD's Syntactic Inadequacies

Mallory Selfridge
Department of EE and CS
University of Connecticut
Storrs, Ct. 06268

1.0 Introduction

The CHILD program has accounted for a large amount of data on child language learning, and manifests a developmental progression which is psychologically accurate [4,5,6,7]. However, its representation of language syntax is known to be inadequate in a number of ways. This is manifested in CHILD's inability to learn to understand and generate various syntactic constructions which children do learn. This paper is an initial attempt to address these issues. It first reviews the CHILD program, and then it describes those aspects of its syntactic representation upon which this performance rests. Third, it describes a subset of the problems which this representation introduces, and finally outlines possible directions for further development.

2.0 The CHILD Program

CHILD is a computer program which models the development of a child's language comprehension and generation abilities written in Franz LISP and currently running on a DEC VAX 11/780. It begins with world knowledge and language experiences similar to those received by children, and learns a subset of the word meaning and syntax which children learn. After learning, CHILD can correctly understand utterances which it previously misunderstood, and can generate English describing events it "observes."

CHILD's language comprehension process is a version of the CA program [1] which incorporates mechanisms derived from Wilks' [8] preference parsing. CHILD's analysis process combines Conceptual Dependency (CD) [2] word meanings to form a CD representing the meaning of the entire utterance. Understanding begins when the meanings of input words are placed in a short term memory. CHILD then retrieves semantic requirements associated with particular slots in the CD itself, and also syntactic features associated with those slots but specific to that particular word. It searches the short term memory for a word meaning which best satisfies those features, and fills the empty slot with that meaning. The syntactic features are formed from the positional predicates "precedes" and "follows." These relate the position of a candidate slot filler to either the word they were stored under, a filler of another slot in that

word's meaning, or a lexical function word. Each slot in the meaning of a word has a collection of features describing where in the input a filler is expected to be. In order to understand different voices, these features are organized into disjunctive "feature sets." Each set characterizes one order in which slot fillers appear. During understanding, feature set selection is performed by considering which set most successfully characterizes the input. In this paper, this representation will be called "concept order syntax"

CHILD learns syntax by acquiring syntactic features and build disjunctive feature sets. After having understood an utterance, CHILD examines a record of the input, and examines the meaning of every input word. It then examines every empty slot in each such meaning. It accesses the record of the input to find where in the input the filler for that slot occurred. It creates a description of this position using "precedes" and "follows." CHILD must then decide whether this description constitutes a new feature set or should be merged with an existing feature set. CHILD's strategy is based on a suggestion by Iba [8]. CHILD compares the features extracted from the current input with any existing feature sets: the position description is merged with a previous set only if one is a subset of the other. Otherwise, the description is learned as a new feature set.

3.0 Why is Concept Order Syntax Good?

The power of concept order syntax appears to derive first from its formulation as a set of independent features. This means that the syntactic decisions that have to be made during understanding and generation are not binary "something is either in the correct location or it isn't" decision. Rather, these decisions are more comparative: "this location is better than that." This results in smoother and more robust understanding and generating. A second advantage of independent features is that it makes learning easier. The features observed to be true in the input are merely added to those features already known to describe filler positions. This is far easier than trying to decide which word-class a word is in, for example. A third advantage of syntactic features is that it simplifies the understanding process enormously. This is because a large portion of understanding is manipulating semantic features. Having syntactic features means that syntactic processing can be handled by the same mechanisms that handle the semantic features, and at the same time. This is par-

tially responsible for CHILD's robust performance. Finally, independent features make grammaticality judgements easy, since false features during understanding allow both a non-binary measure of the extent to which a slot filler was out of position, and also provide a basis from which to describe the incorrect position. Thus the first critical attribute of concept order syntax is the representation of syntax by independent features.

The second critical attribute of concept order syntax is that it is closely tied to word meaning. That is, concept order syntax is "semantically-indexed." This is reflected in two ways. First, syntactic features themselves specify where particular slot fillers within a specific word meaning are found. Second, the features sets are stored under the meaning of the word. Notice that these two aspects are not exactly the same thing. For example, one could have traditional grammar rules stored under word meaning, or one could have syntactic features stored not under word meaning but under the basic concept which comprises not only the meaning of a particular word but perhaps several others as well.

The third critical attribute of concept order is that it is derived from mechanisms which can plausibly be attributed to human language mechanisms. Specifically, it uses order information applied to concepts, slot fillers, and lexical items, a short-term memory, the mechanisms needed to retrieve the features and test them against the items in the short-term memory, and the concept of "slot" and "slot filler" applied to a knowledge representation. Now, humans can certainly perceive linear order, have concepts and lexical items, have the equivalent of a short-term memory, and the ability to test features against mental objects. It is somewhat less clear that they have memory containing conceptual case-frames. However, almost every proposal for modelling human memory uses a variant of this technique, and some have been very successful in accounting for psychological data (Anderson, Schank).

Describing natural language syntax using a representation which has the three attributes of a) independent features, a) semantic indexing, and c) psychologically plausible components thus seems to be largely responsible for the success it has had in accounting for the data that it does. Any extension to concept order syntax designed to enhance its ability to account for complex syntactic constructions should therefore maintain semantic indexing, independent features, and the primitive components which it currently uses. Probably such extensions must add additional primitive components; in this case they should be drawn from those plausibly attributable to the human cognitive system.

4.0 What Can't Concept Order Syntax Handle?

Experience with CHILD to date has shown the existence of a number of problems with its syntactic representation. This section will describe a subset of the known problems. First, concept order syntax cannot associate alternate meaning representations with alternative syntactic forms, or

feature sets. Thus a passive sentence is understood to mean exactly the same thing as its active version, while there is generally agreed to exist some minor meaning differences concerned with "focus". More drastic differences exist between declarative sentences and questions. Consider the difference between "That is Tom" and "Is that Tom?" If concept order syntax could associate different meaning with different syntactic feature sets, then it could possibly handle sentences like these.

The second problem is that of redundancy of storage of concept order syntax. Currently, all words which require syntactic information have that information stored directly associated with them, rather than in some central location. Thus currently concept order syntax manifests "syntactic lexical specificity." For example, object words such as "table" require syntactic knowledge to specify where in the input might appear fillers for their COLOR and SIZE slots. This information is stored directly under "table," associated with its meaning. Similar information is stored under every word whose meaning has COLOR and SIZE slots. This is done for several good reasons, most notably because it supports easy learning. Unfortunately, since every object word should have virtually the same syntactic knowledge stored under it, this scheme results in tremendous storage inefficiency. Worse, consider the process of learning a new word. One learns a new word after hearing it only once, yet immediately knows where in the sentence to find the fillers of the slots. It would appear that people's knowledge of syntax is stored in some more general way, and concept order syntax must be extended to handle this.

Third, concept order syntax cannot represent "syntactic embedding," in which a word has one kind of syntax if it is part of the top level of meaning representation, and somewhat different syntax if embedded within the meaning representation. Thus concept order syntax could not generate "I want to go to the store," but rather would say "I want that I go to the store." This particular problem is one which traditional theories of syntax are supposed to handle rather well.

Finally, concept order syntax has no ability to represent "agreement". For example, CHILD using concept order syntax would understand "the dog eats" and "the dogs eats" in exactly the same way, and would not be aware of any difficulty. Agreement is a relatively widespread linguistic phenomenon, and one which CHILD should certainly be able to learn.

5.0 Possible Extensions to Concept Order Syntax

Given the above shortcomings of concept order syntax, one would be tempted to abandon it as an extreme simplification and unworthy of further attention. However, since it has supported a high-performance model of child language learning, it appears worthwhile to explore directions in which it can be extended. The first extension is to couple syntax with meaning more tightly. This is apparently required by the problem that both active

and passive versions of the same sentence are understood to mean exactly the same thing. A tighter coupling of syntax and meaning would allow passive sentences to have different meanings than their active versions. It would also allow the use of feature sets in question understanding, allowing discriminations between "Is that Tom?" from "That is Tom."

One possible technique to couple syntax and meaning more tightly is to interpose an intermediate layer of meaning representation between the feature set and the word meaning proper. In such a scheme, when a word is read in during understanding, its primary meaning would be placed on the c-list, as currently. Then understanding would proceed, and when feature set selection was completed the secondary meaning associated with that feature set would be added to the primary meaning. For example, the word "give" might have ATRANS as its primary meaning, including a FOCUS slot, and the secondary meaning associated with the feature sets would specify the filler of the FOCUS slot.

Thus when a feature set was selected, the FOCUS slot would be filled with the name of the ATRANS slot being focussed. Thus the secondary meaning in this case would be the filler of a slot in the primary meaning.

The second possible extension concerns the problem of syntactic lexical specificity. Rather than associating syntactic knowledge with individual words, common syntactic feature sets should be stored "centrally," and each word should index those centrally stored feature sets. Since the centrally stored syntactic knowledge must contain exactly the same syntactic knowledge now stored directly under each word, central storage will have no less power than the current approach. It will, however, reduce storage space make learning syntax easier, and help explain the phenomena of syntactic generalization noted above.

The third possible extension concerns the problem of syntactic embedding. In many ways this is the most difficult problem of any here, and there is no clear answer. Indeed, this question is a central one for linguistics. While a definite extension cannot be described here, the direction in which such an extension lies can be described. One way characterization of the problem is that certain syntactic features are appropriate if referring to slot fillers at the top level of meaning representation. If the slot filler is not at the top level of meaning representation, on the other hand, then other syntactic features may be appropriate. This suggests that information should be included in a syntactic feature which refers to levels of meaning representation. Specifically, this information would specify that the feature is to be activated only if the meaning containing the slot which indexes the feature is the top-level meaning representation of the sentence. Other features will have similar information activating them only if the meaning containing the slot being filled is not the top level feature. While this approach is certainly very far from an ultimate explanation, it appears to represent an improvement over the current representation. It will enable the system's performance to improve, and it will provide additional experience with the problem of syntactic

embedding.

Finally, there is the question of agreement. How can concept order syntax be extended to describe knowledge of agreement? In "the dog eats" vs. "the dogs eat", the number of dogs seems to have a direct correlation with the presence of "s" on the end of "eat." This suggests a contingent component to agreement: "if there is an 's' following 'eat' then there is one dog, otherwise there are two." Since the number of dogs is clearly semantic, and the presence or absence of "s" clearly syntactic (unless a meaning can be assigned to it), then CHILD should be able to manipulate links between syntax and semantics of the form "if syntactic feature then semantic expectation about a certain slot filler." Such a formulation may allow CHILD to understand and generate correctly agreeing inputs; if so, such a successful representation would be the first step toward learning agreement.

6.0 Conclusions

The CHILD model of child language learning has demonstrated considerable performance, but has been limited by its representation of syntax. This paper is intended to acknowledge the existence of these limitations, and to outline current progress toward reducing some of them. Such progress will enable the CHILD program to learn more complex syntactic constructions, and would allow CHILD to account for additional data. It would also provide further experience with which to uncover additional inadequacies with both concept order syntax and with CHILD itself.

References

- [1] Birnbaum, L. and Selfridge, M. (1981). Conceptual Analysis of Natural Language, in *Inside Computer Understanding: Five Programs plus Miniatures*. Schank R. and Riesbeck C.K. (eds.), Lawrence Erlbaum Associates, Hillsdale, NJ.
- [2] Iba, G. (1979). Learning Disjunctive Concepts from Examples. M.I.T. A.I. Memo #548, M.I.T., Cambridge, Mass.
- [3] Schank, R. C., (1973). Identification of Conceptualizations Underlying Natural Language. R. C. Schank and K. M. Colby (eds.). *Computer Models of Thought and Language* W.H. Freeman and Co., San Francisco.
- [4] Selfridge, M. (1980). A Process Model of Language Acquisition. Computer Science Technical Report 172, Yale University, New Haven, CT.
- [5] Selfridge, M. (1981a). Why do Children Say "Goed"? A Computer Model of Child Generation. Proc. Third Annual Meeting of the Cognitive Science Society. Berkeley, CA.
- [6] Selfridge, M. (1981b). A Computer Model of Child Language Acquisition. Proc. 7th Int. Joint Conf. on Artificial Intelligence. Vancouver, Canada.
- [7] Selfridge, M. (1982). Why Do Children Misunderstand Reversible Passives? The CHILD Program Learns to Understand Passive Sentences. Submitted to the 3rd Annual AAAI Conference, Pittsburg, PA.
- [8] Wilks, Y. (1976). Parsing English II. In Y. Wilks and E. Charniak (eds) *Computational Semantics*, North-Holland Publishing Co., NY, NY

INFERRING (MAL) RULES FROM PUPIL'S PROTOCOLS

D. Sleeman

HEURISTIC PROGRAMMING PROJECT, [1]
 DEPARTMENT of COMPUTER SCIENCE
 STANFORD UNIVERSITY
 CALIFORNIA, 94305.

ABSTRACT

This paper discusses protocols collected from 14 year-old pupils solving Algebraic tasks. In particular, it discusses several approaches to determining the mal-rule (ie an incorrect variant of a correct rule) which a pupil is using. The approaches considered include:

- an off-line analysis by the investigator,
- the definition of a number of perturbations, and attempts to see whether the pupil's answer can be explained by the application of one such perturbation.
- and most significantly, the inference of mal-rules from protocols.

This latter approach is explored in some detail and it is shown how a rule-based system has successfully inferred the mal-rules encountered in earlier protocols. The importance of a rule inference system to a practical student modelling system is stressed.

1. INTRODUCTION

Several authors have pointed out the importance of having an accurate student model when one is attempting to give remedial teaching with an Intelligent Tutoring System. Indeed many of the papers in a recent collection address this issue. [Sle 82a]. One such system BUGGY/DEBUGGY [Bro 78]. [Bur 82] attempts to model the student in terms of a predefined set of bugs and uses predefined heuristics to limit the size of the search space. Analogously, LMS - the Leeds Modelling System, uses a predefined set of rules and mal-rules (incorrect variants observed in

pupils' protocols) to define a space of possible student models. On the other hand a major emphasis with LMS has been the formulation of a search to "contain" the number of hypotheses/models considered. In this formulation, each set of tasks presented to the pupil focuses on a particular rule and it is shown that it is thus possible to dramatically reduce the search space. [Sle 81a] & [Sle 83].

LMS has been implemented as two phases. In the off-line phase, the system generates a complete and non-redundant set of models for each of the task types, and evaluates the predefined tasks against this list of models. [Sle 83]. (These models are sets of ordered productions rules, and the first rule which is satisfied fires. See figure 1a for a list of domain rules and 1b for a list of mal-rules used by pupils in protocols analysed earlier). In the on-line modelling phase the pupil is presented with a task and his response is evaluated against the various responses generated in the off-line phase. As it was anticipated that the users would only have minor manipulative errors, LMS allows the pupil merely to typo in his final answer and none of the intermediary stops (he may give the intermediary steps if he wishes). LMS moves onto the next task when the pupil has given the solution in its simplest form, or has made an error or has exceeded a predefined limit for the number of refinement cycles allowed for each example type.

1. This work was done whilst I was at the University of Leeds, UK. Essentially, this paper previously appeared in the proceedings of the 1982 European Conference on AI, held at Orsay, France. It is intended to present recent developments of this work at the 1983 Machine Learning workshop.

In the initial experiment with a group of 15 year old pupils, LMS managed to produce a comparable diagnosis to investigators who interviewed the pupils several days later, [Sle 82b]. However, more recently we have used LMS with the same data base of rules with a group of 24 14 year old pupils and the results were dramatically different. LMS was unable to analyse a high proportion of the protocols as the appropriate mal-rules had not been predefined; see Figure 2.I for an example of a protocol which LMS could diagnose, and figures 2.II and 2.III for protocols which LMS could not diagnose. (Figure 2 reports several protocols collected during the pupil interviews; analogous protocols were collected with LMS, see [Sle ap] for details). The subsidiary difficulties which arise, if one does increase the number of mal-rules considerably is that the modeller then produces multiple explanations for the same response, and secondly the speed of the on-line modeller drops dramatically as the search space becomes much greater. See [Sle 83] for a quantitative estimate for these effects. In the next section we discuss these issues in more detail.

2. An Analysis of the Difficulties Encountered

a) Multiple Explanations (Models) for a particular response One solution to this difficulty is to use an on-line task generator, to create a discriminatory task, [Sle 81b]. Alternatively, pupils could be asked to input the several intermediary steps in their solution in which case it might well be possible to discriminate between the several "illegal" paths. In particular the steps which were permissible need not be investigated only those where a correct transformation was not applied.

b) Speed of the on-line Modeller The first "solution" suggested above would not help the speed of the system, but the second might, as only those steps which had been done incorrectly need be considered.

c) On-line Modeller unable to analyse the protocol

This is by far the most significant issue and the most challenging to resolve. We shall consider several approaches in varying degrees of detail.

1. Off-line analysis by the Investigator.
2. Computational aides to assist in the analysis by the Investigator.
3. On-line use of "schema" to generate mal-rules.
4. On-line inference of (mal) rules.

2.1 Off-line Analysis by the Investigator

The protocols given in Figure 2, and the additional protocols given in [Sle ap], were analysed in this manner. My initial reaction to several of these protocols was that they were totally random and only after subsequent analyses did I begin to see the consistent pattern which I have now recorded as part of each set. (Unfortunately it was not possible to talk with the pupils immediately following the sessions with
Page 8

LMS. When I did they explained to me very clearly what they were doing [Sle ap]). Because of the difficulties I encountered I implemented the Aide outlined in the next sub-section.

2.2 Aide to assist in the Analysis of Protocols

Having done a preliminary analysis of the protocols I supposed that many of them contain a

"wild" step, and so I postulated a number of "perturbations" including:

$$MX \rightarrow M + X \dots \dots \dots [P1]$$

$$MX \rightarrow MX + M \dots \dots \dots [P2]$$

A system has been implemented which attempts to transform the original task into the pupil's answer using only one "perturbation", but as many applications of the domain's rules and mal-rules as necessary, per "path". Figure 3 gives two examples of such paths. [NB, we are NOT claiming that students undertake all the steps given in Figure 3, but merely that their answers are as if they did. Deciding the particular "grain" size DOES become an important issue when one uses the inferred model as a basis for student remediation.]

Given a reasonable number of such perturbations it is clear that the search space becomes extremely large. However, it should be added that this sub-system did provide several explanations which I had not previously considered.

2.3 Schema to generate Mal-Rules (On-line)

When carrying out interviews with the 1981 group of children some of them gave explanations for their actions which amounted to schema for creating mal-rules.

In figure 2.III we gave a substantial section of pupil AB18's protocol. In the course of our discussion he explained that he was carrying out the teacher-given algorithm of:

"Collecting all the Xs on the left hand side and collecting all the numbers on the right hand side." Pupil AB17, figure 2.II, gave a similar explanation for his action, and added that he was not really sure what to do about the "extra multiply signs".

This gives us a schema for generating mal-rules. For example given the task type:

$$M \times X + N \times X = P$$

This schema gives the following "action sides" for mal-rules:

$$X + X = P - M - N$$

$$X \times X = P + M + N$$

where in the second case the X coefficients are treated "specially", ie the coefficients of the Xs were taken across to the RHS of the equation but the signs were NOT changed. And the form given by pupil AB-17, and quoted in Figure 2.II, namely:

$$*X \times X = P - M - N$$

which he went on to "interpret" as:

$$X \times X = P - M - N$$

Its "complementary" form being:

$$X \times X = P + M + N$$

Similarly, given the task type:

$$M \times X = N \times X + P$$

This schema creates the following forms:

$$X = N + P - M$$

$$X = N + P + M$$

$$X + X = N + P - M$$

$$X + X = N + P + M$$

$$X - X = N + P - M$$

$$X - X = N + P + M$$

For example on task h, pupil AB18 suggested the use of both the third and the fifth forms (see figure 2.III).

Analogously, it appears possible to generate all the mal-rules associated with a manipulative rule by systematically removing one or more of the rule's sub-steps, [2].

2. Brown and Burton's modelling work [Bro 78] assumed that pupil errors were due to perturbations of the correct rules (cf the assumptions which we also made initially in this project, and which seems to be true for some of this domain's rules). This however, is certainly not true for the class of errors which we are discussing here. Indeed, pupils AB-18 and AB-17 do not use the usual "task segmentation" and thus do not even consider applying, at the first stage, a correct rule.

In [Sle ap] I report 3 new mal-rules, variants of SOLVE, SIMPLIFY and BRA2, which can be explained by such a mechanism, as can most of the "original" mal-rules given in Figure 1b. Note that this schema would ALSO generate many mal-rules which I have NOT yet observed.

Although these schema help us understand the protocols given in Figure 2.II and 2.III, several of those reported in [Sle ap] can not be so explained, and so a further more general approach is needed.

2.4 On-line Inference of (Mal)-Rules

There are two principal disadvantages of the approach given in Section 2.2. Firstly, because it is not data-driven the search becomes very broad. Secondly, it relies on a pre-defined set of perturbations and hence would still be unable to spot very "wild" yet consistent, mal-rules. (An analogous criticism was also noted about the approach taken in section 2.3). For instance, during the 1981 experiment with LMS we noted the following mal-rules were used consistently:

$$\begin{aligned} MX+N=P &\Rightarrow (M+N)X=P \\ M+NX=P &\Rightarrow M^*N+X=P \\ MX=N+P &\Rightarrow MX=N \end{aligned}$$

Figure 4 shows a successful path for inferring the mal-rule for the example used in section 2.2. As can be seen from the figure, this process involves the following types of steps:

1. Using Rules in the Backward direction.
(For example, the use of NTORHS in the Backward direction).
2. Use a series of heuristics to try to transform the sides of the equation into the same form as the "target" equation.
This is the rationale of the

transformation applied between stages a to b, b to c, d to e and e to f.

3. When neither of the above can be applied then INFER a rule which completes the path (in this case between stages f and g). [In an earlier study, [Sle ap] we noted that the mis-parse rule appears to be applied to the task when it is first encountered, and so for the moment we shall assume that the inference step will be the last in the chain].

So far we have formulated 10 or so heuristics, a selection of which is given in Figure 5.

Note the following points about the algorithm:

1. There are an infinite number of integer pairs, (a b) which satisfy the relationship:
$$a + b = 1.$$
where "1" is also an integer. However, by the use of the "Algebraic" form for "a" and "b", in HEURSUM1, and the subsequent instantiation of the integers to the RHS of the target equation, this set is reduced to a unique pair, which in the case of the example in Figure 4 is 6 and -8.
2. Figure 4 only shows a successful path. In general a search is involved as frequently several rules are satisfied by a particular state (eg both HEURSUM1 and HEURMULT1 are satisfied by a); and because a rule can be instantiated in several ways (eg the backwards form of NTORHS at c).
3. Some of the branching noted in the last point could be avoided by making the backward form of NTORHS more "sophisticated" by allowing it to access the target equation.

4. A further improvement in the efficiency can be achieved by segmenting the rule-set into related sub-sets, and only having the relevant sub-set in scope at any time.
5. If we look more closely at the inference step we see that it is possible to infer that

$$MX + N = P \Rightarrow X + M + N = P$$

where M, N and P are all integers. However, if the algorithm notes which symbols have remained unchanged then it is possible to infer that the substantive change is:

$$MX \Rightarrow M + X$$

Neves, [Nev 78], uses a similar approach to deduce the essential features of rules given the several steps of a solution culled from an Algebra text.

6. The instantiation heuristics are a little more complex than the rules given in Figure 5.

3. Current Activities

Many of the protocols collected in the recent experiment, [Sle ap], have been analysed, using the approach outline in the last section. To test the generality of this approach we have also used it to infer the NTORNS rule, see Figure 6.

ACKNOWLEDGEMENTS

The pupils of Abbey Grange School, Leeds provided some fascinating protocols. Jaime Carbonell, Pat Langley, K. Lovell, Steve Mallard and Tim Hepworth for discussions concerning this work. Alan Bundy suggested some significant clarifications to the ECAI version of this paper.

REFERENCES

- [Bro 78] J.S. Brown & R.R. Burton (1978). Diagnostic Models for procedural bugs in basic Mathematical Skills, in *Cognitive Science*, 2.2, pp 166-192.
- [Bur 82] R.R. Burton. (1982). Diagnosing bugs in simple procedural skills. In *Intelligent Tutoring Systems*, edited by D.H. Sleeman and J.S. Brown, published by Academic Press, pp167-183.
- [Nev 78] D.M. Neves. (1978). A computer program that learns algebraic procedures by examining examples and working problems in a textbook. *Proceedings of the 2nd National Conference of the Canadian Society for Computational Studies of Intelligence*, pp191-195.
- [Sle 81a] D.H. Sleeman & M.J. Smith, (1981). Modelling student's Problem Solving. *Artificial Intelligence*, pp 171-187.
- [Sle 81b] D.H. Sleeman. (1981) A Rule-based Task Generation System. In *IJCAI Proceedings*, pp 882-887.
- [Sle 82a] D. Sleeman & J.S. Brown, (1982) Editorial in *Intelligent Tutoring Systems*, edited by D.H. Sleeman and J.S. Brown, Academic Press, pp 1-12.
- [Sle 82b] D. Sleeman, (1982) Assessing competence in Basic Algebra. In *Intelligent Tutoring Systems*, edited by D. Sleeman and J.S. Brown, Academic Press, pp 186-199.
- [Sle 83] D.H. Sleeman, (1983). A rule directed Modelling system. In *Machine Learning*, R. Michalski, J. Carbonell, and T. M. Mitchell, (editors), Tioga Press, pp483-510.
- [Sle ap] D.H. Sleeman. (in press) Basic Algebra revisited: a study with 14 year olds. *Journal of Mathematical Behaviour*. (Also published as Stanford HPP memo 83-9).

FIGURE 1
a) RULES for the ALGEBRA domain
(evaluative form and slightly stylized).

RULE NAME	CONDITION/ACTION
FIN2	(SHD X = M/N) (SHD (M N)) or (SHD evaluated)
SIMPLIFY	(SHD X = M/N) (SHD X = M'/N')
SOLVE	(SHD M * X = N) (SHD X=N/M) or (SHD INFINITY)
ADDSUB	(lhs M + - M rhs) (lhs [evaluated] rhs)
MULT	(lhs M * N rhs) (lhs [evaluated] rhs)
XADDSUB	(lhs M*X + - M*X rhs) (lhs (M + - M) * X rhs)
NTORNS	(lhs + - M = rhs) (lhs = rhs - + M)
REARRANGE	(lhs + - M + - M*X rhs) (lhs + - M*X + - M rhs)
XTOLHS	(lhs = + - M*X rhs) (lhs - + M*X = rhs)
BRA1	(lhs < N > rhs) (lhs N rhs)
BRA2	(lhs M<N*X + - P> rhs) (lhs M*N*X + - M*P rhs)

where M, N and P are integers and where lhs, rhs etc are general patterns (which may be null), where +|- means either + or - may occur, where SHD indicates the String Head and where < and > represent standard "algebraic brackets". (NB the respective Rule CONDITIONS and ACTIONS are on alternate lines).

b) Some MAL-RULES for the Domain

RULE NAME	CONDITION/ACTION
MSOLVE	(SHD M*X = N) (SHD X=M/N) or (SHD INFINITY)
MNTORHS	(lhs + - M = rhs) (lhs = rhs + - M)
M2NTORHS	(lhs1 + - M lhs2 = rhs) (lhs1 + - lhs2 = rhs + - M)
M3NTORHS	(lhs1 + - M lhs2 = rhs) (lhs1 + - lhs2 = rhs + - M)
MXTOLHS	(lhs = + - M*X rhs) (lhs + - M*X = rhs)
M1BRA2	(lhs M *<M*X + - P> rhs) (lhs M*N*X + - P rhs)
M2BRA2	(lhs M*<M*X + - P> rhs) (lhs M*N*X + - M + - P rhs)

Using the same conventions as above.

FIGURE 2

Three examples of very consistently used MAL-RULES.

I) Pupil AB-11 on task set 7.

a) The task given was: $4 + 2*X = 16$

Pupil writes 1) $6X = 16$
 2) $X = 2.6666$

b) The task given was: $2 + 4*X = 14$

Pupil writes 1) $6*X = 14$
 2) $X = 2.333$

c) The task given was: $3 + 5*X = 11$

Pupil writes 1) $8*X = 11$
(and is told to leave it in that form)

d) The task given was: $6 - 3*X = 11$

Pupil writes 1) $2*X = 11$
(and is told to leave it in that form)

II) Pupil AB-17 on task set 6

a) The task given was: $2*X + 4*X = 12$

Pupil writes 1) $* X * X = 12 - 2 - 4$
 2) $X ** 2 = 8$
 3) $X = \text{ROOT } 8$

b) The task given was: $2*X + 3*X = 10$

Pupil writes 1) $* X * X = 10 - 2 - 3$
 2) $X ** 2 = 5$
(and is told to leave it in that form)

c) The task given was: $2*X - 3*X = 10$

Pupil writes 1) $* X * X = 10 - 2 + 3$
 2) $X ** 2 = 11$
(and is told to leave it in that form)

III) Pupil AB-18 on task sets 5, 6, 7 and 8.
a) The task given was: $2*X + 3*X = 10$

Pupil writes 1) $2*X = 10 - 2 - 3$
 2) $2*X = 5$
 3) $X = 2.5$

Page 19

b) The task given was: $3*X + 5*X = 24$

Pupil writes 1) $X + X = 24 - 3 - 5$
 2) $2*X = 16$
 3) $X = 8$

c) The task given was: $3*X + 4*X + 5*X = 24$

Pupil writes 1) $X+X+X = 24-3-4-5$
 2) $3*X = 12$
 3) $X = 4$

d) The task given was: $2*X + 4 = 20$

Pupil writes 1) $X = 20 - 2 - 4$
 2) $X = 14$

e) The task given was: $3*X + 6 = 7$

Pupil writes 1) $X = 7 - 3 - 6$
 2) $X = -1$

f) The task given was: $4 + 3*X = 14$

Pupil writes 1) $X = 14 - 3 - 4$
 2) $X = 7$

g) The task given was: $6 + 6*X = 20$

Pupil writes 1) $X = 20 - 6 - 6$
 2) $X = 9$

h) The task given was: $4*X = 2*X + 8$

Pupil writes 1) $2*X = -4 + 2 + 8$
 2) $2*X = 4$
 3) $X = 2$

Pupil then wrote 1) $X - X = 2 + 8 - 4$
 2) $0 = 4$
 and QUILTS.

i) The task given was: $5*X = 3*X + 8$

Pupil writes 1) $0 = 4$
 and QUILTS.

FIGURE 3.

$3X + 5 = 8$	$3X + 5 = 8$
$\Rightarrow [P1]$	$\Rightarrow [P2]$
$X + 3 + 5 = 8$	$3X + 3 + 5 = 8$
$\Rightarrow [ADDSUB]$	$\Rightarrow [ADDSUB]$
$X + 8 = 8$	$3X + 8 = 8$
$\Rightarrow [NTORHS]$	$\Rightarrow [NTORHS]$
$X = 8 - 8$	$3X = 8 - 8$
$\Rightarrow [ADDSUB]$	$\Rightarrow [ADDSUB]$
$X = -2$	$3X = -2$

This figure gives the TRACES for 2 paths one involving the perturbation P1 ($MX \Rightarrow M + X$) and the other P2, ($MX \Rightarrow MX + M$). The implemented algorithm returns a list of all the TRACES which explained the pupil's response, say $X = -2$. Where $\Rightarrow$ implies that the "upper" form is transformed into the "lower".

FIGURE 4.

g) $3X + 6 = 8$
 $\Rightarrow$ [Inference step]
 f) $X + 3 + 6 = 8$
 $\Leftarrow$ [Instantiate to LHS of target eqn]
 e) $X + [6-n] + [n] = 8$
 $\Leftarrow$ [HEURSUM2]
 d) $X + 6 = 8$
 $\Leftarrow$ B[NTORHS]
 c) $X = 8 - 6$
 $\Leftarrow$ [Instantiate to RHS of Target equation].
 b) $X = [-2-n] + [n]$
 $\Leftarrow$ [HEURSUM1]
 a) $X = -2$

Shows a scheme for inferring the Mal-rule $MX \Rightarrow M + X$ given the task $3X + 6 = 8$ and the pupil's response $X = -2$. Where B[R] implies use of the rule, R, in the Backwards direction, where $\Rightarrow$ is as in Figure 3 and $\Leftarrow$ implies that the "lower" form is transformed to the "upper" one.

FIGURE 5.

```

HEURSUM1
  IF lhs(eqn) = lhs(target)
    AND rhs(eqn) = rhs(target)
    AND rhs = 1
  THEN replace(i, [i-n] + [n])

HEURSUM2
  IF lhs(eqn) = lhs(target)
    AND rhs(eqn) = rhs(target)
    AND contains(lhs, i)
  THEN replace(i, [i-n] + [n])

HEURMULT1
  IF lhs(eqn) = lhs(target)
    AND rhs(eqn) = rhs(target)
    AND rhs = 1
  THEN replace(i, [i*n] / [n])

```

Shows a selection of the heuristics which have been implemented; the first 2 were used in the protocol given in Figure 4. All the heuristics, including the more complex instantiation rules have been implemented as Production Rules. (Here "target" is short for TARGET equation, eg $3X + 6 = 8$ in Figure 4, and "eqn" is short for the current equation, which in Figure 4 is initially, $X = -2$).

FIGURE 6.

d) $2X + 3 = 9$
 $\Rightarrow$ [infer rule]
 c) $2X = 9 - 3$
 $\Leftarrow$ [Instantiate to RHS of Target].
 b) $2X = [6-n] + [n]$
 $\Leftarrow$ [HEURSUM1]
 a) $2X = 6$

Showing the inference of
 $MX + N = P \Rightarrow MX = P - N$
 namely the NTORHS rule, given that the original rule-set contained only the SOLVE FIM2 and ADDSUB rules.

Domain-specific Learning and the Subset Principle

Robert C. Berwick

MIT Artificial Intelligence Laboratory, Room 820
545 Technology Sq., Cambridge, MA 02139

Abstract

A key issue for learning theory has been the relative importance of domain independent learning vs. domain specific learning. How much of learning is attributable to general learning methods like "inductive generalization" and how much is attributable to particular techniques and representations that apply only to specific domains? This paper explores this issue by analyzing the role of one general principle in the context of several very specific domains. Angluin [1980] established a necessary and sufficient condition for the acquisition of a (recursive) language (in the sense of Gold [1967]) from positive-only evidence. The effect of this condition is to impose an ordering on possible sequences of guesses about the target language combined with the array of possible data sequences in such a way that the acquisition system always guesses the narrowest possible language compatible with the data given so far. In this paper, this "subset principle" is applied to several areas where extensive domain-specific knowledge is present, including syntactic constructions and phonological distinctive feature systems. Several well-known constraints on these systems are shown to follow the subset principle, providing some evidence that natural learning systems are designed so as to be easily learnable.

Key Terms: Mathematical learning theory, concept acquisition, language acquisition.

I. INTRODUCTION

What is the scope and power of "general" principles of learning? Traditionally, general methods like "inductive generalization" or "analogy" have figured prominently in any discussion of learning. And those who have embraced such general learning principles have ranged across the scientific spectrum, from Skinner to Piaget to Newell and Simon. More recently though, the mere existence of general learning principles has come under serious question:

What I intend to signify in referring to the doctrine of uniformity of mind is...that there are general principles of learning that underlie all of these systems, accounting for their development; "multipurpose learning strategies," as they are called, that apply "across the board." In contrast, it might be proposed various "mental organs" develop in specific ways...and that multipurpose learning strategies are no more likely to exist than general principles of "growth of organs" that

account for the shape, structure, and function of the kidney, the liver, the heart, the visual system, and so forth.
Chomsky [1980] p. 245.

The challenge for the learning theorist is to come up with some general principle that actually counters Chomsky's view. It is harder to do this than one might think; the remainder of this paper is an attempt to discuss one leading contender for a true general principle of learning -- the Subset Principle. In the next section this principle is defined and illustrated formally. As we will see, the effect of the Subset Principle is to impose an ordering on the way that (positive only) evidence is presented to a learner. This restriction shows up in a surprising number of domain independent and dependent contexts; these are taken up in Section III. Among the implications of the principle are: the difficulty of acquisition of disjunctive concepts; Keil's [1979] "M constraint" on predicability trees (roughly, natural semantic networks); the design of human phonological systems and their acquisition; and the existence of otherwise mysterious constraints on natural language syntactic systems. The good news, then, is that there seems to be at least one principle of learning that applies "across the board"; the bad news is that it is a necessary and sufficient condition for acquisition from positive-only evidence -- so it is no surprise that it is obeyed.

II. THE SUBSET PRINCIPLE

We will approach the formulation of the Subset Principle in a somewhat round-about way. Suppose we try to see what blocks simple, incremental, learning procedures from working. The thorn in the side of many learning procedures is over-generalization; the Subset Principle removes that thorn. More particularly, a major and long recognized problem with the use of disjunctions in what is learned is that the danger of over-generalization becomes more acute. By definition, the conjuncts of a disjunctive description have no features in common (save for some very general root predicate). This fact is the bane of simple generalization procedures. For if we attempt to establish one half of the disjunctive clause by observing a positive example, and the other half by another example, then a simple-minded generalizer will often conclude that the correct description of the examples is the intersection of these individual descriptions, namely, the null set. For example, if the correct description of an object is $P \vee Q$, where P and Q are themselves conjunctive expressions, then one positive example can be P , and another Q ; the intersection of the two can be empty.

There are two basic solutions to the problem of disjunction. One is to admit negative evidence, so that one can know if a description is too general. That is the function of the G boundary set in the version space model of acquisition: it delimits the outer boundary of possible descriptions compatible with examples seen so

far by describing the properties of negative examples. But what if negative examples are not available? The only other possibility is to eliminate disjunctive statements.¹

In the version space framework, the non-identifiability of disjunctive languages is revealed by the structure of the lattice representing possible identification hypotheses. Since we are using only positive evidence, we cannot move the boundary set G downwards; only the lower bound can be moved upwards. No matter what possible hypothesis is made at any step i , the acquisition procedure can be forced to make a more refined guess at some later step, moving it to a broader class of languages, simply by adding an example that was not in the language guessed so far. But since the upper bound G cannot be moved, we can only approach G by moving the boundary set S upwards, one step at a time. Each move represents a new guess; since there are a countably infinite number of moves between S and G initially, the acquisition procedure must make an infinite number of guesses, violating identification in the limit. The crux of the problem is that the space of hypotheses is "dense" in the sense that there are collections of positive evidence compatible with an infinite number of possible guesses. Note that this is the reason that the example family of languages provided by Gold [1967], the family of all languages of finite cardinality and one language of infinite cardinality, fails to be identifiable from positive-only evidence: there is a finite collection of evidence sentences that is covered by an infinite number of languages in the family.

1. How much disjunction is too much? The following example shows that if "arbitrary" disjunction is allowed then there exists an infinite collection of languages whose individual members cannot be identified from positive-only evidence.

Example. (Angluin, [1980])

Suppose that a family of languages $\mathcal{L}$ includes the "universal" language Σ^* . In addition, suppose that $\mathcal{L}$ contains languages that are defined by rule systems that can express arbitrary disjunctions. That is, suppose that grammars for these languages can be of the form $A \rightarrow \alpha[\beta_1 \dots \beta_n]$, with no restrictions on the right-hand sides α, β (the rewriting system is still finite in size, of course). The intent is that each part of the right-hand side of the grammar covers one sentence in L_i . In other words, we assume the power to form the language of any member of this family as the union of individual sentences of that language. Let us call these languages the *disjunctive covering languages* of $\mathcal{L}$.

From the standpoint of acquisition, the problem with this family of languages is that any finite collection of samples from a language in $\mathcal{L}$ is covered by two languages: the universal language and the language generated by the grammar, $S \rightarrow L_1(s_1) \vee L_2(s_2) \vee \dots \vee L_n(s_n)$. But then there is a sequence of positive examples drawn from a member of $\mathcal{L}$ that will cause an acquisition procedure to change its guess about the identification of L_i an infinite number of times, and thus the procedure will fail to identify the members of $\mathcal{L}$ in the limit. The sequence is simply members drawn from the universal language. At each step i , an acquisition procedure must guess that examples it has seen have been drawn from either the universal language or the disjunctive covering languages of the examples it has seen. Suppose the procedure guesses the universal language at point i ; then no examples it sees later will force it to change its guess, since no example will contradict the guess it has just made. There is some disjunctive cover language whose examples are just like those of the sequence seen up to point i : simply conjoin the examples seen so far, disjunctively; call this language DL_i . But then the procedure must fail to identify DL_i , since there is the sequence of data consisting of the sequence that forced the procedure to guess the universal language followed by examples drawn just from DL_i on which the procedure does not identify DL_i in the limit. Here we make use of disjunctive power. Conversely, if the acquisition procedure guesses DL_i at point i , one can simply add an example that is in $UL - DL_i$; here we make use of the density of UL . Since we have assumed a countably infinite number of DL_i , the procedure can be made to change its guess an infinite number of times. (This example is a generalization of Gold's [1967] result that any family containing a all the finite languages and one infinite language is not identifiable in the limit from positive-only evidence.)

In short, there are two general situations in which an incremental, conservative approach to acquisition will work:

- (1) If positive and negative examples are available, then a partial ordering of hypotheses such that the accumulation points of the ordering are themselves hypotheses suffices. (This is the version space approach.)
- (2) If just positive examples are available, then a sparse covering of evidence sets suffices. That is, only a finite number of hypothesizable languages L_i can cover any possible evidence set. If general disjunctive statements are allowed -- then this condition cannot, in general, be met. (It might be that some other condition could suffice, however.)

The crucial condition required to ensure identifiability from positive evidence may be stated in another way. The problem with overly-dense families of languages is that, no matter what the positive evidence set, so that data set D_i prompts a guess of language L_i , it is always possible to find another language L_j that is compatible with D_i and can be interposed between D_i and L_i . But suppose that this never happens, that is, that an L_j can never be so interposed. Then either one of two situations must obtain, $L_i \supseteq L_j$, or L_i and L_j are incomparable, but both cover the data set D_i . In case 1, either L_j is identical to L_i -- hence cannot prompt a hypothesis change, under the Gold assumptions -- or $L_j - L_i$ is non-empty, and there is some d in L_j not in L_i . If L_j is the target language, then this example must eventually appear in the information presentation (again under the Gold assumption of a complete presentation), and an identification procedure will have evidence to change its guess. If L_i is the target language, then the procedure will never be forced to change its guess. In case 2, the same situation obtains. If either L_j or L_i is the correct target language, then some positive example must appear that indicates this possibility -- a sentence not in L_i , but in L_j , or a sentence not in L_j , but in L_i . (This is where the recursiveness of the L_i is used.) Therefore, in either case identification in the limit from positive evidence can proceed successfully. Let us say that when a family of languages meets this condition that it obeys the Subset Principle. The point of this condition is to ensure that the acquisition procedure always guesses a subset language if possible, that is, the smallest language that is also compatible with the positive evidence so far encountered. (Note that the sequence of guesses conforms to an incremental search from more specific to less specific languages if the lattice of languages is ordered by the subset relation. Again, what is crucial to the demonstration is the topological structure of the lattice of hypotheses. The particular details of what these hypotheses denote is irrelevant.)

The Subset Principle was independently observed and formalized in a recursive-function theoretic framework by Angluin [1980]. In an important theorem, Angluin demonstrated that the Subset Principle is necessary and sufficient for identifiability from positive evidence.

Given a family of recursive languages, $\mathcal{L}$. $\mathcal{L}$ is identifiable from positive-only evidence iff there exists an effective procedure which on input i enumerates finite sets $T_1, T_2, \dots$ such that

- (i) $T_i \subseteq L_i$
- (ii) For all $j > i$, if $T_j \subseteq L_j$, then L_j is not a proper subset of L_i .

The intent is for the T_i to be "trigger" sets for hypotheses, and the condition amounts to the claim that such finite triggers must exist for positive identification to work.

Since the principle is a necessary condition for positive-only acquisition, it is not surprising to discover that it subsumes a variety of proposals that have been advanced in the linguistic literature as constraints on the order in which acquisition hypotheses should be made. In fact, it appears as though the Subset Principle exhausts what can be said about ordering constraints in acquisition. To support this claim we will review several proposals that have been made regarding the ordering of hypotheses in acquisition, some in detail.

III. APPLICATIONS OF THE SUBSET PRINCIPLE

3.1 The acquisition of phonological segments

According to the distinctive feature theory of phonology originally developed by Prague school structuralists such as Jakobson and pursued by Chomsky and Halle in *Sound Pattern of English* [1968], all natural sound segments such as *a* or *p* can be described via a small number of binary-valued distinctive features. By and large these features have an articulatory or an acoustic grounding, with names suggestive of their place or manner of articulation, such as *high*, *back*, *anterior*, and the like, 20 or so features in all. Given binary values for distinctive features (+ or -), there are 2^{24} possible segments (about 16.8 million). However, most of these segments are not attested in human phonological systems. Furthermore, most collections of segments in a particular natural language do not make use of an extensive number of these possible contrasts.

As Kean [1974] shows, in part the reason for this is that distinctive features are not fixed in isolation. Rather, certain distinctive features can be fixed *only after* certain other features are fixed. For instance, according to Kean's theory the distinctive feature *consonantal* must be fixed before the feature *back* or *continuant*.

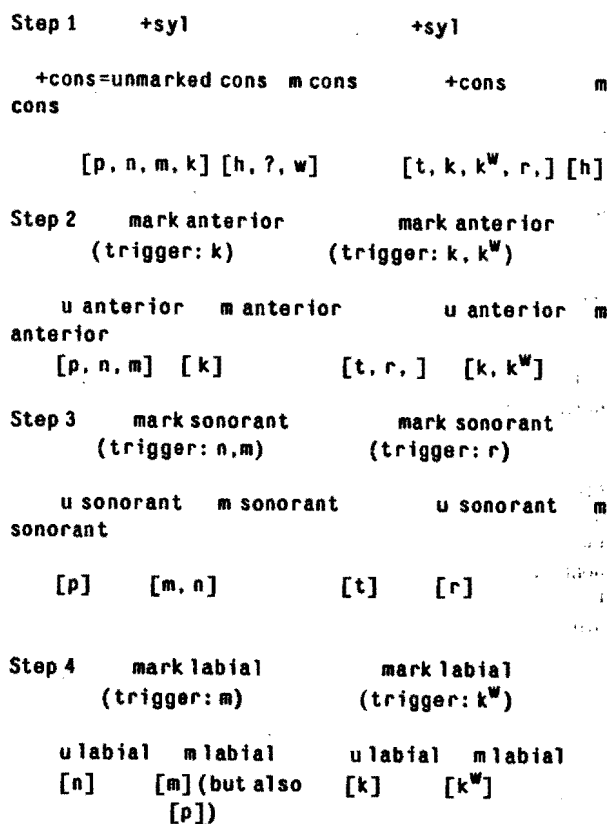
Kean developed this theory as way to explain some of the observed restrictions on possible segmental systems and possible phonological rule systems. But there is another way to interpret such a theory, and that is as a developmental program for how a segmental system is acquired. That is, we can identify the *markedness* of a particular rule, segment, or phonological system as the amount of information required to fix that system developmentally. This structure in turn permits a considerable compression of the information required to fix a particular segmental system. Importantly, the system obeys the Subset Principle. By construing the theory in this new way, one can in fact exhibit an acquisition system in which large numbers of developmental pathways are eliminated because of the *order* in which a small number of parameters are set. Moreover, the developmental model actually *explains* the existence of Kean's constraints, constraints that, while presumably universal, are

otherwise not accounted for. Put another way, taken as a model of acquisition the Kean theory provides an explicit example of how developmental sequencing can actually be an aid for acquisition, above and beyond the constraints provided by models of acquisition that assume an "instantaneous" fixing of a system of knowledge.

According to distinctive feature theory, segments can be distinguished only if they have different values for at least one of the 24 distinctive features. For example, the segments *a* and *i* are distinguishable given a segmental system that sets all distinctive features for the two segments to the same value save for the feature *back*. *a* is unmarked for back, while *i* is marked for back (i.e., is expected). To take another example, the segment *ae* is also marked for back (-back) but is distinguishable from *i* because it is additionally marked +low.

What Kean established is that there is a certain *logical* dependency order in which segmental distinctive features are fixed. For example, a segment must be first marked as consonantal or not; then, once the value for the feature *cons* has been set, value for the distinctive feature *anterior* may be fixed. Now, Kean deduced this order by examining the array of possible phonological systems in natural languages. More interestingly though, if we regard this order as a model of how a sound system is actually acquired, then we get a picture like this:

Hierarchy diagram of example class splits.



Different segmental systems correspond to different patterns of developmental paths. Note that at any given step where a class split occurs one must determine whether the class that is *not* prompting the split is marked or unmarked at step $i+1$. Thus [p] is marked labial at step 4 even though it doesn't prompt the split into marked/unmarked labial.

The important feature of the partitioning process is that splitting occurs at the leading edge of the directed hierarchy graph, by successive refinement of existing classes of segments. Because extension of classes occurs solely via the refinement of existing partitions, the set of segmental classes at step i will be a homomorphic image of all of the other trees before it in the developmental sequence. Also note that the tree is developed by marking just one distinctive feature at a time -- not a necessary constraint, since it is not clear why one could not develop a new class by marking two or more features in one step. The result of this constraint is to guarantee that at any step i in the development of a segmental system the classes of segments will be at most one mark (m) different. For instance, this constraint excludes the following array of marks:

	X1	X2	X3	X4
Feature:				
consonantal	m	u	u	u
anterior	m	u	u	u
back	m	u	u	u
low	u	m	u	u
labial	u	u	m	u
sonorant	u	u	m	u

From one point of view the one-mark constraint is a puzzling one. It is not at all obvious why segmental systems should be designed so that the alteration of a single distinctive feature could convert an a into an a^2 .

The key observation is that the one-mark constraint follows the Subset Principle -- and so then do natural segmental systems. If this observation is correct, then the one-mark constraint fits naturally into a model of incremental acquisition from positive-only evidence (rather than a model of error detection or correction). Note also that marking is always carried out at the immediate fringes of the existing class tree.

To see how important this is, suppose that acquisition is not incremental, so that two or more marks can be added at a single step. It would then be possible to form a new class partition based on marking both the features *labial* and *sonorant* without having first used the feature *sonorant* to form any segment classes. Now there is no class that will correctly accommodate a segment that is [u labial, m sonorant]. One way to remedy this problem would be to allow the procedure to go back and rebuild classes that have already been formed, but this would violate the developmental ordering that has been assumed. The fringe of the hierarchy tree would no longer summarize the possible next states that could be hypothesized, since there could be segments such as n that would demand the interpolation of new classes between older partitions and the current partition. In other words, the one- m constraint amounts to the demand that new classes be the minimally specific refinements of existing classes. This way of ordering hypotheses in a lattice of general to specific descriptions is, as we shall see immediately below, common to several proposed models of so-called "concept acquisition." It is also sufficient to guarantee that the incremental acquisition procedure proposed earlier will work correctly.

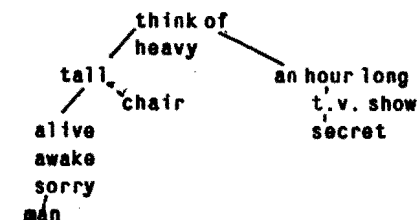
2. Note that this would seem to be an unwise design choice from the standpoint of error detection or error correction; as is well known, in order to be able to correct errors of k bits, then segments would have to be separated by a ball of radius $2k+1$ (since one must guarantee that changes of up to k bits in any two segments still leave one able to determine the original segment).

Put another way, the class of possible natural segmental systems seems to be more restricted than one might expect given a model where segments are obtained by selecting arrays of marked and unmarked distinctive features at random. Segmental systems are constrained in just the ways one might expect if they were acquired on the basis of an incremental fixing of segments according to the Kean feature hierarchy.

3.2 Concept acquisition and the M constraint

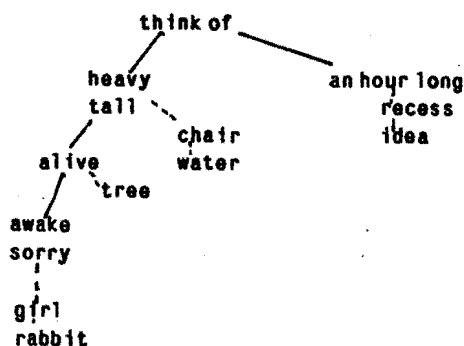
Related research on conceptual development as pursued by Sommers [1971], Keil [1979], and others also implicates the Subset Principle. Briefly, Sommers has suggested that if one arranges terms such as *dog*, *cat*, *love*, etc. into a graph structure whose terminal leaves are terms and where each node is an item that can sensibly be predicated of the all terms below it in the graph, then one never, or rarely, finds human intuitions of sensibility resulting in "M" shaped graphs -- the so-called "M" constraint. Rather, the graph structures take the form of hierarchical trees. For instance, the tree below (reproduced from Keil [1979, page 15, Figure 1]) is typical of one that comports with human intuitions; an "idea" can be predicated as being true, hence lies under a node labeled "true" in a predication tree, but cannot be at the corner, or red, hence does not lie under nodes labeled with these predicates. In contrast, a car can be nearby, at the corner, red, tall, and fixed, but not true.

Developmentally, Keil's key finding was that predicate hierarchies (predicability trees, as he called them) foliate via the refinement of existing classes of predicates, rather than by the radical reconstruction of predicate applicability relationships. This finding was supported by studies of the growth of predicability trees of young children. In other words, Keil found that predicability trees developed via the branching of the fringes of existing predicates -- the same process of refinement as was posited in the phonological model presented above. For instance, at the earliest ages studied (5-6 yrs.), some children's predicability trees looked like this:



(This is a single individual's tree.)

When second graders were tested, their trees were foliated versions of initial trees of this kind:



This developmental assumption is just like that of the phonological model. A new phonological partition is formed when it is realized that a segment is *not* like the segments of its classmates, and hence must be marked. (The hierarchy tree dictates *how* it must be marked.)

There are other similarities between this situation and the case of phonological systems. With rare exceptions, children were found to obey the M-constraint. This constraint is also honored by phonological rules. Further, the lattice of predication had an upper bound, an all-embracing predicate that could be applied to *any* object -- e.g., "think about". The corresponding predicate in the phonological domain is "syllabic" or, perhaps, "segmental".

Again, it is this Subset Principle that is being obeyed here. If refinement occurs at the "leading edge" of the hierarchy tree, then the new representation is a minimal refinement of the old one, in the narrowest possible way. This kind of conservative strategy is not necessary -- unless one is faced with the problem of acquisition based on just positive evidence. Thus the widespread existence of this strategy might be taken as an indication of a reliance on positive evidence here (as in the phonological case).

3.3 Some examples from syntax

The Subset Principle is also ubiquitous in the acquisition of the syntax of natural languages. A catalogue includes (but is not exhausted by) the following examples. In fact, as far as can be determined, the Subset Principle subsumes all the specific ordering principles advanced in current syntactic theory.

3.3.1 Bounding nodes for subjacency

In most current theories of generative grammar, one assumes that movement rules obey a certain "locality principle", in that a movement cannot cross more than a single clause boundary -- the Subjacency constraint.

John is certain [_S trace to like ice cream]

*John seems [_S it is certain [_S trace to like ice cream]]

The man [_{wh}_i [_S I don't know [_{wh}_j [_S trace_j knows trace_i]]]]

Interestingly, this last sentence is grammatical in Italian, as discussed by Rizzi [1978]:

L'uomo [_{wh}-phrase_i che [_{non so} [_{chi}_j [_{trace}_j conosca trace_i]]]]

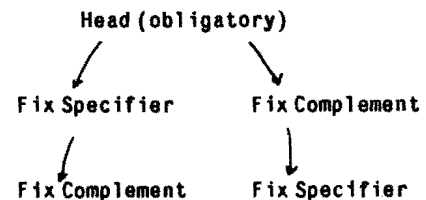
According to Rizzi, this is because it is *S*, not *S*, that is a bounding node for Subjacency in Italian. Therefore, the empty category *trace_j* can be co-indexed with the *wh*-phrase_i because it crosses only a single full clause boundary; the second boundary is an *S*, not an *S*. Apparently, the choice of a bounding node is yet another parameter that must be set in order to "fix" a grammar.

Suppose Rizzi's analysis is correct. How could the choice of bounding node be determined on the basis of evidence received by an acquisition procedure? Once again, let us apply the Subset Principle. If the bounding node for Subjacency is *S*, then a narrower class of languages is generated than if the bounding node for Subjacency is *S*. Therefore, by the Subset Principle, the acquisition procedure's first hypothesis should be to set the bounding node for Subjacency to *S*. In other words, the default assumption is that all languages are like English in this regard. If

this assumption is wrong, then a positive example will appear that violates *S*-bounding -- as in the Italian example above. Then the acquisition procedure can reset the Subjacency parameter to the next "largest" value, namely, *S*.

3.3.2 Acquisition of phrase structure rules

Williams [1981] describes a special case of the Subset Principle as a potential model for the acquisition of the context-free phrase structure rules that underly the basic sentence forms of a language. The idea is simple. According to current linguistic theory, a phrase structure rule obligatorily contains a *Head*, a unit of the same type as the phrase itself, and optionally has *Complements* and *Specifiers*. For example, in an English Verb Phrase the Verb itself is the ultimate (lexical) *Head*; the trailing NPs and PPs, and *S* are the *Complements*; perhaps *Adverbs* are *Specifiers*. Given this model, then the Subset Principle dictates a particular order in which hypotheses should be entertained. The first assumption is to assume that a phrase has no *Specifiers* or *Complements*, since this generates the narrowest possible class of languages. As usual, this initial assumption would be dropped in the face of positive evidence to the contrary -- in this case, the existence of a *Complement* or a *Specifier*. Thus the order of hypotheses would be as follows:



As Williams observes, there is no particular order in which *specifiers* or *complements* are fixed, relative to each other. Further, since either or both can be missing, the end state may be a phrase structure rule lacking in either *Specifiers* or *Complements*. More often, however, *Complements* are obligatory arguments; hence the general order of acquisition will be to fix *Complement* structure before fixing optional *Specifiers* (as also determined by the Subset Principle; see the discussion of obligatory arguments immediately below).

3.3.3 Optional and obligatory arguments and rules

More generally, it has been widely pointed out (see e.g., Baker [1979]) that it is difficult, if not impossible, to determine whether a rule is obligatory or optional on the basis of just positive evidence. For instance, Baker observes that Subject-Auxiliary inversion is obligatory in main clauses. If an acquisition procedure guessed a Subset language and assumed that inversion was optional, it could generate two forms, the second ungrammatical:

Who will John kiss?
Who John will kiss?

Suppose counterfactually that the inversion rule was optional in main clauses. This causes no problem; the acquisition procedure will simply acquire a new rule of analysis to handle that situation as well. In short, one can see that by formulating the acquisition

procedure as one of fixing new rules of analysis, one automatically orders hypotheses so that the default assumption is that a rule is obligatory until proven otherwise. This ordering satisfies the Subset Principle.

The same point holds for determining the argument structure of verbs, such as *walk*: by default, the learner must hypothesize the narrowest possible surface language. Thus if an argument appears with a verb, such as, *John walked to the store*, then the default must be to take the Prepositional phrase *to the store* as an obligatory argument until proven otherwise by explicit positive evidence (such as a sentence where the argument does not appear, as in, *John walked*). Optionality must always be fixed positive examples.

IV. ACKNOWLEDGMENTS

This paper describes work done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the Laboratory's artificial intelligence research is provided in part by the Office of Naval Research under Office of Naval Research contract N00014-80-C-0505. The author would also like to thank Patrick Winston and Noam Chomsky for discussions about various aspects of this work.

V. REFERENCES

- Angluin, D., "Inductive Inference of Formal Languages from Positive Data," Information and Control, vol. 45, pp.117-135, 1980.
- Baker, C.L., "Syntactic Theory and the Projection Problem," Linguistic Inquiry, vol. 10, pp. 533-581, 1979.
- Chomsky, N., Rules and Representations, Columbia, New York, 1980.
- Chomsky, N. and Halle, M., The Sound Pattern of English, Harper and Row, New York, 1968.
- Gold, E.M., "Language Identification in the Limit," Information and Control, vol. 16, pp. 447-474, 1967.
- Kean, M.L., The Theory of Markedness in Generative Grammar, PhD dissertation, MIT Department of Linguistics, 1974.
- Keil, F., Semantic and Conceptual Development: an Ontological Perspective, Harvard, Cambridge, MA, 1979.
- Rizzi, L., "A Restructuring Rules in Italian Syntax," Recent Transformational Studies in European Languages, S.J. Keyser (Ed.), MIT Press, Cambridge, MA, pp. 113-158, 1978.
- Sommers, F., "Structural Ontology," Philosophia, vol. 1, pp.79-85, 1971.
- Williams, E., "Language Acquisition, Markedness, and Phrase Structure," Language Acquisition and Linguistic Theory, S. Tavakolian (Ed.), MIT Press, Cambridge, MA, pp.8-34, 1981.

VALIDATING A THEORY OF HUMAN SKILL ACQUISITION

Kurt VanLehn
Xerox Parc
3333 Coyote Hill Road
Palo Alto, CA 94304

Extended Abstract

It is widely held that ordinary natural language conversations are governed by tacit conventions, called conversational postulates or felicity conditions. Learning a procedural skill is also a communication act. The teacher communicates a procedure to the student over the course of several lessons. The central idea of the theory to be presented is that *there are specific felicity conditions that govern learning*. Five felicity conditions on procedural skill acquisition have been discovered. They constrain how the teacher forms lessons and how the student learns from lessons. Even when lessons are incomplete, in that they do not contain enough information to communicate a procedure, students learn from them in ways determined by the felicity conditions. Thus, the theory can explain certain kinds of mis-learning as well as successful learning.

The theory employs a learning model. The model is a large AI-based computer program, named Sierra. It inputs a sequence of lessons consisting of worked examples and practice exercises. Sierra traverses the lesson sequence, learning a set of procedures (one procedure for each possible setting of certain parameters of individual difference). This set is very important. A key criterion for the theory is that the set of model-generated procedures should exactly match the set of procedures that students actually acquire, including their incorrect, "buggy" procedures. The model's predictions have been compared to data from students learning ordinary mathematical procedures: subtracting multi-digit numbers (1138 students), adding fractions (500 students) and solving simple algebraic equations (27 students).

However, there is more to validating this theory, or any complex AI-based theory, than just testing its predictions. The theory is stated as a set of 32 principles, of which the five felicity conditions are a subset. Sierra is only a tool, albeit an indispensable one, for calculating the predictions made by the principles. Explicating the principles is a necessary first step in validation. But many other questions remain to be addressed: What are the relationships between the principles and the behavior? Could the given cognition be simulated if the principles were violated or replaced by somewhat different ones? Would such a change produce inconsistency, or a plausible but as yet unobserved human behavior, or merely a minor perturbation in the predictions? Which alternatives if any can now be rejected in favor of the chosen principles? Answering these kinds of questions for each of the 32 principles has been the major work in validating the theory.

As an illustration of this kind of validation, consider the following felicity condition:

Learning a lesson introduces at most one new disjunction into a procedure.

In procedures, a disjunction may take many forms, e.g., a conditional branch (if-then-else). This felicity condition asserts that learners will only learn a conditional if each branch of the conditional is taught in a separate lesson—i.e.,

the then-part in one lesson, and the else-part in another.

The argument for the felicity condition hinges on an independently motivated principle: mathematical procedures are learned inductively. They are generalized from examples. There is an important philosophical-logical theorem concerning induction: If a generalization (a procedure, in this case) is allowed to have arbitrarily many disjunctions, then an inductive learner can identify which generalization it is being taught only if it is given *all possible examples*, both positive and negative. This is physically impossible in most interesting domains, including this one. If inductive learning is to bear even a remote resemblance to human learning, disjunctions must be constrained.

Two classic methods of constraining disjunctions are (i) to bar disjunctions from generalizations, and (ii) to bias the learner in favor of generalizations with the fewest disjunctions. The felicity condition is a third, new method. It uses extra input information, the lesson boundaries, to control disjunction. These three methods of controlling disjunctions are competing hypotheses for explaining how human learners control disjunction (along with several other hypotheses that won't be mentioned here).

The validation task is to discover and evaluate the entailments of each of the three hypotheses. In the case of the first hypothesis, it can be shown that barring disjunctions from procedures forces the theory to make absurd assumptions about the student's initial set of concepts—the primitive concepts from which procedures are built. Rejecting the first hypothesis leaves the fewest-disjunctions hypothesis and the one-disjunction-per-lesson hypothesis. The empirical predictions of these two hypotheses are identical, given the lesson sequences that occur in the data. More subtle arguments are needed to differentiate between them. Here are two.

(1) The felicity condition explains why lessons are so universally used in teaching and more importantly, why lesson sequences have the structure that they do. If the fewest-disjunctions hypothesis were true, then it would simply be an accident that lesson boundaries happened to fall exactly where disjunctions were being introduced. The one-disjunction-per-lesson hypothesis has more explanatory adequacy.

(2) The fewest-disjunctions hypothesis predicts that students would learn equally well if the example sequences were scrambled so that lesson boundaries fell at arbitrary points. This empirical prediction needs checking. If it is false, as I am sure it is, then the fewest-disjunctions hypothesis can be rejected on empirical as well as explanatory grounds.

This brief argument sketches the kind of individual support that each of the 32 principles has been given. Such competitive argumentation seems essential for demonstrating the psychological validity of a theory of this complexity.

