PROCEEDINGS OF THE INTERNATIONAL MACHINE LEARNING WORKSHOP

Edited by

R. S. Michalski

Allerton House, University of Illinois, Urbana, June 22-24, 1983.

ORIGINAL

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:

June 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

LEARNING STRATEGIES FOR PROBLEM SOLVING

LEARNING FROM OBSERVATION AND DISCOVERY

COGNITIVE MODELLING AND LANGUAGE LEARNING

AUTHOR INDEX

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 censors 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, censors 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, censors.

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 censors. 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, censors 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 censors can block censors and that censors 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 censors 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 censors. This implemented system inherits some key ingredients from previous work:

- **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.

- **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.

- **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.

- **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:

- 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.

- 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

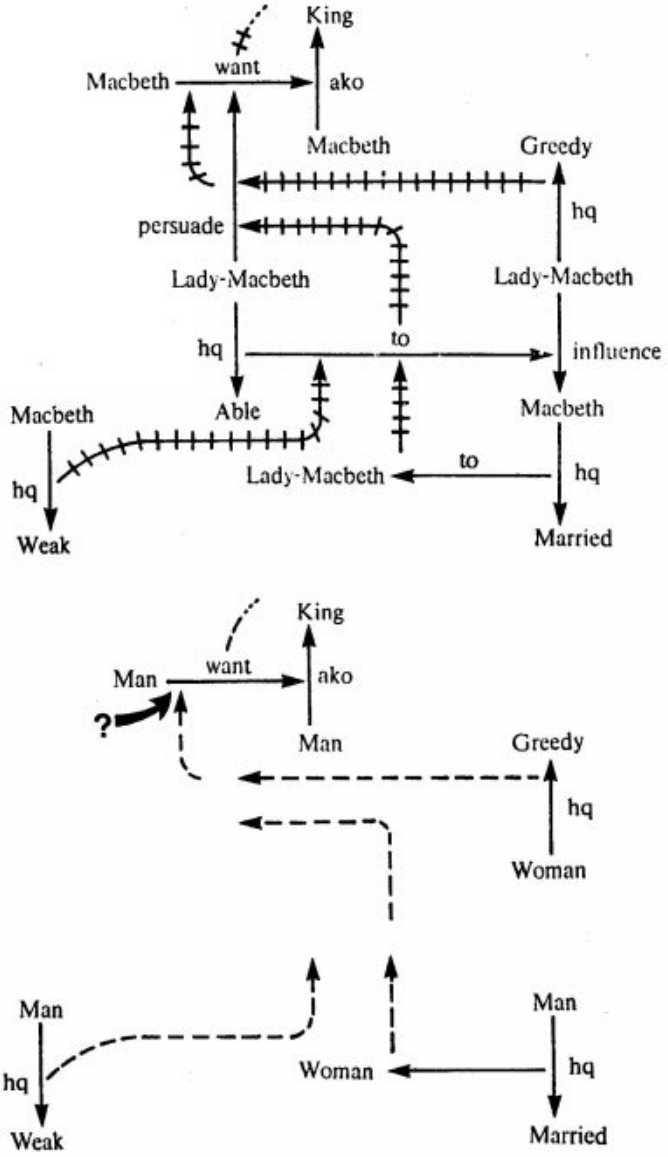


Figure 1: Problems are solved by transferring the existing cause relations of a precedent (crossed lines, part a) onto the problem to be solved (dotted lines, part b). HQ = Has Quality. AKO = A Kind Of.

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:

- The people in the precedent are matched with the people in the exercise. More generally, precedent parts are matched with exercise parts.
- The causal structure of the precedent is mapped onto the exercise.
- It is determined that the mapped causal structure ties the relation to be shown to relations known to be true.
- 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:

- 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:

- 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:

- First, the problem is posed and RULE-1 is fetched. Its *if* parts are satisfied.
- 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.
- 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.
- 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:

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

or alternatively,

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

where the *A*s are in the *if* part of the rule and the *B*s are in the *unless* part.

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

- 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:

- 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.
- The teacher complains when the system identifies a Catholic priest as a bachelor.
- 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.
- 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:

- In collaboration with Boris Katz: the problem of retrieving precedents from a data base so that they need not be given by a teacher.
- 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.
- 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:

  $$A_1 \wedge \ldots \wedge A_n \wedge \neg(B_1 \vee \ldots \vee B_n) \wedge (G_1 \vee \ldots \vee G_n) \Rightarrow C,$$

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

  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]:

- 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].
- There is no way to handle constraints about quantities such as those constraints that appear in the work of Forbus [1982].
- 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].
- There are no satisfying ideas about the role of abstraction in doing matching and indexing and retrieving.

- 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:

- The situations in the domain can be represented by the relations between the parts together with the classes and properties of those parts.
- The importance of a part of a description is determined by the constraints it participates in.
- 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:

- Actor-object representation.
- Importance-dominated matching.
- Analogy-based reasoning using constraint transfer.
- If-then rules learned by solving problems.
- If-then rules improved by modifications based on near misses.
- If-then rules augmented by unless parts.
- 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, additionals 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 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

[1] In 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!

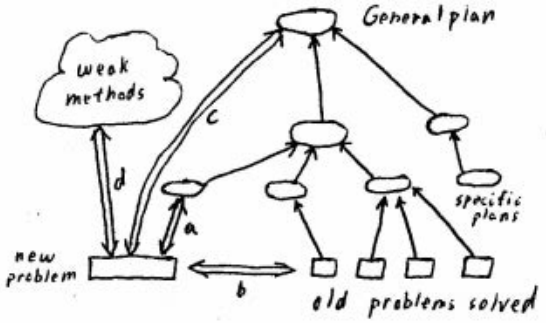


Figure 1-1: Problem solving may occur by a) instantiating specific plans, b) analogical transformation to a known solution of a similar problem, c) applying general plans to reduce the problem, d) applying weak methods to search heuristically for a possible solution, or e) a combination of these approaches.

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 programed 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 one 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 problem[2] (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].

---

[2] This 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 subgoaling 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 for 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 Specificaton 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 68, 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 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.

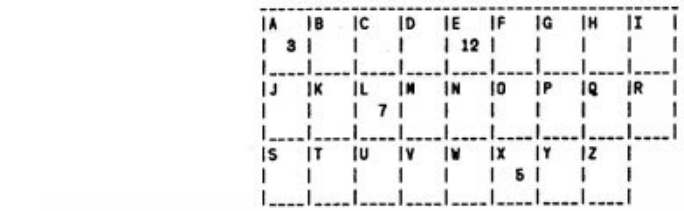
Another problem with theories based principly on the matching of descriptions, particularly as the complexity of these descriptions increases, is that conceptual representations for real situations may contain many objects not taking part in a specific analogy. Winston [Winston 80, Winston 82] suggested 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.

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

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.



> 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.

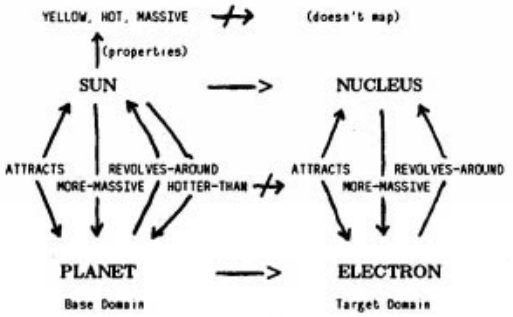
4. An Initial Structure Mapping Theory

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:
(*not*                            (*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 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.

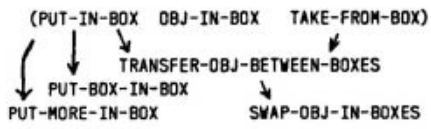
This behavior is modeled in CARL by retrieval and mapping of *related* causal structures from a base domain. CARL first uses the box analogy to support interpreting

"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.

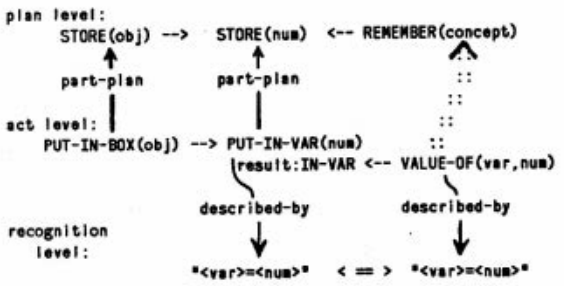
Once this association is formed, CARL can interpret the effects of many other assignment statements by parsing them as equalities, using algebra rules to determine effects on variables, and then mapping the statement interpretation and it's causal effects. The structures mapped are used both to replace 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

1. Albrecht, R., Finkel, L, and Brown, J.R.. *BASIC for Home Computers*. John Wiley & Sons, Inc., New York, NY., 1978.

2. 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.

3. 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.

4. 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.

5. 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 Ehrlbaum and Assoc., Hillsdale, NJ, 1981, pp. 285-309.

6. 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.

7. 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.

8. 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 Jersy, 1982.

9. 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

10. Lebowitz, M. *Generalization and Memory in an Integrated Understanding System* . Ph.D. Th., Yale University, October 1980.

11. Lehnert, W. G. Representing Physical Objects in Memory. Tech. Rept. 131, Yale University. Department of Computer Science , 1978.

12. 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.

13. Schank, R.C.. *Dynamic memory: A theory of learning in computers and people*. Cambridge University Press, 1982.

14. Schank, R.C. and Abelson, R.. *Scripts, Plans, Goals and Understanding*. Lawrence Erlbaum Associates, Hillsdale, New Jersy, 1977.

15. Winston, P. "Learning and reasoning by analogy." *CACM 23*, 12 (December 1980).

16. 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."

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

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 [Dershowitz-Manna77] 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

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. [DershowitzManna75] 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 ϵ. These requirements are conveniently expressed in the form of the following skeleton program:

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

The **achieve** statement,

achieve $|c/d-q|<\epsilon$ **varying** q,

contains the *output specification* which gives the relation between the variables q, c, d, and ϵ 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 ϵ. 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 ϵ, 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 $0\le c\le d$, $\epsilon>0$,

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, ε>0
    purpose |c/d−q|<ε
        purpose q≤c/d<q+s, s≤ε
            (q,s) := (0,1)
            loop L2: suggest q≤c/d<q+s
                  until s≤ε
                  purpose q≤c/d<q+s, 0<s<s_L2
                      if d·(q+s)≤c then q := q+s fi
                      s := s/2
                  repeat
        suggest q≤c/d<q+s, s≤ε
    E2: suggest |c/d−q|<ε
    end
```

The **purpose** statement,

purpose $|c/d-q|<\epsilon$,

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

suggest $|c/d-q|<\epsilon$

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

purpose $q\le c/d<q+s$, $s\le\epsilon$

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

at label L_2. The goal of the loop body is

purpose $q \le c/d < q+s,\ 0 < s < s_{L_2}$,

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 $s \le e$ 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 $d \cdot q \le c < d \cdot (q+2 \cdot s)$. This remains true when the loop exit is taken; along with the exit test $s \le e$, 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 $s \le e$ accordingly becomes $s \le e/2$. 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·s, 2·s≤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<s<s_L1
              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 |a^(1/3)-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

$$\begin{array}{ccc} q & \iff & r \\ c/d & \iff & a^{1/3}, \end{array}$$

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

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

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

$$\textbf{purpose } q \le c/d < q+2\cdot s,\ 2\cdot s \le \epsilon,$$

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

$$\textbf{purpose } r \le a^{1/3} < r+2\cdot s,\ 2\cdot s \le \epsilon,$$

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

$$\textbf{purpose } q \le c/d < q+2\cdot s,\ 0<s<s_{L_1}.$$

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

$$\textbf{purpose } r \le a^{1/3} < r+2\cdot s,\ 0<s<s_{L_3}.$$

At this point the division program introduces a conditional with the

$$\textbf{purpose } q \le c/d < q+s$$

and halves s.

It is here that the analogy breaks down. The division program achieves the above purpose in two cases, by testing if $d\cdot(q+s)\le c$ or not. For example, if $d\cdot(q+s)\le c$ does not hold, then $c/d<q+s$, as desired. On the other hand, the fact that $d\cdot(r+s)\le a$ 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

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

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 $r\le a^{1/3}<r+2\cdot s$ prior to entering the loop. To achieve the subgoal $r\le a^{1/3}$, 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, ε>0
   (r,s) := (0,(a+1)/2)
   loop L3: assert r≤a^(1/3)<r+2·s
         until s≤ε/2
         if (r+s)^3≤a then r := r+s fi
         s := s/2
         repeat
   E3: assert |a^(1/3)-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:

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

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:

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

Applying these transformations to the specification

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

of the division program yields

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

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

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

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

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

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 e>0, δ(w,u)≤v ≡ u≤γ(v,w)
   achieve δ(d,z)≤z<δ(d,z+2·s) varying z,s
   loop L4: assert δ(d,z)≤z<δ(d,z+2·s)
         until s≤e/2
         if δ(d,z+s)≤z then z := z+s fi
         s := s/2
         repeat
   E4: assert |γ(z,d)-z|<e
   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 $\delta(d,z)\leq z<\delta(d,z+2\cdot s)$ **varying** z,s

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∈N
   achieve z=⌊√a⌋ 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 $|\gamma(z,d)-z|<e$ **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 $z\leq\sqrt{a}<z+1$, $z\in Z$ **varying** z

(where 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 $z\leq\gamma(z,d)<z+e$,

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

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

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

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

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

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

```
R5: begin comment incomplete integer square-root program
   B5: assert a∈N
   achieve z²≤a<(z+2·s)² varying z,s
   loop   L5: assert z²≤a<(z+2·s)²
         until s≤1/2
         if (z+s)²≤a then z := z+s fi
         s := s/2
         repeat
   assert |√a-z|<1
   achieve z∈Z protecting z≤√a<z+1 varying z
   end
```

This program still contains two unachieved subgoals. The first can be achieved by assigning $(z,s) := (0,(s+1)/2)$. 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 $z\in Z$.) 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 $z\in N$ 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* **6**(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 <u>ab initio</u> 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.L. 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 Aptitude, 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 nondeterministic 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 Eleusis. 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 Dromey, 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

The authors gratefully acknowledge the partial support of the NSF under grant MCS-82-05166 and of the Office of Naval Research under grant No. N00014-82-K-0186.

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^q 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.

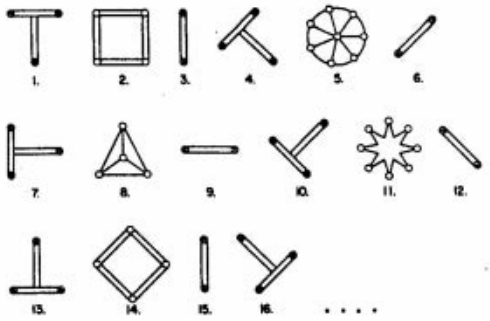


Figure 2-1: A simple NDP problem

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 expressions[1] A breadth-first search of this space, such as is conducted by the candidate-elimination

[1] This 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 Engelmore 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.

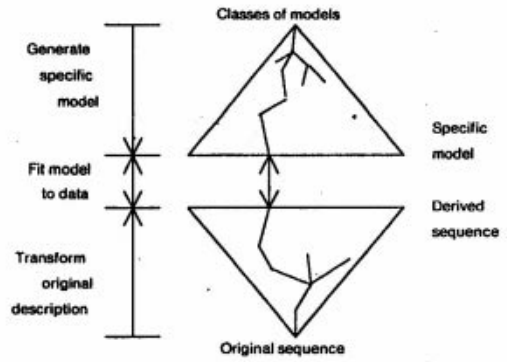


Figure 3-1: Schematic description of the rule discovery process

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 sequence[2]

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

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*) $f_1, f_2, \ldots, f_n$, 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

`[f1(qj)=r].`

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 SUIT 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 $(9*91*3*3*3*3*7)^3 * (9*300*300*3*3*3*3*7)^2 = 2.11221*10^{34}$. If there are four conjuncts in a rule, then we obtain $[2.11221*10^{34}]^4 = 1.99*10^{137}$.

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

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

```
[color(qj)=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*, $D(f_j)$, 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:

```
[f1(qj)=r1][f2(qj)=r2]...[fn(qj)=rn].
```

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

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

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

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

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'.

$$T_{p_1, \ldots, p_k} : \langle S, Q, F \rangle \dashrightarrow \langle S', Q', F' \rangle$$

where p_1, ..., 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 `[weight(qi-1)>10][weight(qi)≤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 `[color(qi)=color(qi-1)][weight(qi)>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 `[length(qi')=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 `[length(qi')=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 `[color(qi)=color(qi-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: $\langle ph_1, ph_2, ... \rangle$. 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 A0, A1, ..., AL are defined that are applicable to the objects in the derived sequence. $A0(b_i)$ has the same value as $A(q_i)$; $A1(b_i)$ has the same value as $A(q_{i-1})$; and so on until $AL(b_i)$, which has the same value as $A(q_{i-L})$. In other words, the original attributes are retained in 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.

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.

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(b1)=2] is true for all blocks b1.

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(bi)=H]`, are written as selectors that refer to objects in the original sequence, such as `[suit(qi-1)=H]`. Similarly, selectors such as `[d-rank01(bi)=3]` are written as `[rank(qi)=rank(qi-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, $\langle q_1, q_2, ..., q_k \rangle$, a non-empty set of *admissible next objects* Q_{k+1}:

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

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", $g(\langle ... 4C \rangle) = Q_{k+1}$ 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 $\langle q_1, q_2, ..., q_k \rangle$ 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 $\langle q_1, q_2, ..., q_k \rangle$.

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

$$q_1 \in g(\langle\rangle)$$
$$q_2 \in g(\langle q_1 \rangle)$$
$$\vdots$$
$$q_m \in g(\langle q_1, ..., q_{m-1} \rangle)$$

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, ..., xn) = r1 v r2 v ... v rm]`

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

`[rank(qi)=9 v 10 v J v Q v K]`

indicates that the rank of object q_i 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(qi)≥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 <, >, ≤, ≥, and ≠, in addition to the basic = operator.

Examples of typical selectors include:

`[rank(qi)≠rank(qi-1)]`
(paraphrase: the RANK of q_i is different from the RANK of q_{i-1})

`[suit(qi)=suit(qi-1)+1]`
(paraphrase: the SUIT increases by one from q_{i-1} to q_i)

`[rank(qi)+rank(qi-2)>10]`
(paraphrase: the sum of the RANKs of q_i and q_{i-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:

```
∀i ([color(qi)=color(qi-1)] ∨
    [rank(qi)=rank(qi-1)])
```

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

```
∀i (C1 ∨ C2 ∨ C3 ∨ ... ∨ Ck)
```

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:

```
L1 => R1
L2 => R2
   .
   .
   .
Lm => Rm
```

where the => 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,

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

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

```
∀i([color(qi-1)=black]=>[parity(qi)=odd] ∨
[color(qi-1)=red]  =>[parity(qi)=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 is 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

```
<[color(qi)=red], [rank(qi)≥rank(qi-1)]>
```

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 <2H 3C 10H 5S AD 6S 6H 6C>.

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

```
< [texture-of-nodes(qi)=solid black] &
  [shape(qi)=T-junction] &
  [orientation(qi)=orientation(qi-1)-45],

  [texture-of-nodes(qi)=clear] &
  < [number-of-nodes(qi)=4],
    [number-of-nodes(qi)=8] >,

  [texture-of-nodes(qi)=cross] &
  [shape(qi)=bar] &
  [orientation(qi)=orientation(qi-1)+45] >
```

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

```
< [color(qi)=red], [color(qi)=black] >
```

and the DNF rule

```
[rank(qi) ≥ rank(qi-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

$$L_1 \Rightarrow R_1 \,\&$$
$$L_2 \Rightarrow R_2 \,\&$$
$$\vdots$$
$$L_m \Rightarrow R_m$$

can be written as the DNF rule

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

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

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

can be expressed as a decomposition rule of the form

$$C_1 \Rightarrow C_2$$
$$C_2 \Rightarrow C_3$$
$$\vdots$$
$$C_{k-1} \Rightarrow C_k$$
$$C_k \Rightarrow C_1.$$

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

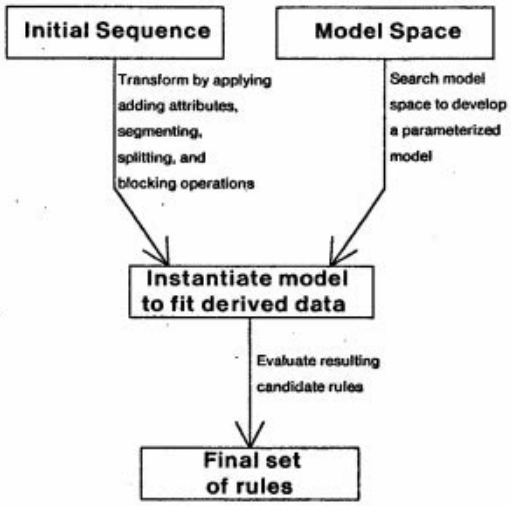


Figure 6-1: The Model-fitting Approach

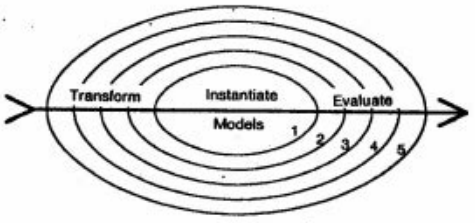


Figure 6-2: The knowledge ring architecture

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 RcCognition) 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
                6D      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.

```
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)=5][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.

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, $P(card_{i-1}, card_i)$ 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 $card_i$ to form the end of a segment. Typical segmentation predicates used are:

$$[rank(card_i) = rank(card_{i-1})]$$
$$[rank(card_i) = rank(card_{i-1}) + 1]$$
$$[color(card_i) = color(card_{i-1})]$$
$$[suit(card_i) = suit(card_{i-1})]$$
$$[parity(card_i) = parity(card_{i-1})]$$

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: `[suit(card1)=suit(card1+1)]`. The derived objects (segments) are denoted by variables $string_i$. The negative event `[suit(string2)=D][color(string2)=red][length(string2)=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  6D  2D  7C  AC  9C  JH  6H  8H  QH  KS
    5S  4D      AH
                7S

The derived sequence:

description of                          positive or
derived object                            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 $\langle q_1, q_2, ..., q_m \rangle$, let us look at block b_i which describes the subsequence $\langle q_{i-L}, ..., q_{i-1}, q_i \rangle$. Let F_j denote the selectors of object q_{i-j} renamed so that they apply to b_i. For example, F_1 could be the selectors **[suit1(bi)=H][rank1(bi)=3]**—selectors that originally referred to object q_{i-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(bi)=2][d-rank01(bi)=-3]** obtained from "subtracting" F_1 from F_0.

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

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

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:

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

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^9 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(card1-1)=black] => [parity(card1)=odd] V
[color(card1-1)=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 PARITY($card_{i-1}$) 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 **[parity(card1-1)=odd]**. There are four such events. The first selector in that conjunction, **[rank(card1)=9 v 4 v 2]**, is obtained by forming the internal disjunction of the values of **rank(card1)** 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)=15 v 12 v 18]
```

Figure 6-7: Trial decomposition on the PARITY($card_{i-1}$) 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, 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 PARITY($card_{i-1}$) and then on FACED($card_{i-1}$), 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(card1)=3 v 4 v 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  5S                  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(card1)≥jack]` implies `[face(card1)=true]`, and therefore, it did not remove the former selector. Similarly, because of the interaction of the two conditions, `[rank(card1)>rank(card1-1)]` 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**	**6D**	**8C**	**JS**	**2C**	**6S**	**AC**	**6S**	**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)≤6][SUIT(CARDI)=SUIT(CARDI-1)+1] V
[RANK(CARDI)≥6][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 Crysalis 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. L., "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," *IBM 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.

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).

<>	Angle brackets denote sequences of objects, e.g., <2 4 6 8> and also periodic rules, e.g., <[color(qi)=red],[color(qi)=black]>.
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 $\langle q_1, q_2, \ldots, q_k \rangle$.
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 v r2 v 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 u"adding" selectors F_i and F_j.
=>	Logical implication.