

LEARNING FROM NOISY DATA

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

ABSTRACT

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

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

INTRODUCTION

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

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

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

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

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

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

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

ALGORITHM FOR DEVELOPING CLASSIFICATION RULES

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

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

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

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

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

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

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

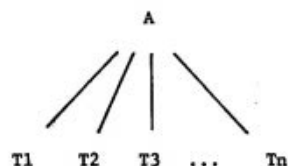
The presence of description errors complicates this situation somewhat. If the descriptions of the objects contain noise, it may be that the attributes are no longer adequate for the classification task. That is, we may end up with one or more training (sub)sets containing objects

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

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

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

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



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

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

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

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

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

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

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

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

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

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

THE PROBLEM

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

EXPERIMENT 1

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

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

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

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

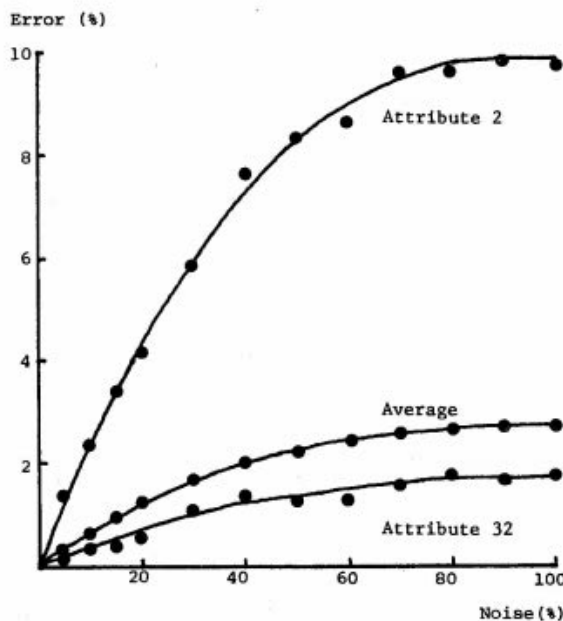


Figure 1. Classification error as a function of noise: attributes 2, 32, and average for all attributes (Expt. 1)

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

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

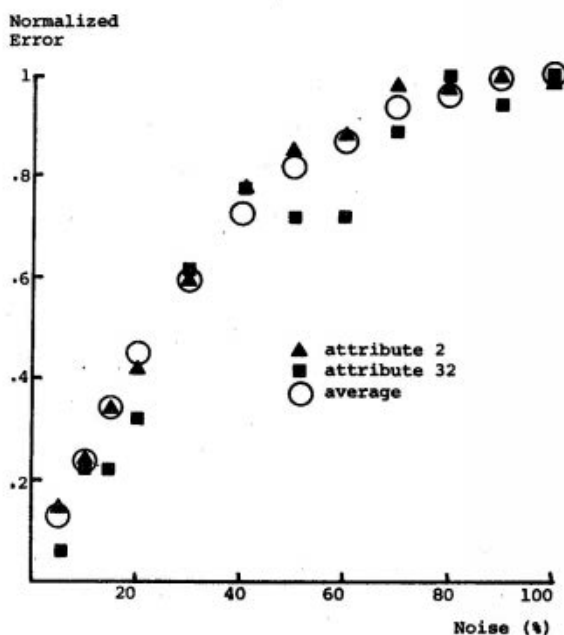


Figure 2. Normalized error for single-attribute noise (Expt. 1)

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

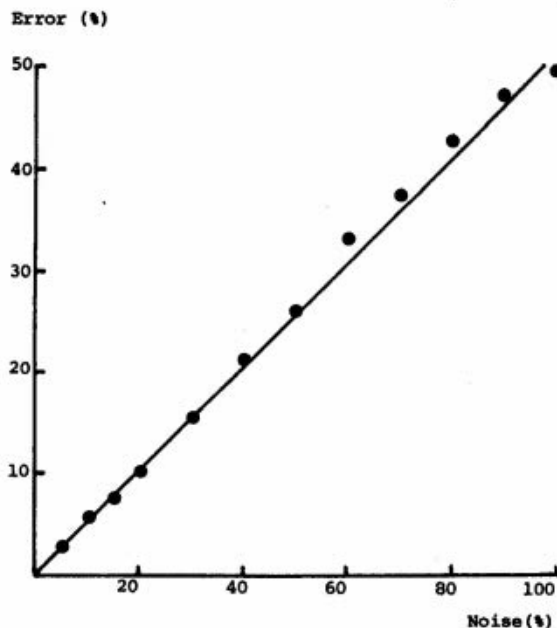


Figure 3. Classification error from noise in class information (Expt. 1)

linearly with noise in the class information.

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

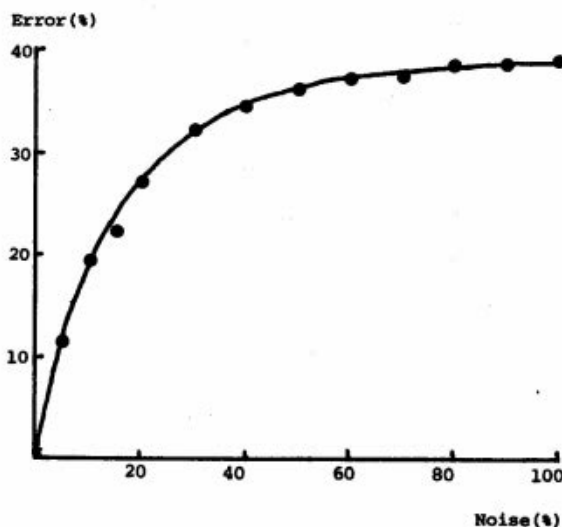


Figure 4. Classification error from noise in all attributes simultaneously (Expt. 1)

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

EXPERIMENT 2

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

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

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

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

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

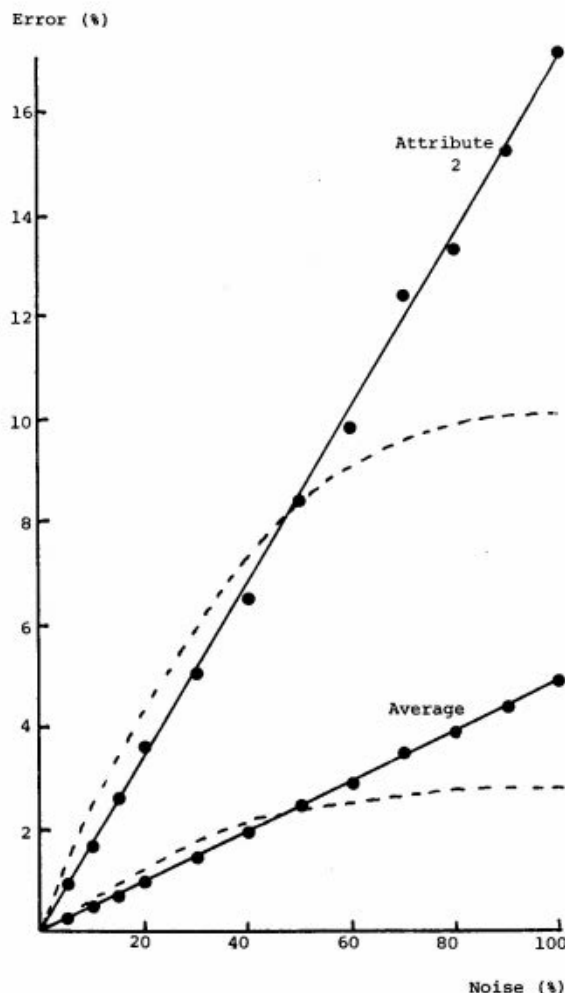


Figure 5. Classification error from single-attribute noise (Expt. 2)

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

EXPERIMENT 3

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

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

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

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

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

FINDINGS

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

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

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

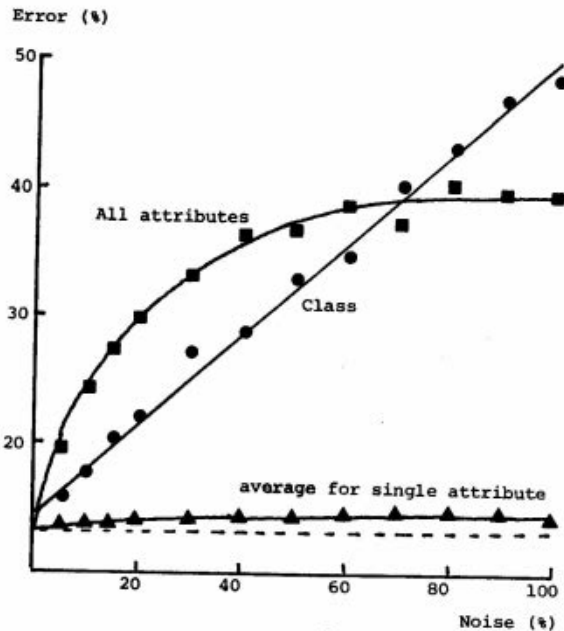


Figure 6. Classification errors (Expt. 3)

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

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

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

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

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

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

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

- (5) Noise is less of a problem for approximate classification rules.

A classification rule is approximate if it is simpler than the corresponding correct rule. This can come about either because it was formed from a training set containing only a small fraction of all objects or because there is a substantial amount of noise in one or more attributes. For example, the decision tree formed from 111 of the 551 objects contained 49 nodes, about the same as the tree formed from all objects when all attributes were corrupted with 50 percent noise. Figures 4 and 6 show that the marginal performance degradation with increasing noise in both cases is significantly less than the corresponding marginal rates for the correct decision tree. If either the necessity of producing a rule from a small training set or of using several inherently noisy attributes limits the classification accuracy of the rule, noise in the other attributes will not result in much additional degradation (but see 1 and 3).

CONCLUSION

The work reported here falls into three parts. First, a tree-building procedure has been developed that can cope with noisy data. Second, the procedure has been implemented (in C) and extensive trials carried out on a real classification problem with controlled addition of noise. Finally, as a consequence of the experimental results obtained in these trials, conjectures have been advanced about the use of noisy attributes and the expected benefits of reducing noise.

This study has been almost entirely empirical. The next step is obviously to try to understand, at a more abstract level, some of the observed phenomena, e.g. the different degradation curve shapes for noise in attribute and class information. It would also be interesting to find out whether similar results are obtained with alternative systems for constructing classification rules.

ACKNOWLEDGMENTS

I would like to thank John Potter of N.S.W.I.T. for several helpful discussions on the statistical aspects of the modified rule-building algorithm.

REFERENCES

- Hogg, R.V. and Craig, A.T. Introduction to Mathematical Statistics (3rd Edition), New York: Macmillan, 1970.
- Quinlan, J.R. "Learning efficient classification procedures and their application to chess endgames", in Machine Learning, Michalski, Carbonell and Mitchell (Eds.), Palo Alto: Tioga Press, 1983.

INDUCTIVE RULE GENERATION IN THE CONTEXT OF THE FIFTH GENERATION

Donald Michie
Machine Intelligence Research Unit
University of Edinburgh
Edinburgh EH8 9NW

ABSTRACT

The Japanese Fifth Generation Computer Systems Report (Moto-oka, 1982), setting out the explicit goal of dominating information technology world-wide, has brought home to countries further advanced in most of these skills than the Japanese how essentially fragile is their lead.

This paper advances some propositions which are in danger of being overlooked as we rush into action.

PART 1 : SEVEN PROPOSITIONS

It is convenient to list the following propositions concerning Intelligent Knowledge Based Systems (IKBS).

Proposition 1.

With the possible exception of image-processing, VLSI research is relevant to IKBS only in so far as VLSI design-automation offers to IKBS an interesting and important application area.

Proposition 2.

Novel architectures are usually conceived as departing in kind from the von Neumann scheme. But even quantitative shifts of emphasis (*e.g.* towards very large cheap memories) may introduce qualitatively new issues.

Proposition 3.

The distinction between languages and relational database systems is becoming artificial.

Proposition 4.

Language debates have so far ignored IKBS' most pressing requirement, namely a means of generating programs that are analysable and executable by machine and human brain alike.

Proposition 5.

For every feasibly computable function it is often assumed that a compact specification in some form of logical language is always what the program-writer (human or machine) really needs. But it is sometimes possible to generate an extensional specification which offers him/it an entire new way of proceeding.

Proposition 6.

Automating the construction of knowledge-bases is pre-requisite to the viability of any future IKBS industry. Feasibility of computer induction has been shown in more than one laboratory. Vigorous follow-up is indicated.

Proposition 7.

IKBS will find itself end-stopped unless machine-executable concept-description languages are implemented and made available, based on a new view of programming: the Principle of Symmetry.

We now venture a (limited) definition of IKBS:

"Intelligent knowledge-based systems are capable of improving their own knowledge bases *via* precept, example and discovery."

As a human counterpart for the IKBS we select the apprentice, - an abnormally articulate one, able to expound the master's teaching, once assimilated, better than the master can.

PART 2 : INDUCTIVE LEARNING

A number of computer programs have been developed which "learn" by generating rules from examples, *i.e.* by induction (Michalski and Chilausky, 1980; Quinlan, 1979, 1982; Shapiro and Niblett, 1982; Niblett and Paterson, 1982; and papers in Michalski, Carbonell and Mitchell, 1983).

As a result of a long-established and sustained interest in machine learning (see the *Com-*

puter Journal, Michie, 1982a) the Edinburgh group has become a practised "machine-induction shop", one of only two. The other is R.S. Michalski's group in the University of Illinois. We can now routinely synthesise machine-executable descriptions (in the form of PASCAL programs) for classification tasks too complex for human experts either to program, to verbalise complete strategies for, or even reliably to perform. According to the style adopted, the machine-generated descriptions can be made to take either "Martian" or human form. In the second case, in spite of its synthetic origin, we call a machine-made description a "concept". A recently developed programming technique (Michie and Shapiro, 1982) has added the following capability: when a machine-made concept is "run" on a trial case, the system not only classifies the case but also displays an explanatory narrative in intelligible English as to how it arrived at the classification. This is a help both to the end-user and to the knowledge-engineer himself.

A brief description of methods and principles used by one of the Edinburgh learning programs, ACLS, is given below. The program is a development from Quinlan's ID3, in turn derived from Hunt, Marin and Stone's (1966) CLS (Concept Learning System).

ACLS (Analog Concept Learning System)

A "training set" of examples of classifications within a chosen field is presented to ACLS as a list of records. All fields of the record except the last are entered with the values of those attributes considered by the user to be relevant to the classification task. These are known as "primitive attributes". The last field is entered with the name of the decision class to which the record is to be assigned (e.g. CIRCLE, ELLIPSE, TRIANGLE, SQUARE, POLYGON, OTHER; or MALIGNANT, BENIGN, IMAGINARY; or ALLOWED, DISALLOWED, SPECIAL-CASE; or just TRUE, FALSE).

From these example records, ACLS derives a classification rule in the form of a decision tree, branching according to the value of the attributes. ACLS also generates a PASCAL conditional expression logically equivalent to the decision tree, and this PASCAL code can be run to classify new test examples. Whenever a new example is found which refutes the current rule, ACLS can be asked to restructure the rule so as to accommodate the new case, and to display, output or store it as before.

The set of attributes should be chosen by an expert in the given problem domain to be sufficient for classifying the data. In choosing the set of attributes the maxim is: if in doubt, put it IN. Any attribute which is in reality redundant will be set on one side by ACLS and not included in the tree (or corresponding PASCAL code) which it generates.

IT IS NOT NECESSARY THAT THE EXPERT SHOULD BE ARMED WITH A CLEAR MENTAL PICTURE OF THE RULE WHICH HE HIMSELF USES WHEN PERFORMING THE GIVEN TASK.

His role is simply to supply the list of primit-

ives and to act as an oracle by assigning example data to what he considers to be their correct classes. ACLS observes his behaviour as a classifier, and from this constructs the simplest rule it can, using the primitives supplied, which will assign the same example data to the same classes as he does. As the training set grows with the addition of fresh examples, so the ACLS-synthesised rule grows more sophisticated. As it does so it tends to approximate to a form on which the expert is likely to comment: "That looks like the way I think I probably do it".

THE EXPERT IS THUS ENABLED TO TRANSFER TO THE MACHINE A JUDGMENTAL RULE WHICH HE ALREADY HAD IN HIS HEAD BUT HAD NOT EXPLICITLY FORMULATED.

We list below four closely related questions under investigation with the aid of ACLS and similar programs:

- (1) the possibility that a learning program, such as the one described above, can be used to extend the mental grasp of the human user;
- (2) the use of the machine-synthesised programs as instructional texts;
- (3) measurement of the "knowledge" (see later) contained in machine-synthesised programs;
- (4) development of a human-intelligible "concept description language" as a mandatory vehicle for output from induction systems.

For a number of reasons, chess is specially qualified as a test domain for experiments of this kind. However, in order to confirm the generality of observed results, after basic work has been done in this measurable "test-tube" world, ACLS-type programs can be used to investigate knowledge enhancement in the domain of high school algebra, robot vision, industrial engineering fault diagnosis, chemistry, etc. Here, properly controlled experiments are less possible, so that it is desirable that knowledge enhancement in the context of chess be looked at before use of this powerful, but as yet poorly understood, instrument spreads too far into social practice. In this connection the US Department of Justice's Federal Bureau of Investigation and the US Army's Construction Engineering Research Laboratory have already purchased and installed version-1 ACLS packages for the IBM PC. This version has no built-in constraints of the type envisaged in item (4) and hence could in theory be used as a generator of potentially dangerous software in the sense of Kopec and Michie (1983).

Knowledge measurement

Tests using an information theory approach to measure the knowledge-content of machine-produced ("synthetic") programs are under way in Illinois and Edinburgh (Michie, 1977, 1982b). Relevant theoretical work has also recently started at Stanford (Yao, 1982).

Use is made of machine-oriented definitions, such as the "knowledge-content" of a program, "computational advantage", "penetration", "grasp"

and rate of knowledge acquisition. "Difficulty" of a problem can also be expressed within the same numerical calculus.

The current objective is to build the measurement system referred to above into rule-synthesis programs. These programs will then be enabled not only to produce rules which can be executed by machine but at the same time will be able to give the user a numerically based evaluation of each rule as it is synthesised. Recent results have suggested the possibility that the very act of synthesising and displaying rules in this style may help a trained practitioner in the given domain greatly to extend his skill and grasp.

In the perspective of knowledge-measurement alternative categories of solution program are possible:

- (1) too calculation-intensive to be machine-feasible
- (2) too store-intensive to be machine-feasible
- (3) machine feasible: too calculation-intensive to be brain-feasible
- (4) machine-feasible: too store-intensive to be brain-feasible
- (5) machine-feasible and potentially brain-feasible.

Knowledge engineering is the art of writing programs to fall into category (5), or, more precisely, that sub-category of (5) which is not only potentially brain-feasible by reason of its store-time trade-off properties, but also satisfies certain necessary structural properties (see later). Only programs of category (5) are sufficiently memory-sparing to be understood and sufficiently processor-sparing to be hand checkable. They lie within the human window.

Can machine optimality be combined with brain-feasibility? This is determined in each case by the shape of the trade-off curve of the given function. No general guarantee can be given that the two goals can be combined in any particular case. Moreover, from the point of view of brain feasibility, there are other problems.

"Super-programs"

Recent results have shown that programs constructed by systems such as Quinlan's ID3 can be, in one sense, "super-programs" and at the same time quite incomprehensible to people. By a super-program we mean one that is at least twice as attractive in terms of execution cost as the best which could be written by a programmer.

The Figure shows the execution costs on a CYBER 175 for three different programs written to solve the same difficult problem. The first implements a standard algorithmic approach. The second was the fastest pattern-coded solution which J.R. Quinlan, author of ID3 and also an outstanding programmer, could achieve after many months of work. The third was a machine-generated program, produced

by ID3 itself, equipped with a file of primitive descriptors and a file of example data. The last clearly qualifies as a super-program (see figure). Further, however hard they try, chess experts cannot understand it. Even though it constitutes a complete and correct description, it does not qualify as a concept description.

Work triggered by this observation has shown the reason for this. It lies in the way the decision tree generated by ID3 (or the corresponding PASCAL expression) is structured. We have therefore developed a method known as "structured induction". This is a hierarchical approach in the sense that the rules, or concepts, generated by the program using the original primitive attributes, themselves become the primitives for the next pass through the data (Shapiro and Niblett, 1982). Shapiro has subsequently applied the approach successfully to the ending King and Pawn *versus* King and Rook (Pawn on a7), generating a complete and self-documenting classifier for a space of over 200,000 positions, using some 350 expert-supplied examples. Since human experts are not self-documenting, no codified classificatory theory of the domain previously existed. The classifier thus constitutes a substantial and original contribution to end-game theory.

The method described constrains the output of the inductive generator to take the form of decision structures which people can recognise as concepts. The constraints have been embodied in the syntax of a rule language CDL-1 (Concept Description Language - 1), for which a compiler is now operational on the VAX-750.

<u>Lost-3-ply experiments</u>	
(CDC Cyber 72 at Sydney University)	
49 attributes	
(mixture of	35 pattern based
	4 low-level predicate
	10 high-level predicate)
715 distinct instances	
177-node decision tree found in 34 seconds	
Classification method	CPU time (msec)
Minimax search	285.0
Specialised search	17.5
Using decision tree	3.4

Upper part of Figure relates to the inductive synthesis of a recogniser for the "lost-3-ply" property in the chess ending king and knight v. king and rook.

Lower part compares run-time performance of two hand-made and one machine-synthesised recogniser.

(from Quinlan, 1982)

PART 3 : A KNOWLEDGE REFINERY

Expert systems now have commercial promise, surprisingly in view of the limitations of method discussed in the main paper. In one or two cases they are even making money. A system of over 2,000 rules called R1 is used at Digital Equipment Corporation for configuring customers' needs (see McDermott, 1982). It already outperforms their best technical salesmen. But in knowledge refining, the focus is not on the product which expert systems were originally designed to deliver, namely interactive advice in conversation with a client, but on the unexpected by-product, the finished knowledge base itself. Expert system languages can be used:

- to get knowledge into the machine
- to test it
- to fill gaps
- to extend it
- to modify it.

Finally the knowledge can be put back into the human world in unrecognisably improved shape. This important observation was first made in the following manner:

Ivan Bratko, a Yugoslav computer scientist and chess master, was on a visit to Edinburgh. He chose for study an elementary piece of chess knowledge, namely, how to mate with king and rook against king, from any legal starting position. Although an elementary mate, it is not easy to program, taking (unless artificial intelligence methods are used) approximately 2-3 months of a graduate student's time. Furthermore, the resulting program is never completely correct in its play.

Using one of the Edinburgh advice languages, Bratko was able to complete the programming in two weeks. Subsequently he proved, both exhaustively by computation and also by symbolic proof, that the program was complete and correct. He then translated the program rules back into English. This produced six rules only: they were complete and correct, unlike grandmaster codifications which turn out to contain errors in those cases which have been tested by exhaustive computation (Clarke, 1977; Bratko, 1978; Kopec and Niblett, 1980), and moreover to consist mainly of gaps.

The improvement in the knowledge representation when back-translated from an advice program was unexpected. However, when the possibility was drawn to the attention of workers in other laboratories, they confirmed that they too were able to see indications of the same phenomenon. For example, MYCIN, developed in Feigenbaum's laboratory, was the first expert system of the modern type (Shortliffe, 1976). Although its knowledge is not broad enough to make the program of significant utility to clinicians, the system is heavily used by medical students, who find the knowledge-base more complete and easier to follow than a text book.

The following are some other cases where back-translation from an advice program has produced an improved text for the human learner.

Internal medicine

The internal medicine project is a collaboration between Dr. Jack Myers, the equivalent of a grandmaster in internal medicine, and a computer scientist, Harry Pople (Pople *et al.*, 1977). The aim is to codify Myers' clinical knowledge in the form of an expert system. Clinicians are already using print-outs of parts of the knowledge-base.

Organic chemical synthesis

In organic chemical synthesis at the University of California, organic chemists use the knowledge base from Wipke's program SECS (Wipke, 1977). In the domains covered by the program, they regard it as superior to conventional sources.

More striking cases occur where the system has inbuilt learning capabilities. When using these programs, the expert conveys his concepts to the system by means of examples and not through direct programming, just as he would if communicating with a student or colleague. The following two examples come into this category.

Soybean pathology

Until recently, the best classification of soybean diseases was that by the plant pathologist, Dr. Jacobsen, who collaborated in a study directed by R.S. Michalski in the University of Illinois (Michalski and Chilausky, 1980). Michalski's program, AQ11, is capable of generalising over examples: thus, in this case the machine was part-author of the resulting classificatory system, which gave 99% correct allocation of diseased plants as against approximately 83% for the old Jacobsen system. With great difficulty, using the machine to test, recycle and debug his theory, Jacobsen refined his own system so that it achieved a 93% success rate. Unable to improve it further, he then decided to accept the machine-generated taxonomy. He now uses this as reference material in his work.

Mass spectroscopy

This is the domain of the DENDRAL project at Stanford. The inductive learning module of the program, Meta-DENDRAL, produced rules (previously lacking) for the interpretation of mass spectra of the family of organic chemicals known as the mono- and poly-keto-androstanes. These rules formed the basis of a paper subsequently published in the chemical literature (Buchanan *et al.*, 1976).

The question arises: does a different program have to be written for each of these application areas? The knowledge put in by the expert must of course be domain-specific, but an induction program which assists his input of knowledge should typically show generality, as in the following example.

Lymphatic cancers

In the differential diagnosis of lymphatic cancers Ivan Bratko successfully used an Edinburgh

learning program derived from Ross Quinlan's ID3, which was in turn based on Earl Hunt's CLS (Concept Learning System) developed in the 1960's. The Edinburgh program was developed using the domain of chess. Strategies for chess end-games were conveyed by examples. The resulting correct and complete theory of King and Pawn against King is itself of interest to chess masters. Bratko then used the same program, in collaboration with a clinician, to construct a diagnostic scheme for the lymphatic cancers. In this experiment, with help only in extracting data from medical case histories, Bratko was able to perform as if he were a clinical expert, although knowing nothing about medicine (Bratko and Mulec, 1980).

Requirements for a refinery

What are the necessary ingredients for knowledge refining?

- (1) Knowledge-engineering software: *i.e.* "advice languages", "inference engines" (that part of an expert system able to make inferences from data supplied, and to retrace and display the lines of reasoning), and "induction modules" (that part of an expert system able to generate rules from examples).
- (2) A good development environment, *e.g.* UNIX or INTERLISP.
- (3) Trained knowledge engineers familiar with the above material.
- (4) One or more experts. Even with inductive knowledge generation automated to the limit of current technique, an expert is still needed to choose the repertoire of primitive measurements or attributes to evaluate as relevant to the given problem domain.

Originally designed to deliver interactive advice, it now appears that expert systems have a by-product which is even more important - the finished knowledge-base itself.

To recapitulate, expert systems can be used:

- to get knowledge into the machine
- to test it
- to debug it
- to fill gaps
- to extend it
- to modify it

and finally

- to put the knowledge back into human hands in improved form.

Thus, the possibility now exists of superseding (as the automobile has superseded the horse) a craft which has been going on for thousands of years - namely, the writing of manuals and texts on how to do things.

BIBLIOGRAPHY

Bratko, I. (1978). Proving correctness of strategies in the ALI assertional language. *Information Processing Letters*, 7, 223-230.

Bratko, I. and Mulec, P. (1980). An experiment in automatic learning of diagnostic rules. *Informatica*, 4, 18-25.

Buchanan, B.G., Smith, D.H., White, W.C., Gritter, R., Feigenbaum, E.A., Lederberg, J. and Djerassi, C. (1976). Applications of Artificial Intelligence for chemical inference, XXII, Automatic rule-formation in mass spectrometry by means of the Meta-DENDRAL program. *J. Amer. Chem. Soc.*, 98, 6168-6178.

Clarke, M.R.B. (1977). A quantitative study of King and Pawn against King. *Advances in Computer Chess 1* (ed. M.R.B. Clarke), pp. 108-18. Edinburgh: Edinburgh University Press.

Hunt, E.B., Marin, J. and Stone, . (1966). *Experiments in Induction*. New York: Academic Press.

Kopec, D. and Michie, D. (1983). *Mismatch between Machine Representations and Human Concepts: Dangers and Remedies*. Brussels: EEC Directorate-General for Science, Research and Development.

Kopec, D. and Niblett, T. (1980). How hard is the play of the King-Rook-King-Knight ending? *Advances in Computer Chess 3* (ed. M.R.B. Clarke), pp. 57-81.

McDermott, J. (1982). XSEL: A computer salesperson's assistant. *Machine Intelligence 10* (eds. J.E. Hayes, D. Michie and Y-H. Pao), pp. 325-337. Chichester: Horwood, and New York: Halsted Press.

Michalski, R.S., Carbonell, J. and Mitchell, T. (eds.) (1983). *Machine Learning: an Artificial Intelligence Approach*. Palo Alto: Tioga.

Michalski, R.S. and Chilausky, R.L. (1980). Knowledge acquisition by encoding expert rules versus computer induction from examples: a case study involving soybean pathology. *Int. J. for Man-Machine Studies*, 12, 63-87.

Michie, D. (1977). A theory of advice. *Machine Intelligence 8* (eds. E.W. Elcock and D. Michie), pp. 151-158. Chichester: Horwood, and New York: Halsted Press.

Michie, D. (1982a). Experiments on the mechanisation of game learning: 2. Rule-based learning and the human window. *Computer Journal*, 25, 105-113.

Michie, D. (1982b). Measuring the knowledge-content of expert programs. *Bulletin of the Institute of Mathematics and its Applications*, 18, 216-220.

Michie, D. and Shapiro, A. (1983). Research memorandum, in preparation.

- Moto-oka, T. (ed.) (1982). *Fifth Generation Computer Systems*. Proceedings of the International Conference on Fifth Generation Computer Systems, Tokyo, October 1981). Amsterdam, New York and Oxford: North Holland.
- Niblett, T.B. and Paterson, A. (1982). *ACLS MANUAL*. Edinburgh, UK, and Champaign, USA: Intelligent Terminals Ltd.
- Pople, H.E., Myers, J.D. and Miller, R.A. (1977). DIALOG: a model of diagnostic logic for internal medicine. *Proc. 5th Inter. Joint Conf. on Artif. Int.* (IJCAI-77). Pittsburgh: Computer Science Department, Carnegie-Mellon University. (This program was subsequently re-named INTERNIST. It is now called CADUCEUS.)
- Quinlan, J.R. (1979). Discovering rules by induction from large collections of examples. *Expert Systems in the Microelectronic Age* (ed. D. Michie), pp.33-46. London, Paris and New York: Gordon and Breach. See also Quinlan's chapter in Michalski, Carbonell and Mitchell (1983).
- Quinlan, J.R. (1982). Semi-autonomous acquisition of pattern-based knowledge. In *Introductory Readings in Expert Systems* (ed. D. Michie), pp.192-207. New York, London and Paris: Gordon and Breach.
- Shapiro, A. (1983). The role of inductive learning in expert systems. *Ph.D. Thesis*. Edinburgh: Machine Intelligence Research Unit.
- Shapiro, A. and Niblett, R.B. (1982). Automatic induction of classification rules for a chess end-game. *Advances in Computer Chess 3* (ed. M.R.B. Clarke), pp.73-92. Oxford: Pergamon.
- Shortliffe, E.H. (1976). *Computer-Based Medical Consultations: MYCIN*. New York: Elsevier, North Holland.
- Wipke, W.T. (1977). Computer-assisted 3-dimensional synthetic analysis. *Computer Representation and Manipulation of Chemical Information* (eds. W.T. Wipke, S.R. Heller and E. Hyde), pp. 147-174. London and New York: Wiley Interscience.
- Yao, A. (1982). Theory and applications of trapdoor functions. *Proc. 23rd Annual Symposium on Foundations of Computer Science* (Chicago), pp. 80-91. Los Angeles: IEEE.

Knowledge Acquisition and Learning in EXPERT

Casimir A. Kulikowski
Department of Computer Science
Rutgers University
New Brunswick, New Jersey 08903

ABSTRACT

Learning techniques are frequently de-emphasized in building expert systems because it is assumed that a human expert will provide the knowledge base. Yet, rarely are such knowledge bases complete or accurate, and much testing and refinement is needed before expert performance can be achieved. A data base of problem-solving examples with independently validated conclusions can be very useful to:

- a) induce reasoning rules using partial knowledge of causal and hierarchical relationships among data and hypotheses; and
- b) suggest improvements (generalizations or specializations) in the rules provided by the human expert. Two such systems have been developed to help construct and refine rules for the EXPERT consultation scheme. They have been successfully used to construct and validate knowledge bases in several medical domains.

SEEK: A System for Empirical Experimentation with Expert Knowledge

Expert knowledge for solving complex medical consultation problems changes frequently as new advances in clinical research appear, and as practical experience with the management of diseases is accumulated. Just as the human expert must keep up-to-date within a specialty, so must any of the expert systems that we develop. We have designed the SEEK system [1,2,3] to enable the builder of an expert consultation model to quickly and easily update and improve the performance of the reasoning model, based on results from medical cases with well-established diagnostic and therapeutic end-points.

SEEK prompts the model builder to enter the description of reasoning rules in the form of a table, where each conclusion can be defined in terms of classical medical terminology (major and minor findings, as well as necessary and exclusionary requirements). Each conclusion can have levels of confidence attached to it, such as definite, probable or possible, to account for the uncertainties and ambiguities in the reasoning. SEEK automatically converts the tabular description into

EXPERT rules, and runs the cases in the data base against the resulting model.

When disagreements between the expert's conclusions and those of the model are detected by SEEK, it suggests a set of experiments that can be carried out to modify the model's rules that have the potential of improving performance. These experiments usually consist of suggestions to generalize or specialize the logical scope of a rule or group of rules. Rather than carry out these experiments automatically, SEEK presents the model designer with a ranking of the experiments, and the designer must interact with the system to make the final choice. The experiments are generated by rule refinement heuristics which make use of the performance of the prototype reasoning model to instantiate specific advice on the modification of predicates or uncertainty levels in the rules. Experiments are typically tried out on an incremental basis, modifying one rule or a set of related rules at a time, and comparing the new with the previous performance. Because the overall performance of a model depends on the interaction of many rules, modification of a few of them will only change part of any faulty results, so various heuristics in SEEK have been developed to propose and evaluate the priority of the different specializations and generalizations based on expected improvements in performance.

Once an experiment has been selected and carried out, SEEK gives further advice on how to refine the rules. The designer will typically go through a sequence of experiments before settling on a final structure for the model. And even then, as new and more complex cases are incorporated within the data base, SEEK can be used repeatedly to see if they warrant any changes in the reasoning rules.

Because EXPERT models are compiled, we are able to run many experiments against quite substantial data bases (many hundreds of cases), keeping all in computer memory, which is needed to carry out the kind of global performance assessment that we are doing. In the past year, SEEK has been tested with a large knowledge base in the area of rheumatology developed by our

collaborators in rheumatology at the University of Missouri - Columbia (Drs. Gordon Sharp and Donald Lindberg). Heuristic rules that propose the experiments have shown their value in suggesting changes that resulted in significant diagnostic performance improvements for a large knowledge base (about 1000 observations, 400 hypotheses and 1000 rules). SEEK has also proved itself in helping build a large dermatology model.

Knowledge-based Induction of Rules

When the knowledge base designer lacks the time to construct a preliminary or prototype set of reasoning rules, it may be useful to try to directly induce them from the data bases of problem solving cases. Initially, it is typical for the human expert to specify the reasoning primitives (observations and hypotheses) and their interrelationships (mostly causal, taxonomic and instantiation forms). These relations can be used with a variety of learning heuristics to extract reasonable rules of inference from the data base of cases. We have taken this approach for models where the expert was able to provide us with such a structure; in glaucoma and rheumatology. Our learning program [4] uses a blackboard representation, with multiple knowledge sources to handle the different conclusions that the model is to arrive at, and the formation of the rules from the data that pertain to them.

Each knowledge source represents the a-priori knowledge of one physiological system or process and its mechanisms of dysfunction. It imposes a structure on the training data which implicitly defines permissible clusters of data and intermediate hypotheses. Competing interpretations can be proposed by having different knowledge sources view the data as manifestations of different physiological processes. A knowledge source has both declarative and procedural components. The declarative part is a static relational network of elements, each of which represents a particular state of dysfunction. They are: observations most closely attributable to a state; taxonomic links to subsystems; and causal rules linking one state to another. The procedural part consists of three functions: PROPOSE, EXPECT and GOAL. The PROPOSE function is a constrained generator of new candidate rules. It is invoked when the system chooses as its focus of attention an element which is part of the knowledge source network. It translates prior knowledge about all the permissible associations that include the focused element into pre-candidate inference rules, with the element as the consequent or right-hand side of the rules. It then proposes a

new candidate rule for each rule which is supported by pre-existing rules and which explains a portion of the present training cases. Pre-candidate rules with the same right-hand side element are generated in a general-to-specific order, so concise rules are found first whenever possible. The EXPECT function looks at new candidate rules and decides whether sufficient evidence exists for some of their elements to predict activity in other elements in the knowledge source. These are model-directed expectations flowing down taxonomic links or along causal links. The latter allow the system to re-focus attention to a different knowledge source, when the links connect one knowledge source to another. The GOAL function looks for ways of satisfying a causal rule so as to significantly strengthen one of the candidate rules supporting it. It is used as a mechanism for breaking deadlocks, when the system is unable to choose among competing graph structures of candidate rules.

The knowledge sources communicate through a global blackboard where the candidate rules are recorded. Each node on the blackboard is a rule with pointers to previously generated rules that are needed to support it. In addition, there are lists of positive and negative training instances which match the rule graph, and a numerical evaluation function.

The process by which the rules are learned can be viewed as a type of hypothesize-and-test procedure. The knowledge sources use the structure of prior causal and taxonomic knowledge to generate hypotheses about reasonable candidate rules, test these against the patterns of observations from the data base, and place those which are supported onto the blackboard. Once this has been done by all the knowledge sources, a support set of rules is produced to subsequently confirm or deny expectations of causal connectivity generated by the knowledge sources. Focus of attention heuristics activate the knowledge sources that contain causal consequences of an expectation, and these are tested against patterns in the data base and the blackboard. The result is an extended set of candidate rules, which is then pruned by various heuristics to produce a final covering set of rules for the training cases.

This system has been tested with subsets of the CASNET [5] glaucoma knowledge base and the rheumatology knowledge base [3]. We were able to generate rules that were comparable to those produced independently by the human experts, and in several instances produced more efficient rules for covering the cases in our data base.

In relation to a system such as SEEK, this approach represents a preprocessing or

alternative approach to developing a prototype model. We are now investigating the role of additional medical constraints on the strategies of rule generation.

Acknowledgements

The author wishes to thank Sholom Weiss, Peter Politakis and George Drastal for their collaboration and research within the Rutgers Research Resource on Computers in Biomedicine. This work was supported in part by grant RR-643 from the National Institutes of Health, Biotechnology Resources Program, Division of Research Resources.

References

- [1] Weiss, S. and Kulikowski, C., Data Base Aids for Designing Consultation Systems, Proceedings of the Third World Conference on Medical Informatics (MEDINFO '80), pp. 794-797. 1980.
- [2] Politakis, P. and Weiss, S.M., A System for Empirical Experimentation with Expert Knowledge, Proc. Fifteenth Hawaii International Conference on Systems Sciences, pp. 649-657. 1982
- [3] Politakis, P. and Weiss, S.M., Using Empirical Analysis to Refine Expert System Knowledge Bases, Journal of Artificial Intelligence (in press, 1983).
- [4] Drastal, G. and Kulikowski, C.A., Knowledge Based Acquisition of Rules for Medical Diagnosis, Journal of Medical Systems, 1982.
- [5] Weiss, S.M., Kulikowski, C.A., Amarel, S., and Safir, A., A Model-Based Method for Computer-Aided Medical Decision-Making, Artificial Intelligence, v.11, pp. 145-172, 1978.

HIERARCHICAL MEMORIES: AN AID TO CONCEPT LEARNING

Claude Sammut and Ranan Banerji

St. Joseph's University, Philadelphia, PA 19131

ABSTRACT

Marvin is a concept learning program which learns concepts by focussing on an example object for the concept and forms hypotheses, which it tests by showing objects to the trainer and asking him questions about their belonging to the concept being learned. Descriptions of concept are stored as sets of Horn clauses. The description of one concept may use the names of other concepts, so the interdependencies of concepts can be learned. A concept may consist of sets of objects, relationships between objects, functions which transform objects, or any other predicate.

Key Terms: Learning, machine learning, induction, artificial intelligence, predicate logic, automatic programming.

1. INTRODUCTION

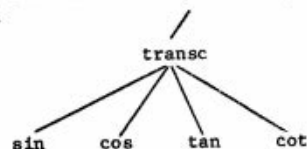
When we characterize learning as a search process, we assume the following:

1. There is a language which is used to describe concepts.
2. A state in the search space is a collection of descriptions written in this language. Given a set of examples, a goal state is a set of descriptions such that each positive example satisfies some description and no negative example satisfies any description.
3. There is a set of generalization rules for transforming states into new states.
4. A strategy for using the set of examples as a guide to choose a useful sequence of transformations which leads one to the goal state.

The problem of learning is to search through the space with a sequence of generalizations until a path to the goal is found. Learning systems such as LEX [Mitchell, 1982], INDUCE [Michalski, 1982] and MIS [Shapiro, 1981] have a fixed set of generalization rules. In programs such as CONFUCIUS [Cohen, 1978] and Marvin [Sammut, 1981], we have investigated learning systems in which the generalization rules change depending on the state, i.e. on the set of descriptions learned. Put another way, while in previous work the transformations operated on single sentences in the state, in our work the rules take the whole state into consideration. This makes for a

significant difference in the efficiency of the descriptions learned and, under many circumstances on the efficiency of the search process itself.

To contrast these two approaches, let us first look at a fixed language system such as LEX. This program is intended to learn how to perform symbolic integration. Expressions in LEX's language represent algebraic expressions. The system's generalization rules describe the way in which one expression may be transformed into a more general one. We reproduce below, part of the grammar for LEX's description language.



Note that the tree structure provides an ordering among expressions in the description language, indicating what kinds of expressions are more general than others. Thus, the designers of LEX have supplied it with a lot of background knowledge to enable it to learn to integrate. The tree not only provides the program with the simple concepts it needs (viz. real number, transcendental function, etc), it also shows the relationship between concepts. This relationship fixes the generalization rules once and for all. [Utgoff, 1983] explains how this "bias" increases the efficiency (indeed, the possibility) of learning. In our opinion this explanation is completely valid, unless the initial bias is unsuited for the descriptions one needs to learn.

Utgoff and Mitchell have pointed out this need for updating the bias and have shown how certain terms can be incorporated into the language and the partial order. However, there are further needs of language enrichment which can not be incorporated into the structure that they suggested. There is often a need for enriching the language by adding new disjunctions. The language is set up for generalizing the term $\sin(x)$ to $\text{transc}(x)$. It can be extended to include generalizations like $\sin(x)$ OR $\cos(x)$ without much trouble, say by calling the generalized term "subtrig(x)". However, terms like $\sin(x)\cos(x)$ can only be generalized to $\text{transc}(x)\cos(x)$, $\sin(x)\text{transc}(x)$ or $\text{transc}(x)\text{transc}(x)$. The tree structure prevents generalizations like $\sin(x)\cos(x)$ OR $\tan(x)\cot(x)$. The structure of the language has to be changed if one needs terms

like that.

Contrast with this the language used by Marvin, which allows the easy extension of the language by using sets of Horn clauses as descriptions, just as MIS does. The program is designed to emulate teacher/student learning. Unlike LEX, it is not given a hierarchy of concepts initially. The relationships between concepts are built up as the trainer presents new concepts. As in human teacher/student interactions, it is expected that simple concepts will be taught before complex ones. For example, addition is taught as a concept before multiplication is taught since the description of multiplication may use addition. The concepts which Marvin learns and the relationships between them form a model of the world in which the program exists.

Learning is not always aided by a benevolent teacher. Circumstances may force situations where the learning system tries to learn a more complex concept before learning a simpler concept upon which the complex one depends. Scientific investigation is an example of learning where the scientist (student) must deal with an indifferent trainer (nature). The history of science consists almost entirely of an account of how we have built models of the world, only to have to rebuild them when new concepts were later admitted, often with great reluctance, to be true. Shapiro's program, MIS, was an attempt to develop a system which could propose theories and, when necessary, trace through the derivation of a theory to discover an error if the theory was proved to be wrong.

We wish to investigate this kind of learning further. However, there are two difficulties with Shapiro's approach. Firstly, MIS uses a language determined entirely by the trainer. As we have already said, this limits the flexibility of the learning system. Secondly, MIS creates generalizations without taking advantage of the concept hierarchy to improve efficiency. Since Marvin was designed to avoid these difficulties, the new system is being built on the foundations already laid in this work.

However, before we discuss the problems that occur when we extend Marvin in this way, it may be worthwhile to discuss Marvin itself.

2. MARVIN

Marvin learns concepts by playing a game of "20 questions" with the trainer. The trainer shows the program an object which is an example of the concept which is to be learned (the target concept). Marvin uses this example to construct a hypothesis or trial concept by generalizing the description of the object so that it specifies a class of objects. A generalization is said to be consistent if the concept it describes is less general than the target.

To find out if a generalization is inconsistent, Marvin constructs object which are contained in the trial and shows them to the trainer. If the trainer tells Marvin that one of the objects is

not contained in the target then the generalization is inconsistent. When this occurs, Marvin tries to create a new trial which is more specific than the previous one. This is a restriction instead of a generalization. The program has been designed so as to minimize the number of objects which must be presented to the trainer in order to determine that a generalization is inconsistent.

When the program has learned a concept, it stores the description in its memory. A generalization is made when Marvin recognizes an object as belonging to a concept it has already learned. Marvin continues to make as many generalizations as it can. When it has exhausted all possibilities, the learning process stops and the final trial is stored in memory.

A concept is represented by a set of Horn clauses of the form:

P <- Q & R & S.

P, Q, R and S are predicates and the standard interpretation is applied: P is true if Q and R and S are true. The clause will be called a definition of the concept P. An alternative interpretation for a clause is: Q & R & S represents an instance of P, so Q & R & S may be generalized to P. Thus, a concept is also a generalization rule. For example, we may define a playing card as something having a suit and a face-value (called "value" below).

```
playing_card(X) <-
    suit(X, S) &
    any_suit(S) &
    value(X, V) &
    any_value(V).
```

"any_suit" may be defined as:

```
any_suit(X) <- atom(X, hearts).
any_suit(X) <- atom(X, diamonds).
any_suit(X) <- atom(X, clubs).
any_suit(X) <- atom(X, spades).
```

"atom(X, hearts)" means that "hearts" is an atomic value. "any_value" and other concepts such as "same_value" may be defined in the same way:

```
any_value(X) <- atom(X, king).
any_value(X) <- atom(X, queen).
. . . . .
any_value(X) <- atom(X, ace).
```

```
same_value(X, Y) <-
    suit(X, S) &
    suit(Y, S) &
    any_suit(S).
```

Marvin's memory is a database of such clauses. Predicates are also used to describe objects, for example,

```
suit(a, s) &
atom(s, hearts) &
value(a, v) &
atom(v, queen)
```

describes an object "a" which is a queen of hearts.

The predicates representing an object are called primary predicates. When an object is described to Marvin, the program searches its memory to determine if there is a subset of the primary predicates which matches the right hand side of a clause stored in memory.

In this example, "atom(s, hearts)" matches the right hand side of one clause in "any_suit" so we can infer that any_suit(s) is true. Similarly, any_value(v) can also be inferred. Since

any_value(v) & any_suit(s)

is true, then playing_card(a) is true.

Definition 1: Given a set of clauses:

$S^1 \leftarrow M^1$

$S^k \leftarrow M^k$

and a conjunction of predicates, C, we say that C implies C' if there is an Mⁱ and a substitution, σ , such that

$\sigma M \leftarrow C$ and $C' = C + \sigma\{S^i\}$

We treat atomic predicates in a conjunction as members of a set. "<=", "*", "+" and "-" mean subset, intersection, union and difference.

Definition 2: Given a conjunction, C⁰, we define a sequence C⁰...Cⁿ such that Cⁱ⁺¹ is implied by Cⁱ and Cⁿ does not imply anything. We write A(C⁰) = Cⁿ, representing the set of all predicates implied by C⁰.

If, in the original description of the object we replace "atom(s, hearts)" by the concept it implies i.e. "any_suit(s)", then we obtain the more general description:

suit(a, s) &
any_suit(s) &
value(a, v) &
atom(v, queen)

The description can be generalized further by replacing "atom(v, queen)" by any_value(v)".

The replacement operation can be described more formally as follow. We may think of any conjunction of predicates as a trial concept. In particular, the object shown as an example by the trainer is our first trial concept.

Definition 3: If Tⁱ is a trial concept and M is a subset of Tⁱ, and there exists a clause in memory S $\leftarrow$ M' such that with the substitution, σ , M = $\sigma M'$ then we define a replacement operation which will create a new trial, Tⁱ⁺¹, such that

$T^{i+1} = T^i - M + \sigma S$

Tⁱ⁺¹ is a generalization of Tⁱ. If there is more than one clause in the definition of the concept S the Tⁱ⁺¹ is more general than Tⁱ, since S describes a larger set than is described by M.

Suppose we are trying to teach Marvin the concept "pair":

pair(a, b) $\leftarrow$
playing_card(X) &
playing_card(Y) &
same_value(X, Y).

by giving it the example, T⁰:

pair(a, b) $\leftarrow$
value(a, v) &
atom(v, ace) &
suit(a, sa) &
atom(sa, spades) &
value(b, v) &
suit(b, sb) &
atom(sb, hearts).

Note that the values of a and b refer to the same object. The program may begin generalizing this description and eventually produce the following trial, T¹:

pair(a, b) $\leftarrow$
value(a, v) & (1)
suit(a, sa) & (2)
value(b, v) &
suit(b, sb) &
any_value(v) & (3)
any_suit(sa) & (4)
any_suit(sb).

which is a consistent generalization.

Definition 4: If T⁰...T^k is a sequence of trials such that Tⁱ generalizes to Tⁱ⁺¹ then we say T⁰ satisfies T^k.

Definition 5: Trial T is consistent with the target T' if any object which satisfies T also satisfies T'.

Since that values of cards "a" and "b" refer to the same object, equality of values has been preserved. Whenever two objects with the same description are entered, they are given the same name so that equality may be considered as part of the concept. By generalizing further, Marvin tries to remove this equality. For example, a new trial, Tⁱ⁺¹, produced from Tⁱ may be:

pair(a, b) $\leftarrow$
card(a) &
value(b, v) &
suit(b, sb) &
any_value(v) &
any_suit(sb).

The numbered statements in Tⁱ form the set, M. These imply "card(a)", i.e. S in the definition above. Only (3) has not been replaced because it is still part of the description of "b". Tⁱ⁺¹ is too general since the value of card "a" is no longer related to value of card "b". When an inconsistent generalization is produced, we must find a way of creating another, more specific trial.

A generalization results in a set of predicates in a trial being replaced by a single, more general, predicate. The replacement operation throws away some specific information contained in the set, M , in favour of the more general statement, S . If the new trial is consistent then the information lost was not important. However, if the generalization is inconsistent then too much information was lost. To make T^{i+1} more specific, Marvin re-examines the predicates in M to determine which ones contain essential information. In the example, (1), "value(a, v)" is necessary to maintain the condition that the two cards have the same value. Once this has been determined, the program tries to remember other concepts which contain a statement like (1).

Definition 6:

Let $T_1 = T - M_1 + S_1$
and $T_2 = T - M_2 + S_2$
where $M_1 * M_2 \neq \emptyset$

then $T_1 + T_2$ is a restriction of T_1 .

same_value(X, Y) is S_2 in definition 6, and may be added to the trial in an attempt to make a consistent trial T^{i+2} .

To test a trial T^{i+1} , Marvin shows the trainer an instance of T^{i+1} . If the program can create an object which does not belong to the target then T^{i+1} is inconsistent. The object will satisfy all the predicates in T^{i+1} but not satisfy any predicates in $A(T^i)$ except those implied by T^{i+1} . Thus, "same_value(a, b)" must be false so that the equality of values can be tested.

Definition 7: Let T^{i+1} be a generalization of T^i . Any object which satisfies T^{i+1} but negates each element of $A(T^i) - A(T^{i+1})$ is called a crucial object. If T^{i+1} is inconsistent and T^i is the target then no crucial object of T^{i+1} satisfies the target.

Let us now summarize Marvin's learning algorithm:

To learn a new clause:

The trainer gives the program an example of the concept.
- Find the list of replacement operations obtainable from the primary predicates.
- Use these operations to generalize the initial trial.
- Store the resulting concept in memory.

To find the list of replacement operations:

Let P initially be the set of all primary predicates.
- If P is a conjunction, find the set, I , of all concepts implied by P .
- Append I to P .
- Repeat this procedure until no more implied concept can be found. The result is $A(P)$.
The list of replacement operations consists of all the clauses used to find the implied concepts in $A(P)$. (see defn 1).

To generalize a trial T^i :

- Choose the first concept in the list of replacement operations.

- Perform the replacement to obtain T^{i+1} . (see defn 3).
- Try to "qualify" T^{i+1} .
- T^{i+1} cannot be qualified, abandon this generalization.
- Repeat this procedure with the next operation in the list.

To qualify a trial T^{i+1} :

- Construct a crucial object which satisfies T^{i+1} but does not satisfy $A(T^i) - A(T^{i+1})$ (defn 7).
- Ask trainer if object satisfies target. If it does then T^{i+1} has been qualified.
- If not then restrict T^{i+1} to new trial T^{i+2} .
- Try to qualify T^{i+2} .

To restrict a trial, T^{i+1} :

- Search list of replacement operations for an $S_2 \leftarrow M_2$ such that $M_1 * M_2 \neq \emptyset$. M_1 is the replacement for T^i .
- Perform replacement on T^{i+1} to obtain the new trial T^{i+2} . (see defn 6).

To construct an example from a trial, T^{i+1} :

- To construct an example from P & Q :
Construct P , Construct Q .
- To construct an example from an atomic predicate P and there is a clause $P \leftarrow B$ in memory, and B is not in $A(T^i) - A(T^{i+1})$, if B is already true, do nothing, otherwise construct example using B .
- To construct an example from an atomic predicate P and there is no clause $P \leftarrow B$, add B to conjunction representing example.

3. CONCEPTS THAT MARVIN HAS LEARNED

In this section we describe some of the concepts which Marvin has been able to learn.

Since concepts are represented as Horn clauses in first order logic, they may be regarded as logic programs. Thus Marvin is able to perform as an automatic programming system as well as a learning system.

Marvin has learned arithmetic concepts, including representing numbers as strings of bits, comparing those numbers and adding them. Also in the domain of program synthesis, Marvin has learned what lists are, how to append lists, reverse them, and sort lists of numbers using an insertion sort. Sorting required that the knowledge of numbers and lists be combined so that the elements of the list could be compared.

All of the program concepts above are disjunctive and recursive. To illustrate how such concepts are learned, consider the definition of a list of alphabetic characters. A list is either nil or an object which has a head (a letter) and a tail (a list).

The learning algorithm in §2 showed how a single clause could be learned by presenting an instance of the concept. If the concept is disjunctive then a new example is shown for each disjunct to be learned. Learning the alphabet is merely rote learning. The clauses shown below are entered by

the trainer and stored without generalization.

```
letter(X) <-value(X, a).
. . . . .
letter(X) <- value(X, z).
```

To teach Marvin about lists, the trainer shows it nil first:

```
list(n) <- value(n, nil).
```

The object "n" is a list because its value is nil. The term "value(n, nil)" is an atomic predicate which has no interpretation. This will be stored in memory as:

```
list(X) <- value(X, nil).
```

The object name "n" has been replaced by the variable "X". A further example of a list is:

```
list(w) <-
  head(w, h) &
  value(h, a) &
  tail(w, t) &
  value(t, nil).
```

Matching predicates in this example with the right hand sides of clauses stored in memory, the program will generalize "value(h, a)" to "letter(h)" and "value(t, nil)" to "list(t)".

In a different domain, Marvin is capable of learning simple grammars, such as:

```
S → NP VP
NP → Pnoun
NP → DET NOUN
Pnoun → John
Pnoun → Mary
DET → a
DET → the
NOUN → apple
NOUN → giraffe
VP → IVERB
VP → TVERB NP
IVERB → sleeps
IVERB → dreams
TVERB → dreams
TVERB → eats
```

Although we have no space to show the learning session for this grammar, it is an interesting task for Marvin since some concepts such as VP must be partially learned (one disjunct), then NP must be learned and then VP completed. See [Sammut, 1981] for a complete description.

A more difficult language recognition problem which Marvin has learned is that posed by [Hayes-Roth, 1978]. The task was to learn the rules to transform a sentence in the active form, such as, "The little man sang a lovely song" to the passive form, "A lovely song was sung by the little man".

No work in concept learning would be complete without some reference to Winston's ARCH [1970]. Marvin learned to describe physical objects with the properties: orientation, shape, being on top

of another object, touching another object. The program was then shown a single example of an arch. It generalized the description of the arch and tested the generalizations by showing the trainer its own descriptions of would-be arches until it produced a correct description.

4. BUILDING HIERARCHICAL MEMORIES

At present, the trainer is responsible for teaching concepts in an order which will allow the program to build a hierarchical memory. If the teaching sequence is not correct then Marvin may never be able to learn some concepts. For example, if we want to teach Marvin what a royal flush is then it must know which cards are court cards. What happens if these have not been taught before an example of a royal flush is shown?

In the present system, the program only shows the trainer one example to test a generalization. If the training sequence is carefully ordered, this is sufficient. If not, Marvin may produce an incorrect description. The concept "any_value" which we introduced earlier made no distinction between court cards and other cards. When constructing an example to show the trainer, Marvin could, by chance, choose court cards to test the generalization to "any_value". This would appear correct to the trainer, even though the concept should only refer to court cards. Thus, an inconsistent generalization could be passed as correct.

This problem could be avoided if Marvin showed more examples. This works for cards because there is only a finite number of objects in the domain. If the program is learning about number or lists, how many examples should be shown? The design of a good experiment to test a hypothesis only comes with some experience in the domain being investigated. Therefore, we should expect the program to make mistakes and develop some strategy for recovering when the mistake is discovered.

The choice of learning strategy will largely depend on the confidence the learning program has in its model of the world and the training data. That is, if contradictory results occur frequently then the model or data must not be correct. What to do when a contradiction appears may be controlled by heuristics which describe the confidence in a particular concept. For example, if Marvin is playing poker with its trainer (a somewhat far-fetched possibility) and the trainer realizes that the program's definition of a royal flush is incorrect, Marvin must be able to resolve this contradiction to its world model.

A possible course of action is as follows:

- initially, the example may be rejected as due to noisy data. This may be worthwhile doing if the concept in question has been used successfully elsewhere.
- if contradictions recur, they may be kept as special cases (exception conditions).
- If the recur frequently enough, then the model must be changed.

Let us simplify the learning task to "royal pair" rather than royal flush. Given the concepts we introduced earlier, Marvin could learn:

```
royal_pair(X, Y) <-
  playing_card(X) &
  playing_card(Y) &
  same_value(X, Y).
```

Clearly, this is an overgeneralization and will be discovered when Marvin's poker opponent points out that what it thought was a royal pair was not, i.e. it discovers a negative example such as:

```
not(royal_pair(X, Y)) <-
  suit(X, SX) &
  atom(SX, hearts) &
  value(X, V) &
  atom(V, 10) &
  suit(Y, SY) &
  atom(SY, diamonds).
```

Our problem now is to identify which generalization went wrong. Shapiro uses a method called backtracing to work back through the derivation of a hypothesis. At each point where a generalization was made, his program, MIS, asks an oracle (the trainer) if the generalization was valid. To do this with Marvin, we need to know the replacement operations performed to determine that X and Y were supposed to be a royal pair.

By questioning the trainer, asking which replacement caused the inconsistent generalization, the program will find that "any_value(V)" should not occur in the description. Hence all inferences following from this premise must be false, or at least must be qualified by additional restrictions.

There are several alternatives to the way in which we may proceed from here. If a concept like "any_value" has been used frequently in the descriptions of concepts that have proved useful then we may wish to leave the definition of "any_value" unchanged and just add an exception to the definition of royal pair. If the negative example used the value, 10, then we may add the qualification that the cards must have the same value as long as it is not 10. E.g.

```
royal_pair(X, Y) <-
  playing_card(X) &
  playing_card(Y) &
  value(X, V) &
  value(Y, V) &
  not(atom(V, 10)).
```

Assuming less confidence in "any_value", we may decide that any reference to it in "royal_pair" must be removed. Therefore generalizations made with "any_value" must be undone.

Suppose that Marvin has seen two positive examples of a royal pair: pair of kings and a pair of queens. This was generalized to a pair of any value. If the generalization "any_value" is then the program may return to a concept with two disjuncts: "a pair of kings or a pair of queens". A further example involving a jack would add a

third disjunct.

Now we run into another problem. Will we allow an arbitrary number of disjuncts to be added whenever contradictions in the model occur? With some introspection, Marvin may see that disjuncts involving kings, queens and jacks are created often. Therefore, it might be worthwhile to create a new concept for them, thus eliminating some disjuncts.

For example, if there are two clauses:

```
W <- Q & R & S.
W <- Q & R & P.
```

then they may be rewritten as:

```
W <- Q & R & SP.
SP <- S.
SP <- P.
```

Throughout this discussion we have used very vague terms such as "confidence" and "it may be worthwhile". This is because we are considering learning in an environment which does not enable the program when it has found the "right" description. An example of this in real life is that a scientific theory cannot be proven correct, only incorrect. The best we can do is to accumulate evidence to support the theory. The way we find the evidence is not always systematic but is often based on experience. Thus, we propose that a system which learns by discovery must also learn "confidence in its model".

5. CONCLUSION

The examples we have used in explaining the method of operation of Marvin may have been deceptive in their simplicity. A more complex example would have been more convincing of the powers of the program, but would probably have obscured its structure. It is important to note that the simple learning techniques devised for Marvin can lead to quite complex concepts.

The major motivation behind this work is the realization that, no matter how sophisticated a language be devised initially to describe objects, there is no assurance that the descriptions of object classes will remain conjunctive. The major difference between Marvin and many previously designed programs lies in the fact that it can handle disjunctions with almost the same ease as it can handle conjunctions. Both its strengths and weaknesses stem from this fact.

Let us now take stock of the advantages one loses once one gives up the insistence on conjunctive descriptions. The major advantage lay in the fact that once a positive example is focussed on at the beginning of the learning process, it is known that the final description will be the conjunction of a subset of all the predicates which the object satisfies. The only problem that remains is to weed out the unnecessary predicates. An easy way of doing this, especially if these predicates are independent, is to create test objects which

differ from the focus object in differing only by a single statement. If the test object lies in the concept then the corresponding statement is irrelevant.

Things get somewhat complicated if the statements are not independent. But as long as the description remains conjunctive, the problems are not insurmountable, as our above discussion of Marvin has shown. As a matter of fact, we have also shown that as long as the requirements of memory hierarchy can be met (i.e. the disjuncts of a concept are disjoint and the simpler concepts are placed in memory before the complex ones), even disjunctions can be handled.

That, then, is the main contribution of Marvin: to show that the advantages of conjunctive descriptions need not be given up entirely if one can retain a hierarchical memory structure. Of course, to retain these advantages, one has to give up the obsolete graph structures and nets of old and take advantage of the flexibilities of Horn Clauses. What causes the real problem is the question, "How does one retain a hierarchical memory structure while learning concepts?" Marvin has side-stepped this question by assuming a benevolent teacher who gives examples of simpler concepts before showing more complex concepts. We have described our initial thoughts about how one can try to keep memory hierarchical while trying to learn concepts without the help of a benevolent teacher.

6. REFERENCES

- Cohen, B.L., A Theory of Structural Concept Formation and Pattern Recognition, Ph.D. Thesis, Dept. of Comp. Sci., University of New South Wales, 1978.
- Hayes-Roth, F. and McDermott, J., "Knowledge Acquisition from Structural Descriptions", Communications of the ACM, Vol. 21, pp 401-411, May 1978.
- Michalski, R.S., "A Theory and Methodology of Inductive Learning", Machine Learning, Michalski, Carbonell, Mitchell (Eds), Tioga, Palo Alto, 1983.
- Mitchell, T.M., Utgoff, P.E., Banerji, R.B., "Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics", Machine Learning, Michalski, Carbonell, Mitchell (Eds), Tioga, Palo Alto, 1983.
- Sammur, C.A., Learning Concepts by Performing Experiments, Ph.D. Thesis, Dept. of Comp. Sci., University of New South Wales, 1981.
- Shapiro, E.Y., "Inductive Inference of Theories From Facts", Research Report 192, Yale University, 1981.
- Utgoff, P.E. and Mitchell, T.M., "Acquisition of Appropriate Bias for Inductive Concept Learning", Proc. Second National Conference on Artificial Intelligence, Pittsburgh, 1982.
- Winston, P.H., Learning Structural Descriptions From Examples, Ph.D. Thesis, MIT-AI Lab, 1970.

LEARNING AS A NON-DETERMINISTIC BUT EXACT LOGICAL PROCESS

Y. Kodratoff, J.-G. Ganascia
L.R.I., Bât. 490
Université de Paris-Sud
F-91405, Orsay CEDEX-FRANCE

ABSTRACT

In this paper we intend to show that learning is closely related to problems of generalization of expressions and changes in description languages. We claim that one must use properties of description, without forgetting anything, during the generalization process. So the definition of generalization we propose here is the one used in classical logic. On the other hand the forgetting action, which is central in learning, is but a change of description. In this sense a good description keeps only the meaningful characteristics for the learning task. Our aim is to demonstrate that regardless of the point of which the change of description takes place (either during or after the generalization process), any change which leads to forgetting anything, or to completing the description has to be preceded by an analysis of the generalization success or failure.

Key terms : Learning; machine learning, pattern matching, generalization, change of description, structural matching, canonical form.

I. INTRODUCTION

The problem of learning is felt as a central one in A.I. since "a computer that doesn't learn really cannot make much of a claim on being intelligent" [Shank 82]. In spite of this strong social pressure pushing us to study learning, we realize very well that the field improves at an amazingly low speed.

A description of the main ideas relevant to the field has been given by Dietterich and Michalski [Dietterich & Al. 81] in a particularly understandable paper. We shall here consider that the reader is aware of [Dietterich & Al. 81] since our own paper criticizes and refines the generalization concept used by these authors.

The ideas more or less implicitly found in [Hayes-Roth & Al. 78] and [Vere 81] are presented in a more systematic way and their consequences are analysed. Our main point is that generalization implies that the descriptions to be generalized must be given in quite a standard way. If this is not the case then some preliminary "massaging" is necessary as shown in section 2.2. The reader is reminded that Dijkstra used to say that some specification massaging is necessary before program

construction. In the following we hope to specify what "massaging" is.

II. KNOWLEDGE REPRESENTATION

The simple point we want to emphasize here is that generalization can be used as one of the tools necessary for building an efficient representation of knowledge.

If one admits that a good generalization algorithm must draw relevant information from a set of examples, then a failure to obtain this information must bring us to the idea that our representation of knowledge is not well adapted to the examples. If we have little knowledge about the world in which the learning is taking place, then we can only imagine some random changes in the representation. On the contrary, in a well-known and structured world, representation changes can be made in a systematic way.

Besides imagination or other meta-rules, generalization is an essential tool in the invention of the correct description language because it gives some properties the language must display so that one can learn from it.

The effect of fitting a description into a theory is also a very efficient way to find new descriptions (the meta-rules are given by the theory itself).

We shall now illustrate the above statements with two examples.

2.1. A "Bongard" example

Let us try to learn the difference between the following sets $S_1 = ((V, \Delta, *), (V, \Delta), (*), (V, \square))$ and $S_2 = ((V, \Delta, *, \square), (V, *), (\Delta, +), (\Delta, *))$, where S_1 is a set of examples and S_2 a set of counter examples.

The problem is : find a description of S_1 which excludes S_2 . The generalization algorithm we shall use has been given by Vere [Vere 80]. Let us suppose that we start form a description where the only information is the presence or absence of each geometrical pattern. For instance, $(V, \Delta, *)$ can be described by $(1, 1, 1, 0, 0)$ because we implicitly associate the first item of the quintuplet to V , the second one to Δ , the third one to $*$, the fourth one to $\square$ and the fifth one to $+$. The representation of S_1 and S_2 becomes : $S_1 = (A_1 = (1, 1, 1, 0, 0), A_2 = (1, 1, 0, 0, 0), A_3 = (0, 0, 1, 0, 0))$.

$D_3 = (\text{BLOCK } A) \ \& \ (\text{RED } B)$
 $D_4 = (\text{BLOCK } C)$

Then our "massaging" keeps the information that there are two objects, one of them being a block, the other one being coloured. If we assume that there exists at least one object which is coloured in red in our universe the generalisation of D_3 and D_4 should be
 $DG = \exists (BLOCK \ x) \ \& \ \exists y (COLOR \ RED \ y).$

Besides, there examples being trivial no very significant information is kept by our massaging but we hope that everyone is aware that we introduce here the possibility of a deduction during generalization, less trivial examples will be given in the following.

III. PREDICATE GENERALIZATION

It is quite evident that any description is a kind of logical formula which takes the value TRUE on the examples.

Generalizing these logical formulas is recognized as an important field of A.I. research (cf [Hayes-Roth & Al. 78], [Michalski 83], [Dietterich & Al. 81], [Vere 80], ...). On the other hand, classical logics define generalization for terms (recursively defined on free variables and functions) and not for formulas (recursively constructed from logical connectives and predicates depending on free or bound variables). As far as we need in some way to be able to generalize formulas in order to achieve even an elementary learning, two reactions are possible. One is the one used in [Hayes-Roth & Al. 78] and in [Vere 80]: stick as much as possible to classical logics but define a new way of generalizing. The other reaction will be described here (and has already been presented in [Kodratoff 82]): stick as much as possible to the classical definition of generalization but change the way of computing the canonical form of a logical formula. Our main reasons are as follows.

3.1.1. Learning implies anyhow that the computation of a canonical form is not the same as in classical logics. For instance, in any rewriting system computing a canonical form (as well for the well-known disjunctive or conjunctive normal forms (that are not canonical) as for the new Hsiang form [Hsiang 82]) an expression known for being TRUE must be rewritten as TRUE. On the contrary, in a learning context, a description is nothing but a tautology known for being TRUE: you introduce $SQUARE(A)$ in your description only if you know that A is actually a square.

As far as classical logics must be violated, we want to be as precise as possible in the estimation of what is new and what remains in learning versus classical logics systems.

3.1.2. The classical definition leads to difficulties due to the necessity of introducing operators that separate variables during the generalization process [Hayes-Roth & Al. 78] [Vere 81].

It is easy to see (an example follows) that

this problem is linked to the idempotency of logical AND. As we shall also see in the following, our presentation of generalization takes this difficulty into account. This is now exemplified by the following problem, where $\&$ is the logical AND, $A, B, C, \dots$ are constants while $x, y, z, \dots$ are variables. Let

$E1 = SQUARE(A) \ \& \ RED(A) \ \& \ SMALL(A)$ (There is a small red square) and

$E2 = SQUARE(B) \ \& \ RED(B) \ \& \ SQUARE(C) \ \& \ SMALL(C)$ (there are two squares, one is small, the other one is red).

The problem is to find the "best" generalization Eg of $E1$ and $E2$.

A partial matching generalization will lead either to
 $Eg1 = SQUARE(x) \ \& \ RED(x) \ \& \ SMALL(y)$ or to
 $Eg2 = SQUARE(y) \ \& \ RED(x) \ \& \ SMALL(y)$
 which are, for $x \leftarrow A, y \leftarrow A$, both instances of $E1$ but, for $x \leftarrow B, y \leftarrow C$, only partial instances of $E2$.

An alternative solution is the use of $\&$ idempotency relative to $SQUARE(A)$, stating that:

$E11 = SQUARE(A) \ \& \ RED(A) \ \& \ SQUARE(A) \ \& \ SMALL(A)$

is totally equivalent to $E1$ (this is sure from the logic's point of view, but is it also sure from a learning point of view?).

The generalization of $E11$ and $E2$ is:

$Eg = SQUARE(x) \ \& \ RED(x) \ \& \ SQUARE(y) \ \& \ SMALL(y)$ which has $E1$ as instance for $x \leftarrow A, y \leftarrow A$ (and because $SQUARE(A) \ \& \ SQUARE(A)$ reduces to $SQUARE(A)$) and which has $E2$ as instance for $x \leftarrow B, y \leftarrow C$.

A difficulty arises from the fact that $E2$ and $E3 = SQUARE(D) \ \& \ RED(D) \ \& \ SQUARE(E) \ \& \ SMALL(E)$ generalize also to Eg . Intuitively, this is a bad feature since $(E1, E2)$ is a different set of examples than $(E2, E3)$ (clearly, the first one is more particular) and they lead to the same "knowledge". The solution to this problem, discussed in [Hayes-Roth & Al. 78], [Kodratoff 82] and [Vere 80] is that one should look at the instances which produce the generalization and mark two variables u, v that never have the same instances in one example by: $DIFFERENT(u, v)$, and two variables w, t that have the same instance in one example by $SAME(w, t)$. For instance, the generalization of $E1$ and $E2$ should be

$Eg = SQUARE(x) \ \& \ RED(x) \ \& \ SQUARE(y) \ \& \ SMALL(y)$ since it is always implicitly stated that x and y can be instantiated by the same value, which happens in $E1 = E1$. On the contrary, the generalization of $E2$ and $E3$ should be:
 $Eg' = SQUARE(x) \ \& \ RED(x) \ \& \ SQUARE(y) \ \& \ SMALL(y) \ \& \ DIFFERENT(x, y)$ since the instances of x and y are different in $E2$ and $E3$ which implicitly contain the information $DIFFERENT(B, C)$ and $DIFFERENT(D, E)$ which should be used.

Marking each couple of variables when they have the same instantiation, or when they have different instantiation is equivalent to say that two variables which did not appear simultaneously in one $SAME$ or in one $DIFFERENT$ relationship, may be either identical, or unidentical. Another solution to this problem consists in marking two variables: either they have the same instantiation, with the $SAME$ relationship, or they may have the same instantiation by the MBS (May Be the Same) relationship. So two variables which are not marked are implicitly $DIFFERENT$.

For instance the generalization of E_1 and E_2 should be with the above convention
 $E_g = \text{SQUARE}(x) \& \text{RED}(x) \& \text{SQUARE}(y) \& \text{SMALL}(y) \& \text{MBS}(x, y)$
 and the generalization of E_2 and E_3 should be
 $E'_g = \text{SQUARE}(x) \& \text{RED}(x) \& \text{SQUARE}(y) \& \text{SMALL}(y)$
 Refer to [Kodratoff 82] for more details on the non-determinacy which is introduced in that way.

3.1.3. Unification in equational theories is a well-known field of research (see for instance [Stickel 81] or [Martelli & Al. 82]) and matching in an equational theory (even a complicated one) is quite possible [Costa 81].

This kind of problem comes down to the research of a canonical form of terms. Instead of changing the classical definition of generalization, we felt it more economical to modify the definition of a canonical term and to introduce "learning canonical terms".

3.1.4. Besides, it is not at all counter-intuitive to state that the usual canonical form of a logical formula is not particularly well-suited in order to express the trivialities one should start from when learning.

3.1.5. In the following, we shall briefly recall the definitions of generalization, then describe our way to compute the canonical forms of a term (learning canonical forms) and discuss some of its consequences.

We shall continue to use the above notations as well as the following: $x + A$ means that the variable x must be substituted by the constant A , V is the logical OR, $A < B$ means that A is more general than B , $A < > B$ means that A and B are not comparable from a generalization point of view.

3.2. Generalization

3.2.1. Generalization of terms

Let t_1 and t_2 be two terms. Saying that t_1 is more general than t_2 ($t_1 < t_2$) iff there is a substitution σ such that $\sigma t_1 = t_2$.

A substitution has the form $\sigma = (x_1 + t_1)$ where each x_i is a variable and each t_i is a term.

Example: $E_g = \text{SQUARE}(x) \& \text{RED}(x) \& \text{SQUARE}(y) \& \text{SMALL}(y)$

$E_2 = \text{SQUARE}(B) \& \text{RED}(B) \& \text{SQUARE}(C) \& \text{SMALL}(C)$

E_g is more general than E_2 since there exists a substitution σ such that $\sigma E_g = E_2$ where

$\sigma = (x + B, y + C).$

$E_{g1} = \text{SQUARE}(x) \& \text{RED}(x) \& \text{SMALL}(g)$

E_{g1} is not comparable with E_2 , indeed there does not exist any substitution such that $\sigma E_{g1} = E_2$.

3.2.2. Vere's definition of generalization [Vere 80]

3.2.2.1. Let us first consider conjunctions of elementary operators; any expression A is supposed to take the form $A = A_1 \& \dots \& A_n$ where each A_i is elementary (i.e. contains no logical connectives). He defines the set associated to A by

$\{A\} = \{A_1, \dots, A_n\}.$

Then A is more general than B iff there is a substitution such that $\sigma A = B'$ where $\{B'\} \subset \{B\}$.

Otherwise stated, σA is equal to a subpart of B .

Example: with the Vere's definition $E_g < E_2$ and $E_{g1} < E_2$.

There exists in fact a substitution

$\sigma, \sigma = (x + B, y + C)$, such that $\sigma E_g = E_2$, which implies $E_g < E_2$. On the other hand, $E_{g1} \{E_2\}$ being the set associated with E_2 , and $\{\sigma E_{g1}\}$ the set associated with σE_{g1} , $\{\sigma E_{g1}\} \subset \{E_2\}$, so $E_{g1} < E_2$.

3.2.2.2. For disjunctions of conjunctions, this definition becomes:
 let $G = g_1 \vee \dots \vee g_n$, $G_b = g_{b1} \vee \dots \vee g_{bm}$
 then G is more general than G_b iff $\forall j \exists i \text{ } g_{ai} < g_{bj}$.

Example:

$F_1 = (\text{SQUARE}(x) \& \text{RED}(x)) \vee (\text{SMALL}(y) \& \text{BLOCK}(y) \& \text{GREEN}(y))$

$E_2 = \text{SQUARE}(B) \& \text{RED}(B) \& \text{SQUARE}(C) \& \text{SMALL}(C)$

$F_1 = F'_1 \vee F''_1$ and there exists σ such that

$F'_1 = E'_2$ with $\{E'_2\} \subset \{E_2\}$, $\sigma = (x + B)$

Therefore F'_1 is more general than E_2 , and applying the above definition we can conclude that F_1 is more general than E_2 .

IV. OBTAINMENT OF A LEARNING CANONICAL FORM

There is an infinite number of "learning canonical forms" for a given term. On the other hand, we claim that there is a finite number of learning canonical forms of a term as compared to another. We shall always therefore speak of one of the canonical forms of t_1 and t_2 . Each of them is the least generalization of t_1 and t_2 which structurally matches with t_1 and t_2 . (Structural matching is defined in 4.2.1.). Actually if t structurally matches with t_1 and t_2 , then t is one of the generalizations of the learning canonical forms of t_1 compared with t_2 . Therefore structural matching may lead to the obtention of learning canonical forms, and limit their number. Each of them is obtained by applying the same rules which depend either on the representation, as in the AND idempotency, or on the description, as in the use of the taxonomy of predicates. The order in which one attempts structural matching, the rules used, and the priority of rules define the strategy. For instance, let be G_1 and G_2 two terms:

$G_1 = (\text{CIRCLE } A) \& (\text{SMALL } A) \& (\text{SQUARE } B) \& (\text{LARGE } B) \& (\text{ON } A \ B)$
 $G_2 = (\text{CIRCLE } C) \& (\text{SMALL } C) \& (\text{SQUARE } D) \& (\text{SMALL } D) \& (\text{LARGE } E) \& (\text{CIRCLE } E) \& (\text{ON } C \ D) \& (\text{ON } D \ E).$

We want to generalize G_1 and G_2 using idempotency of $\&$, and the two following predicates taxonomies:



If we take the same properties of descrip-

tions and representations for any given strategy of generalization, then the result depends on the strategy.

Let us suppose that, in the first strategy used, the predicates describing the shape of objects (i.e. descriptors which belong to the FORM's taxonomy) have the priority for structural matching. Using this strategy the generalization of G1 and G2 should be :

$$G' = (\text{CIRCLE } x_0) \& (\text{SIZE SMALL } x_0) \& (\text{SQUARE } y_0) \& (\text{SIZE } u_0 y_0) \& (\text{CIRCLE } z_0) \& (\text{SIZE } v_0 y_0) \& (\text{ON } x_0 y_0) \& (\text{ON } t_0 w_0) \& (\text{MBS } z_0 x_0) \& (\text{MBST } x_0) \& (\text{MBS } t_0 z_0) \& (\text{MBS } w_0 y_0) \& (\text{MBS } t_0 y_0) \& (\text{MBS } w_0 z_0).$$

On the other hand, if one wants to generalize G1 and G2 using another strategy giving priority to the predicates describing the size of the objects (i.e. descriptors which belong to the SIZE's taxonomy), we obtain the following generalization:

$$G'' = (\text{FORM CIRCLE } x_0) \& (\text{SMALL } x_0) \& (\text{FORM } u_0 y_0) \& (\text{LARGE } y_0) \& (\text{FORM } v_0 z_0) \& (\text{SMALL } z_0) \& (\text{ON } x_0 t_0) \& (\text{ON } z_0 y_0) \& (\text{MBS } x_0 z_0) \& (\text{MBS } y_0 t_0) \& (\text{MBS } t_0 z_0).$$

Notice that we use a convention for two variables. Those which are not simultaneously present in one MBS or in one SAME relationship are implicitly different (see 3.1.2.). Now we may examine the two generalizations G' and G'' : they are not comparable ($G' < > G''$) from a generalization point of view. G' fits together variables which correspond to objects having the same shape, and G'', to objects having same size. Therefore the strategy indicating the order in which we have to consider the different descriptors is not artificial, it has consequences for the generalized formulae.

Notice that we do not claim that :

- our way of computing a learning canonical term is unique. Using another strategy than the one here presented corresponds to another type of learning. For instance, we distinguish a "civilian" learning (where the logical operators idempotency has priority : it amounts to using the same object several times in different situations) as opposed to a "military" learning where each object is destroyed when used (in practice, this can be taken into account also by using operators associated with numbers so that $\text{PERSHINGMISSILE}(x,n)$ is different from $\text{PERSHINGMISSILE}(x,n+1)$).

- starting from unique learning canonical forms of t1 and t2, one gets a unique generalization. On the contrary, reference [Kodratoff 82] explains how one may reduce the size of the combinatory explosion of all the possible different generalizations.

4.1. The generalization axioms

There are cases where a substitution cannot be found and where the generalization ordering is quite clear. Instead of using another definition of generalization we introduced the following axioms which order some terms relative to generalization :

lization :

4.1.a. $\exists xP(x,y) < P(x,y)$

4.1.b. $P(A,y) < \forall zP(z,y)$

Axioms 4.1.a. and 4.1.b. deal with quantifiers but their intuitive meaning is not self-evident. To give some intuitive justification we have to take into account the following : let us suppose that E1 and E2 are two expressions, that $x_1, x_2, \dots, x_n$ are the free variables belonging to E1, and $y_1, y_2, \dots, y_p$ are the free variables belonging to E2.

If E1 and E2 have the following properties :

* $E1 < E2$

* $\{y_1, y_2, \dots, y_p\} \subset \{x_1, x_2, \dots, x_n\}$

Then, for all $(x_1, x_2, \dots, x_n)$ E2 is true implies that there exists same distribution of free variables which are not present in E2 such that E1 is true.

Example 1 :

E1 = $\text{SQUARE}(x) \& \text{RED}(x) \& \text{SMALL}(y) \& \text{SQUARE}(y)$

E2 = $\text{SQUARE}(x) \& \text{RED}(x) \& \text{SMALL}(x) \& \text{SQUARE}(x)$

*E1 < E2 because there exists σ such that $\sigma E1 = E2$;
 $\sigma = (x \leftarrow x, y \leftarrow x)$

*x which is the only free variable in E2 is present in E1. For each x E2(x) is true implies that there exists y (y = x) such that E1(x,y) is true.

Example 2 :

Another example may be more convincing : we shall demonstrate (see 4.2.4.) that (LOWER x y) is more general than (EQ x 1) & (EQ y 2). This proposition, not surprisingly, corresponds to our intuitive feeling because for all x and for all y, (EQ x 1) & (EQ y 2) implies (LOWER x y).

Now if we try to apply the above assertion to formulae which contain quantifiers, we can verify that it is, in fact, true :

*Axiom 4.1.a : $\exists xP(x,y) < P(x,y)$. Applying the last assertion to this axiom leads us to : for all x and for all y $P(x,y) \Rightarrow \exists xP(x,y)$ which is always true.

*Axiom 4.1.b : $P(A,y) < \forall zP(z,y)$

Similarly axiom 4.1.b leads to : for all y $\forall z P(z,y) \Rightarrow P(A,y)$ which is a tautology.

4.2. The elementary rewriting steps

4.2.1. Structural matching

Let us say that t1 matches structurally t2 when t1 becomes identical to t2 (modulo a variable renaming) when all the leaves of t1 and t2 are given variables names. The messaging introduced in section 2.2. indicates that learning canonical forms structurally match each other as far as possible.

4.2.2. Using idempotency

It amounts to a use of idempotency in order to favor a structural matching of t1 and t2.

For instance, suppose that $t_1 = P \& Q$ and $t_2 = Q$, one then uses idempotency when one tries to find if Q can take the form $Q' \& Q''$ such that Q' structurally matches P . The term t_2 is then rewritten as $t_2' = Q' \& Q$ which is logically equivalent to t_2 but allows us to attempt a matching of $t_2 = t_2'$ and t_1 .

4.2.3. Detecting a matching

Given t_1 and t_2 , one attempts to find substitutions such that $\sigma t_1 = t_2$ or $\sigma t_2 = t_1$.

4.2.4. Using knowledge of descriptors

When two expressions do not structurally match we can enforce the match by using rules describing the semantic of descriptors. As we shall see in 4.2.5. if no structural match does appear, one may "lose" information by "abstracting" terms. We illustrate, in the following example an application of these rules :

It is intuitively clear that $(\text{LOWER } X \ Y)$ is more general than $(\text{EQ } X \ 1) \ \& \ (\text{EQ } Y \ 2)$, but actually, this is not easy to demonstrate. In order to do this, we need to use some semantic rules :

- (1) $\forall X (\text{EQUAL } X \ X) = \text{True}$
- (2) $(\text{LOWER } 1 \ 2) = \text{True}$
- (3) $(\text{LOWER } X \ Y) (\text{EQUAL } X \ Z) = (\text{LOWER } Z \ Y)$
- (4) $(\text{LOWER } X \ Y) (\text{EQUAL } Y \ Z) = (\text{LOWER } X \ Z)$

Now let us suppose that :

$$\begin{aligned} T_1 &= (\text{LOWER } X_1 \ Y_1) \\ T_2 &= (\text{EQ } X_2 \ 1) \ \& \ (\text{EQ } Y_2 \ 2) \end{aligned}$$

There is no substitution σ such that $\sigma T_1 = T_2$. In order to demonstrate that T_1 is more general than T_2 , first we generalize T_1 and T_2 .

Let T the least generalization of T_1 and T_2 . Then we try to prove that $T = T_1$. In fact $T < T_2$ and $T = T_1$ implies that $T_1 < T_2$.

Obtainment of T :

We attempted structural matching between T_1 and T_2 , but it failed : there is no descriptor EQ in T_1 and no descriptor LOWER in T_2 . We try to make apparent these descriptors by using semantic rules.

Therefore we obtain :

$$\begin{aligned} T_1' &= (\text{LOWER } X_1 \ Y_1) \ \& \ \forall X (\text{EQ } X \ X) \\ T_2' &= (\text{EQ } X_2 \ 1) \ \& \ (\text{EQ } Y_2 \ 2) \ \& \ (\text{LOWER } 1 \ 2) \end{aligned}$$

As there is no universal quantifier in T_1' , we make it disappear by using the axiom 4.1.b² :

$$\begin{aligned} T_1'' &< T_1' \\ T_1'' &= (\text{LOWER } X_1 \ Y_1) \ \& \ (\text{EQ } X_1 \ X_1) \ \& \ (\text{EQ } Y_1 \ Y_1) \\ T_2' &= (\text{LOWER } 1 \ 2) \ \& \ (\text{EQ } X_2 \ 1) \ \& \ (\text{EQ } Y_2 \ 2) \end{aligned}$$

We try to detect the match between T_1'' and T_2' .

$$\begin{aligned} \sigma_1 &= (X_1 \leftarrow X_1, 1 ; Y_1 \leftarrow Y_1, 2) \\ T_1''' &= (\text{LOWER } X_0 \ Y_0) \ \& \ (\text{EQ } X_0 \ X_0) \ \& \ (\text{EQ } Y_0 \ Y_0) \\ T_2' &= (\text{LOWER } X_0 \ Y_0) \ \& \ (\text{EQ } X_2 \ X_0) \ \& \ (\text{EQ } Y_2 \ Y_0) \end{aligned}$$

There is no match because, in T_2' , X_2 appears in the $(\text{EQ } X_2 \ X_0)$ and the only variables in T_1''' are X_0 and Y_0 . So T_1''' is too particular, and we generalize it by creating two new variables :

$$\begin{aligned} T_1''' &< T_1'' \\ T_1''' &= (\text{LOWER } X_0 \ Y_0) \ \& \ (\text{EQ } X_3 \ X_0) \ \& \ (\text{EQ } Y_3 \ Y_0) \ \& \\ & \quad (\text{SAME } X_3 \ X_0) \ \& \ (\text{SAME } Y_3 \ Y_0) \end{aligned}$$

Now we may obtain T which is the least generalization of T_1 and T_2 , by using the substitution $\sigma_2 = (U_0 \leftarrow X_3, X_2 ; V_0 \leftarrow Y_3, Y_2)$

$$\begin{aligned} T &= (\text{LOWER } X_0 \ Y_0) \ \& \ (\text{EQ } U_0 \ X_0) \ \& \ (\text{EQ } V_0 \ Y_0) \ \& \\ & \quad (\text{MBS } X_0 \ U_0) \ \& \ (\text{MBS } Y_0 \ V_0) \end{aligned}$$

The reader can check that T is more general than T_1 and T_2 . Now, using rules (3) and (4) we obtain :

$$\begin{aligned} T &= (\text{LOWER } U_0 \ V_0) \text{ which implies :} \\ T(X_0, Y_0, U_0, V_0) &= T_1(U_0, V_0). \end{aligned}$$

4.2.5. Developping or reducing to TRUE or FALSE values

One can also attempt to achieve the structural match between t_1 and t_2 by using the tautologies :

$$A \ \& \ \text{TRUE} = A \quad \text{and} \quad A \ \vee \ \text{FALSE} = A,$$

in either direction depending on t_1 and t_2 . For instance, suppose that $t_1 = P(x) \ \& \ Q(x)$ and $t_2 = P(x)$, then we can try to prove that $Q(x) = \text{TRUE}$ so that t_1 reduces to t_2 or $Q(A) = \text{TRUE}$ so that the canonical form of t_2 is $t_2' = P(x) \ \& \ Q(A)$. In each case, the learning canonical forms of t_1 and t_2 are brought to structural matching.

4.2.6. Description transformation

One may bring t_1 and t_2 to structural matching by considering that the "missing operators" of, say, t_1 are actually variables of a more general operator which must be introduced into the description (how to invent these new operators is not considered here. As the following examples show one needs a good knowledge of the universe semantics). For instance, suppose that one wishes to compare $t_1 = P(x) \ \& \ Q(x)$ and $t_2 = P(x)$.

Instead of using idempotency, it may be more useful to consider P and Q as instances of (functional) variables and find some more general operator G such that $G(P, x)$ and $G(Q, x)$ have the same interpretation than $P(x)$ and $Q(x)$. This is useful when one has some information about $G(y, x)$. For instance, consider the assertion $\text{RED}(x)$ (there are objects that can have the red hue), it may be useful to rewrite it as $\text{COLOR}(\text{RED}, x)$ when, for instance, one knows that all the objects in our universe are colored, i.e. $\forall x \exists y \text{COLOR}(y, x) = \text{TRUE}$, especially in view of applying 4.2.5.

4.3. Abstraction

In theory, this brings us to second order logic but in practice this may well reduce to a kind of hierarchy among operators which is considered as being part of the knowledge basis describing our universe.

As in the example (see IV.), the generalization of G_1 and G_2 leads either to G' which intro-

duces general descriptor, SIZE, or to G" which introduces general descriptor FORM.

Nevertheless it happens that whatever description transformation is used, a description which appears in an example may not appear in another. In this case structural matching is impossible. In order to generalize, one may remove the term, but this lead to forgetting information. This is the classical A.I. way to generalize, which is symbolized by the use of the "Dropping condition rule" [Dietterich & Al. 81].

We could criticize this way of generalizing because it does not follow the classical logic definition of generalization, unless we assume that "True" is more general than any particular assertion and "False" less general.

For instance, let $E1 = P(y) \& Q(z)$ and $E2 = P(x)$. On one hand Vere's definition leads to $E2 < E1$ and, in the same way, the "Dropping condition rule" used by Michalski (see [Dietterich & Al. 81]) leads to $E2 < E1$. On the other hand in our theory, if $P=Q$ then $E1 < E2$ (see 4.4.8.), unless $P \neq Q$. However if P and Q belong to the same predicate taxonomy then the two formulae $E1$ and $E2$ are comparable from a generalization point of view only if $z \neq y$ (i.e. z and y are not the same parameter) (see 4.4.2.). In this case $E2 < E1$ (see 4.4.4.). Lastly if P and Q do not belong to the same predicate taxonomy then $E2 < E1$ (see 4.4.3. and 4.4.5.). If we assume as Vere and Mikalski, that whatever P and Q are, $E2$ is more general than $E1$, this implies if $P(x) = \text{True}$ that $\text{True} < Q(z) \& \text{True}$. Therefore, using the fact that $Q(z) \& \text{True} = Q(z)$, this leads to $\text{True} < Q(z)$ whatever Q may be. In the same way, it is easy to demonstrate that with Vere and Michalski's definition "False" is more particular than any other assertion.

In the case studied above, there is no way to obtain the structural matching unless we "lose" some pieces of information, but we claim that it is not permissible to omit anything during the generalization process. In order to go through the structural matching, we need to make an abstraction: This operation is equivalent to removing the term which makes the matching impossible, while keeping this term in memory. Therefore, after the generalization process, the least generalization will match with the examples, modulo terms which have been abstracted. Because we have kept these terms, we can precisely verify the matching, and analyze the failure of generalization by reading these terms. We can therefore conclude that, first, learning becomes an exact process, because nothing is lost, and secondly, the generalization process may guide the changes in description by showing the result and the terms abstracted; thereby displaying redundancies and lacks of descriptions.

4.4. Examples of the consequences of our definitions

We give examples into a universe of colored shapes so that $\text{RED}(A)$ can be rewritten as $\text{COLOR}(\text{RED}, A)$ and $\text{BLOCK}(A)$ as $\text{SHAPE}(\text{BLOCK}, A)$. Notice that we assume that it is impossible to learn from partial descriptions into universes where this

kind of knowledge is not known.

The starred examples are more or less in contradiction with the usual definitions of generalization and they will be further commented.

4.4-1. $\text{RED}(A) < \text{RED}(A) \& \text{BLOCK}(A)$

*4.4-2. $\text{RED}(A) < > \text{RED}(A) \& \text{BLUE}(A)$ (in general)

4.4-3. $\text{RED}(A) < \text{RED}(A) \& \text{BLOCK}(B)$

*4.4-4. $\text{RED}(A) < \text{RED}(A) \& \text{BLUE}(B)$

4.4-5. $\text{RED}(x) < \text{RED}(y) \& \text{BLOCK}(y)$

*4.4-6. $\text{RED}(x) < > \text{RED}(y) \& \text{BLUE}(y)$

4.4-7. $\text{RED}(x) < \text{RED}(y) \& \text{BLOCK}(z)$

*4.4-8. $\text{RED}(y) \& \text{RED}(z) < \text{RED}(x)$

4.4-9. $\text{RED}(A) \vee \text{BLOCK}(A) < \text{RED}(A)$

*4.4-10. $\text{RED}(A) \vee \text{BLUE}(A) < \text{RED}(A)$ (under some conditions)

4.4-11. $\text{RED}(A) \vee \text{BLOCK}(B) < \text{RED}(A)$

4.4-12. $\text{RED}(y) \vee \text{BLOCK}(y) < \text{RED}(x)$

*4.4-13. $\text{RED}(y) \vee \text{BLUE}(y) < \text{RED}(x)$ (under some conditions)

4.4-14. $\text{RED}(y) \vee \text{BLOCK}(z) < \text{RED}(x)$

4.4-15. $\text{RED}(y) \vee \text{RED}(z) < \text{RED}(x)$

Proving 4.4-2.

In general, $\text{RED}(A) \& \text{BLUE}(A)$ is equivalent to an other operator, either FALSE or REDBLUESPOTTED(A) or ROSY(A) etc... Using a distance notion which is intuitively clear in that case (and altogether not defined in this paper), replacing $\text{RED}(A) \& \text{BLUE}(A)$ by one of its equivalent form leads to an expression "nearer" from $\text{RED}(A)$ (because a failure of matching with & is avoided), and therefore those equivalent forms are canonical and $\text{RED}(A)$ is not comparable to any of them (unless it so in a very specific universe).

Proving 4.4-4.

One changes the description and $t1 = \text{RED}(A)$ becomes $t1' = \text{COLOR}(\text{RED}, A)$ and $t2$ becomes $t2' = \text{COLOR}(\text{RED}, A) \& \text{COLOR}(\text{BLUE}, B)$. In $t1$ one has the knowledge that B has a color (which is not valid in regard of $t2$ but because one supposes that all objects of the universe are colored, this may be untrue and $t1$ and $t2$ would then be uncomparable), i.e. $\exists x \text{COLOR}(x, B) = \text{TRUE}$. $t1'$ becomes $t1' = t1' \& \text{TRUE}$
 $= \text{COLOR}(\text{RED}, A) \& \exists x \text{COLOR}(x, B)$
 $= \exists x (\text{COLOR}(\text{RED}, A) \& \text{COLOR}(x, B))$
 which is more general than $t2'$ by 4.1.a;

Proving 4.4-8.

One uses idempotency and

$t1 = \text{RED}(y) \& \text{RED}(z)$ is more general than $t2' = \text{RED}(x) \& \text{RED}(x)$ because $\sigma t1 = t2$, where $\sigma = (y \rightarrow x, z \rightarrow x)$.

Proving 4.4-10.

Let $t1 = \text{RED}(A) \vee \text{BLUE}(A)$ and $t2 = \text{RED}(A)$. We change the representation and write :
 $t1 = t1' = \text{COLOR}(\text{RED}, A) \vee \text{COLOR}(\text{BLUE}, A)$ (which, as opposed to 4.4-2. has generally no implication) and $t2 = t2' = \text{COLOR}(\text{RED}, A)$.
 There are two cases :
 either A is the object which has all colors, i.e.
 $\forall x \text{ COLOR}(x, A) = \text{TRUE}$, then $t1 < t2$
 or A is not spotted by all colors, i.e.
 $\forall x \text{ COLOR}(x, A) = \text{FALSE}$ and it follows :
 $t2'' = t2' \vee \text{FALSE}$
 $= \text{COLOR}(\text{RED}, A) \vee \forall x \text{ COLOR}(x, A)$
 $= \forall x (\text{COLOR}(\text{RED}, A) \vee \text{COLOR}(x, A))$
 and $t1' < t2''$ because of 4.1.b.

Proving 4.4-13.

The same proof as in 4.4-10. applies, except that one must be in a universe where there is not at all any object spotted by all colors, i.e. $\forall x \text{ COLOR}(x, y) = \text{FALSE}$ to be able to compare the two clauses.

V. ALGORITHM FOR GENERALIZING

5.1. Difficulties of such an algorithm

Besides the well-known difficulties due to the use of Associative-Commutative functions, some supplementary problems are introduced by the use of idempotency and semantic rules. The central problem is the obtainment of a learning canonical form : it is solved by an incremental matching process which manage it.

Our approach may lead to obtain a great number of distinct generalizations, but some of them only are usefull. The role of the strategy is to diminish the size of the search space. The strategy may depend itself on the kind of problem we have to solve. It indicates the order in which descriptors must be considered. For instance, in a block world, the central point may be the size of the blocks, or their shape or the above relationship between blocks, etc..., or it may be independent of the problem semantic, and for example, take only into account the number of variables which simultaneously appear in the generalized formula.

5.2. Description of algorithm

Step 1

Verify that each descriptor which appears at least in one example, appears in all the examples. If this is not the case, we try to make appear these descriptors by using some semantic rules. If this is impossible we need to make an "abstraction" (see 4.3.).

Step 2

One strategy being given, we choose the des-

criptors on which we have to generalize first.

Step 3

We mark each occurrence of apparition of each descriptor having been chosen in step 2, on each example, in order to compute the different possible generalizations.

Example :

Let us suppose that our strategy leads us to generalize examples E_1' , E_2' and E_3 on the descriptor SMALL :

$E_1' = (\square B) \& (\bigcirc C) \& (\text{LARGE } A) \& (\text{SMALL}_1 B) \& (\text{SMALL}_2 C) \& (\text{INSIDE } B A) \& (\text{ABOVE } A C) \& (\text{ABOVE } B C)$
 $E_2' = (\square D) \& (\bigcirc f) \& (\text{LARGE } E) \& (\text{SMALL}_3 F) \& (\text{SMALL}_4 D) \& (\text{INSIDE } F E) \& (\text{ABOVE } D E) \& (\text{ABOVE } D F)$
 $E_3 = (\square G) \& (\bigcirc H) \& (\bigcirc I) \& (\text{LARGE } H) \& (\text{SMALL}_5 G) \& (\text{SMALL}_6 I) \& (\text{INSIDE } I H) \& (\text{ABOVE } G H) \& (\text{ABOVE } G I)$

We may firstly generalize on SMALL_1 , SMALL_3 and SMALL_5 , or on SMALL_1 , SMALL_4 and SMALL_5 , or ... : we have to envisage 8 possible ways for generalizing.

Step 4

We envisage separably each possible generalization. In each one we substitute one variable to the term which appears under the scope of the descriptor having been chosen. We operate simultaneously on all the examples.

Step 5

Let be x_0 the variable that we have substituted in step 4. We try to obtain one learning canonical form in regard to x_0 . One have to verify that all the occurrences of x_0 are the same in all the examples.

If this is not the case :

- firstly, we try to make appear each occurrence of x_0 on all the examples by using either representation rules (see 4.2-2. or 4.2-5.) or knowledge about description (see 4.1., 4.2-3. or 4.2-6).
- Secondly, if we fail to make appear one occurrence of x_0 on at least one example, we make disappear this occurrence on all the examples in which it appears by creating a new variable.

Notice that we use the convention described in 3.1.2. : we mark two variables which are identical by using the SAME relationship, and two variables which may be the same by the MBS relationship. Two variables which does not appear in any SAME or MBS relationship are implicitly different.

Step 6

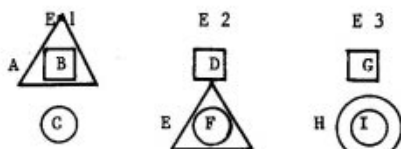
We return to step 2 until there does not appear any occurrence of variable or constant, under the scope of a descriptor, which may not be the same in all the example that we want to generalize.

Notice that the substitutions of new vari-

ables, and the "massaging" which are introduced by them, must not transform any occurrence of variables which have been previously substituted.

5.3. Example

Let be E1, E2 and E3 the three examples which have been presented in [Hayes-Roth & Al. 78] :



Each example is described by a conjunction of clauses. Each of them describes either an object, or a spatial relation between two objects. There are 5 descriptors for objects : Δ which means triangle, $\square$ square, $\bigcirc$ circle, SMALL small size and LARGE large size. There are 2 descriptors for spatial relation : INSIDE and ABOVE.

E1 = (Δ A) & ($\square$ B) & ($\bigcirc$ C) & (LARGE A) & (SMALL B) & (SMALL C) & (INSIDE B A) & (ABOVE A C) & (ABOVE B C)
 E2 = (Δ E) & ($\square$ D) & ($\bigcirc$ F) & (LARGE E) & (SMALL D) & (SMALL F) & (INSIDE F E) & (ABOVE D F) & (ABOVE D E)
 E3 = ($\bigcirc$ H) & ($\square$ G) & ($\bigcirc$ I) & (LARGE H) & (SMALL G) & (SMALL I) & (INSIDE I H) & (ABOVE G H) & (ABOVE G I).

Let us assume that there are two predicate taxonomies. The first relates the predicates describing the shape of the objects (Δ , $\square$, $\bigcirc$). We call it the FORM taxonomy. The second relates the predicates describing the size of the objects, we call it the SIZE taxonomy.



We shall omit the & in the following.

Step 1

The descriptor Δ which appears in E1 and E2 does not appear in E3. The predicate taxonomy of FORM permits us to transform the description :

E1 = (FORM Δ A) (FORM $\square$ B) (FORM $\bigcirc$ C) (LARGE A) (SMALL B) (SMALL C) (INSIDE B A) (ABOVE A C) (ABOVE B C)
 E2 = (FORM Δ E) (FORM $\square$ D) (FORM $\bigcirc$ F) (LARGE E) (SMALL D) (SMALL F) (INSIDE F E) (ABOVE D F) (ABOVE D E)
 E3 = (FORM $\bigcirc$ H) (FORM $\square$ G) (FORM $\bigcirc$ I) (LARGE H) (SMALL G) (SMALL I) (INSIDE I H) (ABOVE G H) (ABOVE G I).

Step 2

Now suppose our strategy manage us to minimize the number of generalization. This guide us to choose the descriptor LARGE or the descriptor INSIDE : there are five descriptors. Let be $N(E_i, D)$ the number of occurrences of descriptor D in the example E_i . The number of ways to generalize on D is equal to : $N(E1, D) * N(E2, D) * N(E3, D)$

Each choice of descriptor leads to a different generalization. The role of the strategy is either to minimize the number of possible generalization, or to indicate the most meaning full descriptor which has to be the focal point in the generalization process. Whatever the strategy is, it leads to some generalizations which are usually not comparable each other from a generalization point of view.

Let us assume now that our strategy contains another criteria which manage us to generalize first, with the descriptors which qualify the objects, as FORM, LARGE and SMALL, and secondly, with the descriptors which describe relationship between objects, as INSIDE and ABOVE.

In our case, the strategy leads us to generalize on LARGE.

Step 3

There is only one occurrence of LARGE in each of the example, therefore, there is only one possible way of generalizing.

Step 4

We substitute x_o to A, E and H in all the examples :

E1 = (FORM Δ x_o) (FORM $\square$ B) (FORM $\bigcirc$ C) (LARGE x_o) (SMALL B) (SMALL C) (INSIDE B x_o) (ABOVE x_o C) (ABOVE B C)
 E2 = (FORM Δ x_o) (FORM $\square$ D) (FORM $\bigcirc$ F) (LARGE x_o) (SMALL D) (SMALL F) (INSIDE F x_o) (ABOVE D F) (ABOVE D x_o)
 E3 = (FORM $\bigcirc$ x_o) (FORM $\square$ G) (FORM $\bigcirc$ I) (LARGE x_o) (SMALL G) (SMALL I) (INSIDE I x_o) (ABOVE G x_o) (ABOVE G I)

Step 5

Occurrences of x_o for each example

E1	(LARGE x_o) (FORM Δ x_o)	(INSIDE B x_o) (ABOVE x_o C)
E2	(LARGE x_o) (FORM Δ x_o)	(INSIDE F x_o) (ABOVE D x_o)
E3	(LARGE x_o) (FORM $\bigcirc$ x_o)	(INSIDE I x_o) (ABOVE G x_o)
	(LARGE x_o)	(INSIDE x_o)

But (ABOVE x_o h) appears only in E1 and no rule permits to make appear it in E2 and E3, in the same way (ABOVE h x_o) appears in E2 and E3 and does not appear in E1.

We have to make disappear (ABOVE x_o C) from E1, (ABOVE D x_o) from E2 and (ABOVE G x_o) from E3.

E1 = (FORM Δ x_o) (FORM $\square$ B) (FORM $\bigcirc$ C) (LARGE x_o) (SMALL B) (SMALL C) (INSIDE B x_o) (ABOVE x_o C) (ABOVE B C) (SAME u_o x_o)
 E2 = (FORM Δ x_o) (FORM $\square$ D) (FORM $\bigcirc$ F) (LARGE x_o) (SMALL F) (INSIDE F x_o) (ABOVE D F) (ABOVE D v_o) (SAME v_o x_o) (SMALL D)
 E3 = (FORM $\bigcirc$ x_o) (FORM $\square$ G) (FORM $\bigcirc$ I) (LARGE x_o) (SMALL G) (SMALL I) (INSIDE I x_o) (ABOVE G w_o) (ABOVE G I) (SAME x_o w_o)

Now step 6 manages us to go to step 2. This leads us to generalize either on (FORM Δ h) or on (FORM $\square$ h).

1 - Generalization on (FORM Δ h)

$y_o + C, F, I$

- E1 = (FORM Δ x_o)(FORM $\square$ B)(FORM Δ y_o)(LARGE x_o)
(SMALL B)(SMALL y_o)(INSIDE B x_o)(ABOVE u_o y_o)
(ABOVE B y_o)(SAME u_o x_o)
E2 = (FORM Δ x_o)(FORM $\square$ D)(FORM Δ y_o)(LARGE x_o)
(SMALL D)(SMALL y_o)(INSIDE y_o x_o)(ABOVE D y_o)
(ABOVE D v_o)(SAME v_o x_o)
E3 = (FORM Δ x_o)(FORM $\square$ G)(FORM Δ y_o)(LARGE x_o)
(SMALL G)(SMALL y_o)(INSIDE y_o x_o)(ABOVE G w_o)
(ABOVE G y_o)(SAME x_o w_o)

Step 5

(ABOVE u_o y_o) which appears in E1 does not appear in E2 and E3. In the same way (INSIDE y_o x_o) which appears in E2 and E3 does not appear in E1. On one hand, we can make appear (ABOVE u_o y_o) in E2 and E3, by the use of δ idempotency. On the other hand, there is no way to make appear (INSIDE x_o y_o) in E1. Therefore we are leading to the following generalization :

- E1 = (FORM Δ x_o)(FORM $\square$ B)(FORM Δ y_o)(LARGE x_o)
(SMALL B)(SMALL y_o)(INSIDE B x_o)(ABOVE u_o y_o)
(ABOVE B y_o)(SAME u_o x_o)
E2 = (FORM Δ x_o)(FORM $\square$ D)(FORM Δ y_o)(LARGE x_o)
(SMALL D)(SMALL y_o)(INSIDE y_o x_o)(ABOVE D y_o)
(ABOVE D v_o)(SAME v_o x_o)(SAME z_o y_o)
(ABOVE t_o y_o)(SAME t_o D)
E3 = (FORM Δ x_o)(FORM $\square$ G)(FORM Δ y_o)(LARGE x_o)
(SMALL G)(SMALL y_o)(INSIDE z_o x_o)(ABOVE G w_o)
(ABOVE G y_o)(SAME x_o w_o)(SAME z_o y_o)
(ABOVE t_o y_o)(SAME t_o G)

2 - Generalization on (FORM $\square$ h)

$y_o + B, D, G$

Using the same rewriting steps leads us to the following generalization :

- E1 = (FORM Δ x_o)(FORM $\square$ y_o)(FORM Δ C)(LARGE x_o)
(SMALL y_o)(SMALL C)(INSIDE z_o x_o)(ABOVE u_o C)
(ABOVE y_o C)(SAME u_o x_o)(SAME z_o y_o)
(ABOVE y_o t_o)(SAME t_o C)
E2 = (FORM Δ x_o)(FORM $\square$ y_o)(FORM Δ F)(LARGE x_o)
(SMALL y_o)(SMALL F)(INSIDE F x_o)(ABOVE y_o F)
(ABOVE y_o v_o)(SAME v_o x_o)
E3 = (FORM Δ x_o)(FORM $\square$ y_o)(FORM Δ I)(LARGE x_o)
(SMALL y_o)(SMALL I)(INSIDE I x_o)(ABOVE y_o w_o)
(ABOVE y_o I)(SAME x_o w_o)

If we pursue the generalization process we obtain two possible generalizations.

The first one corresponds to the first generalization on (FORM Δ h) :

- E1 = (FORM Δ x_o)(FORM $\square$ k_o)(FORM Δ y_o)(LARGE x_o)

- (SMALL k_o)(SMALL y_o)(INSIDE z_o x_o)
(ABOVE u_o y_o)(ABOVE k_o y_o)(SAME u_o x_o)
(SAME k_o z_o)(ABOVE k_o p_o)(SAME p_o y_o)

- E2 = (FORM Δ x_o)(FORM $\square$ k_o)(FORM Δ y_o)(LARGE x_o)
(SMALL k_o)(SMALL y_o)(INSIDE z_o x_o)(ABOVE k_o y_o)
(ABOVE k_o v_o)(SAME v_o x_o)(SAME z_o y_o)
(ABOVE t_o y_o)(SAME t_o k_o)

- E3 = (FORM Δ x_o)(FORM $\square$ k_o)(FORM Δ y_o)(LARGE x_o)
(SMALL k_o)(SMALL y_o)(INSIDE z_o x_o)
(ABOVE k_o w_o)(ABOVE k_o y_o)(SAME x_o w_o)
(SAME z_o y_o)(ABOVE t_o y_o)(SAME t_o k_o)

This leads to the following generalization formula :

- EG1 = (FORM u_o x_o)(FORM $\square$ k_o)(FORM Δ y_o)
(LARGE x_o)(SMALL k_o)(SMALL y_o)(INSIDE z_o x_o)
(ABOVE t_o y_o)(ABOVE k_o y_o)(ABOVE k_o v_o)
(MBS t_o x_o)(MBS v_o x_o)(MBS v_o y_o)(MBS y_o z_o)
(MBS k_o z_o)(MBS k_o t_o)

We can verify that EG1 is more general than E1, E2 and E3 :

- $\sigma_1' = (u_o + \Delta, x_o + A, y_o + C, k_o + B, z_o + B,$
 $v_o + C, t_o + A)$
 $\sigma_2' = (u_o + \Delta, x_o + E, y_o + F, k_o + D, z_o + F,$
 $v_o + E, t_o + D)$
 $\sigma_3' = (u_o + \Delta, x_o + H, y_o + I, k_o + G, z_o + I,$
 $v_o + H, t_o + G)$

The reader can easily check that
 $\sigma_1' EG1 = E1, \sigma_2' EG1 = E2$ and $\sigma_3' EG1 = E3$

The second possible generalization which corresponds to the first generalization on (FORM $\square$ h) is equal to EG1, modulo a variable renaming.

Now, let us assume that, the focal point in learning is not the descriptions of objects, but their spatial relationship. In this case we first generalize on the descriptor INSIDE.

Therefore we obtain one generalization :

- EG2 = (FORM p_o x_o)(FORM q_o k_o)(FORM r_o y_o)
(LARGE x_o)(SMALL k_o)(SMALL y_o)(INSIDE y_o x_o)
(ABOVE w_o t_o)(ABOVE u_o v_o)(MBS u_o x_o)
(MBS w_o y_o)(MBS k_o v_o)(MBS t_o k_o)
(MBS t_o v_o)(MBS t_o x_o)
(MBS v_o y_o)(MBS u_o k_o)(MBS w_o k_o)
(MBS u_o w_o)

The reader can easily verify that EG2 is more general than E1, E2 and E3, but that it is not comparable from a generalization point of view, with EG1.

BIBLIOGRAPHY

- [Bongard 70] Bongard, N., "Pattern Recognition", Spartan Books, New York, 1970.
[Clavieras & Al. 83] Clavieras, B. and Ganascia, J.-G., Lemerle-Loisel, R., "Généralisation de différences entre descriptions", Actes Journées sur les outils de l'apprentissage à partir

d'exemples, Orsay, 1983.

- [Costa 81] Costa, E., "Dérécursorisation automatique en utilisant des systèmes de réécriture de termes, Thèse de 3ème cycle, Paris, 1981, Pub. by L.R.I., Bât. 490, F 91405 ORSAY CEDEX, Pub. Int. n° 118, 1982.
- [Dietterich & Al. 81] Dietterich, G.T. and Michalski, S. R., "Inductive learning of Structural Descriptions : Evaluation Criteria and Comparative Review of Selected Methods", Artificial Intelligence 16, pp. 257-294, 1981.
- [Hayes-Roth & Al. 78] Hayes-Roth, F. and Mc Dermott, J. "An interference matching technique for inducing abstractions", Com. ACM, 21, pp. 401-411, 1978.
- [Hsiang 82] Hsiang, J., "Topics in automated theorem proving and program generation", Ph. D. thesis, Dept. Comp. Sci. Univ. Illinois, Urbana, novembre 1982.
- [Kodratoff 82] Kodratoff, Y., "Generalizing and particularizing as the techniques of learning", Proc. Internatl. Conf. on A. I. and Information-Control of Robots, Bratislava, pp. 131-134, 1982 ; see also the same, submitted to Computers and Artificial Intelligence.
- [Martelli & Al. 82] Martelli, A. and Montanari, U., "An efficient Unification", Algorithm. A.C.M. Vol. 4, n° 2, 1982.
- [Michalski 83] Michalski, R.S., "Inductive learning as rule-guided transformation of symbolic descriptions : a theory and implementation", Proc. Internatl. Workshop on Program Construction, Bonas, 1980, to appear in "Automatic program construction techniques", A. Biermann, G. Guiho, Y. Kodratoff eds., Mc Millan, 1983.
- [Plotkin 70] Plotkin, G. D., "A note on inductive generalization", in Machine Intelligence B. Meltzer & D. Michie eds., American Elsevier, New York, 1970.
- [Sallantin 83] Sallantin, J., "Méthodologie pour la construction de règles à partir de données binaires", Actes Journées sur les outils de l'apprentissage à partir d'exemples, Orsay, 1983.
- [Shank 82] Shank, R. C., "Looking at Learning", Proc. ECAI-82, pp. 11-18, Orsay, 1982.
- [Stickel 81] Stickel, M. E., "A unification algorithm for associative-commutative functions", J. ACM 28, PP. 423-434, 1981.
- [Vere 80] Vere, S.A., "Multilevel counterfactuals for generalizations of relational concepts and productions", Artificial Intelligence, J. 14, pp. 139-164, 1980.
- [Vere 81] Vere, S.A., "Constrained N-to-1 generalizations", unpublished draft, pp. 23, feb. 1981.

ESCAPING BRITTLINESS

John H. Holland
 Dept. of Computer and Communication Sciences
 The University of Michigan
 Ann Arbor, Michigan 48109

ABSTRACT

Expert systems are powerful when working within the domain-specific boundaries of the initial design. It is widely agreed that the usefulness of such systems would be greatly enhanced if they could be made more versatile or less brittle -- tolerant of changes in the domain and underlying model. This paper suggests seven criteria for escaping brittleness [combination, parallelism, declarative and procedural information, categorization, synchronic and diachronic pointing, gracefulness, and confirmation] and gives an example of a class of general-purpose systems, *classifier systems*, that attempt to meet these criteria.

Key terms: Apportionment of credit, Brittleness, Bucket-brigade algorithm, Classifier system, Expert system, Genetic algorithm, Knowledge structure, Learning, Parallelism, Production system.

Expert systems are one of Artificial Intelligence's real successes in the exploration of intelligence. In the longer view, the most significant part of this success may be the bright illumination it throws upon questions of system versatility. Expert Systems, as currently structured, are narrowly domain-specific and brittle in the sense that they cannot respond to slight changes in domain without substantial human intervention (see Duda and Shortliffe [1983]). A close comparison of the structures of Expert Systems with what we know of the structures of natural intelligence provides some interesting criteria for escaping brittleness:

1. Combination: The knowledge structures and models used should be constructed from a well-defined set of elements that can serve as "building blocks". The set of elements may be rich and varied, but it should be clear how the elements can be combined to yield the structures needed. This requirement, if properly implemented, forestalls ambiguous definitions and serves as a powerful deterrent to ad hoc constructions. More importantly, it makes possible comparison of different structures in terms of the elements they hold in common. Without this possibility, discovery of similarities and analogies, transfer of information from one situation to another, and a wide range of other activities vital to flexible problem-solving, become quite difficult. Additionally, structures defined from "building blocks" possess rich, com-

binatorial possibilities for combating the complexity of the environment.

2. Parallelism: In order to avoid a distinct element (or disjoint sets of elements) for each individual object encountered (a "grandmother" node, a "yellow volkswagon with a flat tire" node, etc., etc.) it must be natural to treat the simultaneous activity of an array of elements. This is part and parcel of the discussion of combinatorics. Stated another way, it is the simultaneous invocation of a set of hypotheses at various levels of generality that enables the system to model new situations by transfer of information and conjectures from previous experience.

3. Declarative and procedural information: The "building blocks", when combined, must readily incorporate *both* the declarative ("propositional") and the procedural ("state-transition") information inherent in models.

4. Categorization: It must be simple for inductive procedures to generate and test elements, and combinations of elements, that represent the categories and other generalizations underpinning the models used. Categorization is the system's *sine qua non* for combating the environment's perpetual novelty. Accordingly, it must be easy to describe processes for generating and testing categories. Once again, unless the elements themselves directly implement categorizations, it becomes difficult to provide simple processes for transferring knowledge from one situation, or structure, to another.

5. Synchronic and diachronic pointing: Synchronic pointers -- association of categories to form analogies, hierarchies of generalization, default hierarchies, etc. -- and diachronic pointers -- temporal association yielding transitions and expectation -- are the threads from which models are constructed. As such they must be easily generated and modified by the inductive processes.

6. Gracefulness: It should be simple to insert new, tentative inductions into the overall structure without disturbing well-established procedures and models (except where the new information or induction suggests a critical modification). To this end, the elements constituting the cognitive structure should act as a network of interacting, competing hypotheses. The competition replaces requirements for global consistency with one of progressive confirmation relative to the situations in which the hypotheses are used. This

makes it much easier for the system to incorporate and use new information. Elements, and structures constructed from them, should be easily modified by combinations of bottom-up (input-driven) and top-down (category-driven) procedures. Bottom-up procedures, properly implemented in this context, generalize new information and place it in the cognitive structure at appropriate places for access and use; top-down procedures yield plausible defaults and support categories.

7. Confirmation: As the system accumulates experience the relative reliability (and usefulness) of various hypotheses (elements and combinations of elements) must be readily adjustable and must reflect itself directly in the competition of hypotheses. There is also a second kind of confirmation that must emerge from the combination of elements. A hypothesis based on a very general category, and tentatively invoked in some situation, should "support" and activate more specific conjectures that are to be tested and used to confirm, or disconfirm, the more general hypothesis.

CLASSIFIER SYSTEMS

Classifier systems are general-purpose programming systems designed as an attempt to meet the criteria set forth in the preceding section. Classifier systems have many affinities to the rule-based (production) systems underpinning the usual approach to Expert Systems (see Waterman and Hayes-Roth [1978] for many examples), but with the following major differences:

1. Any number of rules, called *classifiers*, can be active at the same time. There can be no direct conflict between classifiers because the only action of a classifier is to post a message to a global message list -- more active classifiers simply mean more messages on the list. The resulting conflict-free concurrency sidesteps the difficult conflict-resolution problems of one-rule-at-a-time systems (see McDermott and Forgy [1978]), allowing the system to use many rules concurrently to summarize and act upon a situation. The rules become "building blocks" with extensive possibilities for combination and recombination. Moreover, the parallelism makes it easier to specify and control the parallel processes that pervade the real world.

2. Messages are binary strings of fixed length k . Classifiers, following the usual convention for productions, consist of a *condition* part and an *action* part. Because the system only processes messages of fixed length, each condition has a uniform format, specifying the set of messages that can satisfy it. Specifically, conditions are all specified by strings of length k over the alphabet $\{1, 0, \#\}$ where the $\#$ plays the role of a "don't care" symbol. If $M = m_1 m_2 \dots m_k$ is a message and $C = c_1 c_2 \dots c_k$, $c \in \{1, 0, \#\}$, is a condition, then the message M satisfies condition C just in case $m_j = c_j$ wherever $c_j = 1$ or 0 , and m can be either 1 or 0 wherever $c_j = \#$. This definition makes it possible to use a simple matching operator to test whether or not some message satisfies a condition. Despite these specializations, classifier systems are provably general-purpose -- able to execute any algorithm that can be defined in a language

such as LISP or C.

3. When the condition part of a classifier (one or more conditions) is satisfied by some message on the global message list, the action part uses the message to form a new message which is posted to the list. Thus the basic procedure of the system is a simple loop in which each classifier accesses the current message list, determines if its conditions are satisfied and, if so, posts a message to the message list for the next time-step. Since messages are simply uninterpreted strings (taking their only meaning from the rules they activate) there can be no conflict in the posting of messages -- any number of distinct messages can be posted on the list for $t+1$.

4. All external communication (input and output) is via messages to the message list. As a result, all internal control information and external communications reside in the same data structure.

5. Because of the global nature of the message list, tagging (the use of particular bit combinations as prefixes, suffixes, etc.) and related techniques become efficient ways of coupling classifiers, providing predetermined sequences of execution, subroutines, hierarchical organization, diachronic and synchronic pointing, etc. The combination of concurrency and a global list avoids the difficulties discussed by Davis and King [1977] in their review of production systems.

6. Because the order in which classifiers are executed is independent of their order in storage, and because satisfaction of conditions is determined by simple matching, there is no need for an interpreter. This makes it possible to design local, syntactic operators that modify systems in useful ways. For example, tags can be modified in ways that gracefully change the behavior of the system. In particular, a system can gracefully accept new rules, and various kinds of learning procedures (see LEARNING PROCEDURES below) can be imposed, without disturbing well-established (useful) procedures.

This system can be extended in a number of ways that make it easier to use (e.g., adding not-conditions, message processing, etc. -- for details the reader should see Holland [1980]). These additions are important, but the essence of the system is the parallelism that permits both a "building block" approach to the processing of information and a competition between rules. Successful full-scale tests of classifier systems have been run at several institutions (see, for example, Booker [1982], Goldberg [1983], Smith [1980], or Wilson [1982]).

LEARNING

Two kinds of learning algorithm are required if a classifier system is to adapt to changes in the domain and goals presented it. The first is an algorithm that reinforces or apportions credit to rules already available to the system. (See Samuel's [1959] cogent discussion of this problem.) The second is an algorithm for generating new, plausible rules when the ones available prove inadequate. (Samuel calls this "the parameter prob-

lem"). This section describes one example of the first kind, called the *bucket-brigade algorithm* and one example of the second kind, called the *genetic algorithm*.

The bucket brigade algorithm is designed to assign a value to each classifier, called its *strength*, that is adjusted to reflect the classifier's net usefulness as a participant in goal-seeking sequences. It is important that the algorithm do this correctly for classifiers that, early in the sequence, set the stage for later successful actions. (For example, in checkers, this is the problem of crediting an early move that makes possible a triple jump some moves later.) The algorithm works by introducing a competition between classifiers through a modification of the basic match cycle. Satisfied classifiers must compete to get their messages on the message list; each satisfied classifier makes a *bid* based on its strength and only the highest bidding classifiers place their messages on the list. The size of the bid depends not only on the strength, but also on the specificity of the classifier's conditions. (For classifier conditions, specificity is easily measured: the more don't cares (#'s) in the condition, the less specific it is.) Thus, a high bid is indicative of "relevance" (high specificity) and "past usefulness" (high strength).

Formally, when the conditions of a classifier C are satisfied, it makes a bid

$$B(C,t) = cR(C)S(C,t)$$

where $R(C)$ is the specificity, equal to the number of non-#'s in the condition part divided by the length thereof, $S(C,t)$ is the strength of C at time t, and c is a constant $\ll 1$ (say 1/8 or 1/16).

The winning (high) bidders place their messages on the message list and have their strength reduced by the amount of the bid.

$$S(C,t+1) = S(C,t) - B(C,t)$$

(Each winner pays for the privilege of posting a message.) The classifiers {C'} that sent the messages matched by this winner have their strengths increased by the amount of the bid (sharing among them).

$$S(C',t+1) = S(C,t) + aB(C,t),$$

where $a = 1/(\text{no. of members of } \{C'\})$. (The senders are rewarded for setting up a situation usable by C.)

The bucket-brigade algorithm treats each classifier as a kind of "middleman" in a complex economy, the strength of a classifier measuring its ability to turn a "profit". As a middleman, the classifier only deals with its suppliers -- the classifiers that send messages satisfying its conditions -- and its consumers -- the classifiers that are in turn satisfied by the messages it sends. Whenever a classifier wins a bidding competition, it initiates a transaction in which it pays out part of its strength to its suppliers, and then receives similar payments from its con-

sumers. Its strength is a kind of capital. If a classifier receives more from its consumers than it paid out, its strength is increased. This is likely to occur only if the consumer in turn is profitable. This chain leads to the ultimate consumers, the classifiers that directly attain goals and receive payoff directly from the environment. That is, certain actions are immediately rewarded or reinforced by the environment; this *payoff* for goal-attainment is added to the strengths of all classifiers active at that time. The profitability of other classifiers depends upon their being coupled into sequences leading to payoff. Thus, the bucket-brigade assures that early-acting, "stage-setting" classifiers receive credit if they (on average) make possible later, overtly rewarding acts.

(It is worth noting that some of the fixed-point theorems of mathematical economics provide a way of proving the above for environments that have "stable" statistics.)

Once we can assign strengths to classifiers we have a basis for generating new classifiers to enter the competition. In broadest terms, the *genetic algorithm* uses high-valued classifiers as progenitors for new classifiers to be tested under the bucket brigade. Because there are many winners of the competition on each time-step, i.e., the system is still highly parallel, it is possible to test newly generated classifiers without seriously disrupting the system's repertoire in well-practiced situations (see below).

Genetic algorithms have been studied intensively through analysis (Holland [1975] and Bethke [1980] and simulation (De Jong [1975], Smith [1980], Booker [1982], and others). It is vital to the understanding of genetic algorithms to know that even the simplest versions act much more subtly than "random search with preservation of the best", contrary to the conclusions one might reach from a common misreading of genetics as a process primarily driven by mutation.

The central operator of a genetic algorithm is called *crossover*. It acts by exchanging a (randomly chosen) segment in a pair of strings. For example, it might exchange some initial portion of the condition parts of two classifiers; or it might exchange the middle portion of a condition with the middle portion of a message specification. This simple, seemingly non-mathematical, operator produces quite subtle effects under the competition induced by the bucket brigade. In effect, the genetic algorithm favors the construction of new rules from "building blocks" (called *schemata*) that appear as components of classifiers already proved useful. The "parent" rules simply supply copies of their parts for use by the crossover operator; they remain in the system in their original form.

The "building blocks" or schemata, so manipulated by the genetic algorithm, may involve tags or other portions of messages that determine classifier coupling and control sequencing. Thus, the genetic algorithm can encourage variants of useful subroutines and it can generate hierarchical substructures for testing. Theoretical work

and simulations (Holland [unpublished]) make it clear that newly generated classifiers can be introduced to the system without seriously disturbing capabilities already well-established. Classifiers with high strength will always score high in any competition in which they are relevant. In such a competition a newly generated classifier (starting, say, with average strength) will be quite unlikely to be a winner. On the other hand, new rules get their chance when no high-strength rule is very relevant to a situation; then the new rule may be tried and rewarded, or not, by the bucket brigade. This quality of the competition, combined with the classifier system's ability to act coherently without maintaining global consistency, permits the system to maintain a wide range of competing, plausible inductions. The system continually modifies its knowledge structures on the basis of subsequent (dis-) confirmations, while exploiting well-established structures in familiar situations.

[Classifier systems are currently being tested with the help of a program, CSI, that is both a "compiler", allowing one to design and simulate classifier systems on a standard computer, and a "test-bed", providing the means of simulating a wide range of environments for testing the learning algorithms. The current version provides the following facilities: (1) Simulation of a task environment consisting of up to 256 objects, each with up to 32 distinct features, emplaced on a 65,000 by 65,000 grid. Any, or all, of the objects may be mobile. (2) An input interface that consists of an arbitrarily shaped "vision cone" that views a local part of the surface (typically less than 1000 grid points) and uses feature detectors to generate an input message for each object in the vision cone. (3) An emulator for the classifier system that can retain, in random access memory, the description of over 2000 classifiers and a message list of up to 32 messages. This part of the system is written in machine language and can execute a basic time-step (all classifiers matched against all messages) in about 0.1 second. The overall system runs in close to real-time, making it convenient to run long learning sequences in the simulated environment. (4) An output interface that permits manipulation of objects on the grid, movement over the grid, rotation of the visual cone, and in fact any other effector action conveniently specifiable by a subroutine. (5) Parametrised versions of both the bucket-brigade and genetic algorithms, including provisions for contingent activation of these algorithms (such as activation of the genetic algorithm when no classifier responds, to an input message, with a high bid).]

REFERENCES

- Bethke, A. D., *Genetic Algorithms as Function Optimizers*, Ph.D. Dissertation (Computer and Communication Sciences), The University of Michigan, 1980.
- Booker, L., *Intelligent Behavior as an Adaptation to the Task Environment*, Ph.D. Dissertation (Computer and Communication Sciences), The University of Michigan, 1982.
- Davis, R. and King, J., "An Overview of Production Systems", *Machine Intelligence 8*, Elcock, E. W. and Michie, D. (Eds.), American Elsevier, New York, pp. 300-331, 1977.
- DeJong, K. A., "Adaptive System Design -- A Genetic Approach", *IEEE Transactions: Systems, Man, and Cybernetics*, Vol. 10, No. 9, 1980.
- Duda, R. O. and Shortliffe, E. H., "Expert Systems Research", *Science*, Vol. 220, pp. 261-268, 1983.
- Goldberg, D., [Dissertation in progress - Civil Engineering], The University of Michigan, 1983.
- Holland, J. H., "Adaptive Algorithms for Discovering and Using General Patterns in Growing Knowledge Bases", *International Journal of Policy Analysis and Information Sciences*, Vol. 2, pp. 217-240, 1980.
- McDermott, J. and Forgy, C., "Production System Conflict Resolution Strategies", *Pattern-Directed Inference Systems*, Waterman, D. A. and Hayes-Roth, F. (Eds.), Academic Press, New York, pp. 177-199, 1978.
- Samuel, A. L., "Some Studies in Machine Learning Using the Game of Checkers", *IBM Journal of Research and Development*, Vol. 3, pp. 211-232, 1959.
- Smith, S., *A Learning System Based on Genetic Algorithms*, Ph.D. Dissertation (Computer Science), The University of Pittsburgh, 1980.
- Waterman, D. A. and Hayes-Roth, F. (Eds.), *Pattern-Directed Inference Systems*, Academic Press, New York, 1978.
- Wilson, S., "Adaptive 'Cortical' Pattern Recognition", [Internal Report, Research Laboratories, Polaroid Corporation], 1982.

TWO PROGRAMS FOR TESTING HYPOTHESES OF ANY LOGICAL FORM

Clark Glymour

Kevin Kelly

Richard Scheines

Department of History and Philosophy of Science
 1017 Cathedral of Learning
 University of Pittsburgh
 Pittsburgh, PA 15260

ABSTRACT

Programs for machine learning from instances or "structural descriptions" have been limited by restrictions on the logical form of the conclusions inferred. We describe two programs for testing hypotheses of any logical form. The first implements a qualitative confirmation relation due to Hempel. The second, and more useful, implements an approximation to any of several quantitative confirmation relations derivative from Hempel's.

I. INTRODUCTION

Many machine learning programs attempt to infer generalizations from data represented or straightforwardly representable by singular sentences in first-order logic. Such learning programs typically generate conclusions that can also be represented as first-order sentences, usually with some quantifier structure. The conclusions that can be drawn using such programs are inevitably limited by whatever restrictions the program imposes on the logical form of hypotheses, and in most cases these limits are severe. Hunt's (1966) "Concept Learning Systems", for example, were in effect restricted to hypotheses with monadic predicates, a single universal quantifier, and a biconditional form with a single predicate on one flank of the biconditional. A program described by Vere (1975) is restricted to conjunctions with universal quantifiers. An impressive program due to Shapiro (1981) is restricted to Horn Sentences, that is, to universal conditions with conjunctive antecedents and atomic consequences. These restrictions essentially limit the expressive power of the conclusions that can be inferred by the respective systems. In our judgement, an important aspect of the progress of machine learning programs in the last twenty years has been the liberalization of restrictions on the logical form of conclusions, but existing programs cannot yet generate from instances hypotheses that are not subject to prior restrictions on logical form, and even hypotheses with so simple a form as $VxEy(Pxy \vee Pyx)$ seem excluded at present.

A computer program for efficiently testing (or confirming) hypotheses of arbitrary logical form would be a significant step towards a

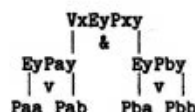
machine induction program with the same generality. We describe below two programs which, given an hypothesis H of any first-order logical form and given an instance or evidence sentence E which is a conjunction of atomic sentences or their negations, determine whether or not E confirms H . The formal ideas behind these programs derive from the philosophical literature on inductive logic, and especially from the work of Carl Hempel. Coupled with an appropriate generator, on which we are now working, the second of these programs offers the prospect of significantly liberalizing the restrictions on logical form in machine learning. The programs described below are written in MACLISP and are implemented on the computer facilities of Carnegie-Mellon University.

II. THE HEMPEL PROGRAM

The first account of confirmation that we have implemented is due to C.G. Hempel (1965a), who developed it for first-order languages without identity, although it has subsequently been extended by Glymour (1980) to languages with identity and definite descriptions. Hempel's confirmation relation is purely qualitative, that is, a singular sentence either confirms an hypothesis or it doesn't, and confirmation has no other degrees. Hempel's account of confirmation satisfies several formal conditions, the most important of which for our purposes is that if an evidence sentence confirms an hypothesis then it confirms all hypotheses logically equivalent to that hypothesis, and analogously for disconfirmation. Let E be a singular sentence in which the individual constants $a_1, \dots, a_n$ occur essentially, and let H be any first-order sentence without identity. According to Hempel, E confirms H if and only if the sentence H is valid in every model of E in which all members of the domain are named by one of the constants $a_1, \dots, a_n$. Thus Paa confirms each of the following: $VxPxx$, $VxVyPxy$, $VxEyPxy$, etc., but does not confirm $ExEyPxy$. (The evidence sentence need not be conjunctive in Hempel's theory, but it seems almost always to be assumed to be conjunctive in machine learning problems, and we will keep that restriction throughout.) Evidence E fails to confirm H if E does not confirm H ; E disconfirms A if E confirms the denial of H .

Hempel shows that E confirms H if and only if E logically entails a certain variable-free sentence determined by H and the individual

constants occurring essentially in E. This sentence, the development of H for $a_1, \dots, a_n$, can be obtained by iteration starting with the first quantifier occurring in the formula, deleting that quantifier and, if the quantifier is universal, conjoining all formulas obtained by substituting one by one the constants $a_1, \dots, a_n$, for all occurrences of the freed variable, or if the quantifier is existential disjoining the substitution instances. Equivalently, the development can be represented as a tree with the hypothesis as root, each vertex other than the root and leaves corresponding either to a conjunction or to a disjunction, and atomic sentences or their negations (basic sentences) as leaves. So represented, the development of $\forall x E y P x y$ for the constants a, b is



Our program estimates Hempelian confirmation provided three conditions are met: the evidence sentence is a singular conjunction; the hypothesis to be tested is expressed so that no quantifier occurs within the scope of a negation; and the hypothesis to be tested contains no predicates not occurring in the evidence sentence. The program executes a depth first search of the development tree of the hypothesis, returning "T" for a leaf if the corresponding formula is found among the conjuncts of the evidence sentence, and "nil" otherwise. Vertices whose daughters are obtained by instantiating an existentially quantified variable are counted as disjunctive vertices, and analogously for universal quantifiers and conjunctive vertices. A disjunctive vertex is evaluated as "T" if and only if at least one of its daughters is evaluated as "T", and a conjunctive vertex is evaluated as "T" if and only if all of its daughter vertices are evaluated "T". Negations are distributed so as to have the smallest possible scope. The depth first search strategy is such that whenever the fragment of the tree already evaluated suffices logically to determine a truth value for the hypothesis to be tested (under the valuation determined by the evidence sentence), the generation and evaluation of the development tree cease and the appropriate value, T or nil, is returned. "nil" signifies that the hypothesis is not confirmed; it does not necessarily mean that the negation of the hypothesis is confirmed.

The program contains a parameter m that can be adjusted by the user when large bodies of evidence or hypotheses with many quantifiers are to be considered. The parameter can be set to any value less than or equal to the number of individual constants occurring in the evidence sentence. With the value of m less than the number of individual constants, the program behaves as described above, except that at each quantifier vertex, there are only m randomly selected daughter vertices. Instead of

instantiating each quantified variable with all of the individual constants, only m randomly selected constants are used. The selection of the m constants is performed independently at each quantifier vertex. When m has a value less than the number of individual constants the program output is no longer certain to be the result that Hempel's confirmation theory would give, but is only an estimate of that result. Coincidentally, of course, the program can return the "correct" result even though some of the data are in error.

When the program is given as evidence all conjuncts $P a_i a_{j+1} \& \dots \& P a_i a_j$ with $i, j = 1, 2, 3, 4, 5$ and $j \neq i+1$, the hypothesis characteristic of discrete order, $\forall x \forall y (P x y \rightarrow \neg E z (P x z \& P z y))$ is confirmed in 0.4 seconds with $m = 5$, and in 0.27 seconds with $m = 2$. The same instance fails to confirm the hypothesis characteristic of dense order and likewise fails to confirm transitivity. With $m = 5$ these determinations require 0.24 and 0.44 seconds respectively.

Canfield (1962) pointed out, and Hempel (1965a) himself agrees, that Hempel's confirmation theory is excessively strict. It fails to allow confirmation by partial instances. Thus $P a a \& P a b \& P b a$ fails to confirm $\forall x \forall y P x y$ because the conjunct $P b b$ does not occur in the evidence. The computer program shows the same feature. The evidence sentence consisting of all conjuncts $P a_i a_j$ with $i, j = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10$ and $i < j$ seems as though it ought to lend some support to the hypothesis of transitivity -- that is, to $\forall x \forall y \forall z ((P x y \& P y z) \rightarrow P x z)$ -- but in fact the hypothesis is not confirmed with $m = 10$. (The decision takes about 0.3 seconds). With $m = 2$ twelve trials produced no confirmation of the hypothesis and with $m = 1$ seven of thirteen trials produced no confirmation. Hempel's confirmation theory is too conservative to be useful.

III. THE PARTIAL INSTANCES PROGRAM

There are many formal ways in which to develop a confirmation measure that will extend Hempel's ideas by giving some weight to partial instances. One can let the degree of confirmation of H on E equal the proportion of the models of E (in which each element of the domain is named by an individual constant occurring essentially in E) that are also models of H. The result is a conditional probability measure, but it has the unattractive consequence that the degree of confirmation of a universal hypothesis cannot be greater than 1/2 unless it is 1. We develop a different idea, deriving from the intuition that since the evidence $P a a \& P a b \& P b a$ provides three of the four conjuncts necessary for Hempelian confirmation of the hypothesis $\forall x \forall y P x y$, the degree of confirmation of that hypothesis on that evidence should be 3/4. One formal measure that captures this intuition is obtained by putting the development of H in full conjunctive normal form, call it H^* . In H^* , give a sentential component that is either atomic or the negation of an atomic formula the score 1 if that component is among the conjuncts of E, give it the score 0 if its negation is among the conjuncts of E, and give it

the score 0 otherwise; give a disjunction the score 1 if at least one of its disjuncts has the score 1, give it the score * if all of its disjuncts have the score *, and give it the score 0 otherwise; give a conjunction (that is, H^*) a degree of confirmation equal to the proportion of its conjuncts that have the score 1, provided none of its conjuncts have the score *, and otherwise give it degree of confirmation 0. Another measure of the same sort can be obtained by considering the minimal conjunctive normal forms of H and, because the minimal conjunctive normal form is not always unique (Hohn, 1966), applying some choice function. Still other measures can be obtained from the disjunctive normal forms. All of these measures give about the same result in the simple and intuitive cases, and they all have the same essential feature, namely they are not probability measures. The sum of the confirmation of H on E and the confirmation of $\neg H$ on E , need not be unity by any of these measures. They are closer to Shafer's (1976) belief functions than to probabilities. We find no powerful theoretical reason for preferring any one of these normal form measures to the others, and any of them will, unlike Hempel's theory of confirmation, permit partial instances to give some weight to hypotheses.

It is not practical actually to determine conjunctive normal forms of developments of hypotheses, and the Partial Instances Program we have implemented is only a crude approximation to any of the measures described above. The program distributes negations so as to give them the smallest scope or cancel them, and it distributes conjunction over disjunction, but it does not convert a formula such as $(p \vee q) \wedge (\neg p \vee r)$ into the equivalent but shorter $(q \vee r)$. Because of the scoring procedure, the Partial Instances Program, unlike the Hempel Program in all but the worst case, must typically search the entire development. Computation times are therefore significantly longer. The same sampling strategy is used in the Partial Instances program as is used in Hempel. The scoring procedure approximates the normal form measures by requiring that the confirmation of a disjunction is the confirmation of the best confirmed disjunct, by requiring that the confirmation of a conjunction is in the simplest case the ratio of confirmed conjuncts to unconfirmed conjuncts, and by working recursively on the logical structure of the development of the hypothesis to be tested. More exactly, the scoring procedure of the Partial Instances Program is as follows: a leaf (an atomic sentence or its negation) is scored (1,1) if it is among the conjuncts of the evidence sentence, * if its negation is among the conjuncts of the evidence sentence, and (0,1) otherwise; a conjunctive node is given the vector sum of the score of its daughter nodes provided none of the latter are scored *, and is given the score * otherwise; a disjunctive node is given the score x/y if x/y is the maximum ratio of the components of the scores of daughter nodes, provided some daughter node has a value other than *, and is given the score * otherwise. A

score * at the root indicates that the hypothesis has been disconfirmed in Hempel's sense, i.e., that its negation has been confirmed to degree 1.

When the Partial Instances Program is given the evidence sentence consisting of all conjuncts $P_{a,j}$, with $i, j = 1, 2, 3, 4, 5, 6, 7$, and $i < j$, the hypothesis $VxVyVz (P_{xy} \wedge P_{yz}) \rightarrow P_{xz}$ is given degree of confirmation $5/12$ with $m = 7$ in 6.4 seconds. The price of avoiding the computation of normal forms is that the program gives different degrees of confirmation to logically equivalent hypotheses, on the same evidence. We believe that this incoherence will prove harmless so long as hypotheses are generated for testing in a standard form. No loss in the possible expressive power of conclusions generated by a machine learning system is entailed by restrictions on logical form, if every hypothesis is logically equivalent to an hypothesis of the form permitted. We are at work on an hypothesis generator which will consider in order of simplicity open formulas that are instantiated at least once by any model of the evidence sentence. For each open sentence, the sentences in prenex form that can be obtained from it by prefixing quantifiers will be tested, one by one. Control of acceptance of hypotheses and cessation of generation and testing will be governed by a combined index of degree of confirmation and explanatory power, where the latter will be based on a computational approximation of a measure developed by Hempel and Oppenheim (1965b).

REFERENCES

- Canfield, J. "On the Paradox of Confirmation", *Metrika*, 5, pp. 105-18, 1962.
- Glymour, C. *Theory and Evidence*. Princeton: Princeton University Press, 1980.
- Hempel, C. "Studies in the Logic of Confirmation", *Aspects of Scientific Explanation*, Glencoe: The Free Press, 1965a.
- Hempel, C. and Oppenheim, P. "Studies in the Logic of Explanation", *Ibid*, 1965b.
- Hohn, F. *Applied Boolean Algebra*. New York: Macmillan, 1966.
- Hunt, E. *Experiments in Induction*. New York: Academic Press, 1966.
- Shafer, G. *A Mathematical Theory of Evidence*. Princeton: Princeton University Press, 1978.
- Shapiro, E. *Inference of Theories from Fact*. Research Report 192, Yale University, 1981.
- Vere, S. "Induction of Concepts in the Predicate Calculus", 4th IJCAI Proceeding, 1975.

Research for this report was supported by the National Science Foundation and the University of Pittsburgh.

LEARNING EQUATION SOLVING METHODS FROM WORKED EXAMPLES

Bernard Silver
 Dept. of Artificial Intelligence
 University of Edinburgh
 Hope Park Square
 Meadow Lane
 Edinburgh EH8 9NW, U.K.

ABSTRACT

This paper describes LP, a program that learns new techniques for solving equations by examining worked examples. Unlike most of the work in this field, where the equations used have been very simple, LP uses equations of A level standard. (A levels are exams taken at 18, and are used for university selection.)

In order to be able to successfully use a new technique, LP learns many different types of information. At the lowest level, LP compares consecutive lines in the worked example, finding differences between them. This allows the program to learn new rewrite rules.

LP also tries to discover the strategic purpose of each step, expressed in terms of satisfying preconditions of following steps. From this viewpoint, the worked example can also be considered as a type of plan for solving the equation. LP extracts the necessary information, and builds a plan which is stored for future use. LP executes the plan in a flexible way to solve new equations.

Key terms: Learning, learning by example, strategy learning, equation solving, planning, meta-level inference, PROLOG.

I. INTRODUCTION

This paper describes LP, a program which learns new techniques for solving symbolic equations. LP learns the new techniques by examining worked examples. These techniques can then be tested on some new problems. LP is built around the equation solving program PRESS, [Bundy and Welham 81, Sterling et al 82], and is intended to be part of a self-improving algebra system, [Bundy and Sterling 81].

The equations used by LP are symbolic, transcendental, non-differential equations, e.g.

$$\log_2 x + 4. \log_x 2 = 5.$$

1.1. Motivation

The motivation for LP arose from our equation solving program PRESS, [Bundy and Welham 81, Sterling et al 82]. This is a non-learning program, the only way to increase the ability of PRESS is to write new code. We have a test set of equations, and each version of PRESS is primarily judged by how many of these equations it can solve. Over several years, new equation solving methods have been added to PRESS, and its performance has markedly increased. However, coding and debugging the new methods can be tedious, and we set ourselves the goal of building a program which could learn new methods automatically.

Other workers have dealt with learning equation solving methods, but our interests are somewhat different.

Most of the other work in this field, e.g. [Brazdil 78, Neves 78, Cotton et al 81, Langley 82], have concentrated on very simple equations, such as

$$2.x + 3 = 5.$$

The machine learning community have undoubtedly found such work interesting. It has been demonstrated that computer programs can learn to solve equations, building up from very little knowledge. Also, some of this work may prove interesting to educationalists, particularly the programs that attempt to model childrens' errors.

However, not only beginners learn, so do experts and people in between these two extremes. We required a learning program with abilities nearer the expert end of the spectrum. While it would be nice for the program to build up to expert performance from a beginners knowledge, we felt that such an approach would be impractical. A more realistic goal is to build a program that starts with abilities around the "fairly good" mark, and learns to the standard of "very good".

LP is able to learn a new method by examining a worked example which uses the new technique. We feel that a single example should be sufficient. As Neves points out in [Neves 78], many textbooks present just one worked example before proceeding to the exercises.

This consideration rules out many concept-learning methods, e.g. [Winston 75, Mitchell et al 81, Young et al 77], as these programs generally need to process several instances before a concept is learned.

1.2. Worked Examples

A worked example should contain enough detail so that the reader (either a human or a program) can understand the technique being demonstrated, but there should not so much detail that the important points are swamped.

We now give a typical instance of the type of worked example used by LP:

$$\cos(x) + 2.\cos(2.x) + \cos(3.x) = 0 \quad (i)$$

$$2.\cos(2.x).\cos(x) + 2.\cos(2.x) = 0 \quad (ii)$$

$$2.\cos(2.x).(\cos(x) + 1) = 0 \quad (iii)$$

$$\cos(2.x) = 0 \text{ or } \cos(x) + 1 = 0$$

$$\cos(2.x) = 0$$

$$x = 180.n1 + 90$$

$$\cos(x) + 1 = 0$$

$$\cos(x) = -1$$

$$x = 180.(2.n2 + 1)$$

The key step in this worked example is the transformation from equation (i) to equation (ii).

II. THE METHODS OF LP

In this section we describe the basic operators of LP, called methods. These are modified PRESS methods.

Equations are solved by applying methods, which in turn may apply rewrite rules. An example of a method is Collection, which applies to equations containing more than one occurrence of the unknown. Collection reduces the number of occurrences of the unknown.* Collection is used in the worked example above to obtain line (iii) from line (ii). This step uses the Collection rule

$$U.V + U.W \Rightarrow U.(V + W).$$

Two other methods that will be referred to later are Factorization and Change of Unknown. Factorization transforms equations of the form

$$E_1.E_2 \dots E_n = 0$$

into

$$E_1 = 0 \vee E_2 = 0 \vee \dots \vee E_n = 0,$$

and solves each factor separately.

The Change of Unknown method transforms equations of the form

$$f(g(x)) = 0,$$

where g occurs at least twice in f , into

$$f(y) = 0, \quad (iv)$$

where

$$y = g(x).$$

The equation (iv) is called the changed variable equation. If it is solved, producing the answer $y = a$, say, then solving the equation

$$g(x) = a \quad (v)$$

solves the original equation. Equation (v) is called the substitution equation.

A method consists of a set of rules applicable to that method, and some information telling when and how the rules should be applied. For the case of rule-learning, in [Bundy & Silver 82], we call this control information. The control information includes preconditions, facts which must be true before the method can be applied, and postconditions which must be true after the application of the method. The postconditions are used both to ensure that the method has achieved the desired effects, and to allow planning.

Many learning programs work in domains where the basic operators are similar to those of STRIPS. The operators of STRIPS, [Fikes et al 72], have preconditions, an add list which contains the facts that the operator makes true, and the delete list which contains the facts that are no longer true after the application of the operator. If the preconditions are satisfied the operator can be applied.

In contrast, LP methods do not have this desirable property. The preconditions of a method represent necessary, but not sufficient conditions. In general, a method is not certain to succeed, even if the preconditions are applicable. This is because the preconditions are too general, but we can not give stronger preconditions that do not involve actually applying the method to test if it is applicable! It seems that this might be a problem in many domains.

*If the number can be reduced to one, another method, Isolation, can be applied to solve the equation.

Similarly, the effects of a method are hard to classify. The postconditions are used to specify what should be true after a method has been applied in the desired way, but there is no guarantee that the method will produce these effects. It is also difficult to say which facts will no longer be true.

2.1. What LP needs to Learn

In order to learn a "new technique", LP may need to learn at several levels. At the lowest level, it may need to learn new algebraic identities. These will be used as rewrite rules. We are not particularly concerned with this low level task.

At the next level up, LP needs to learn new operators, i.e. methods. This will involve learning the control information for the method, and associating with it some rewrite rules. These rewrite rules may be new or old rules. The control information does not depend on which equation is being used.

However, LP needs also to learn the meta-control information, which controls the order in which methods are used. This information is recorded in a plan, called a schema. The schema records how the equation is solved, and can be used to solve new equations. The plans may be equation dependent. For example, the plan may record that the solution involved applying method M followed by method N. In the general case however, even if method M can be applied, it may not be the case that method N will then be applicable.

III. LEARNING FROM EXAMPLES

LP learns by examining worked examples. LP is given a worked example that shows how a particular equation is solved. We will call this equation the generating equation.

3.1. Step Justification

The first task for LP is to discover how each line in the worked example is transformed into the next. We call this step justification. To do this, it examines consecutive pairs of lines, trying to find the method that transforms each line to the next.

Suppose that LP is working on the step from a line p to the next line, called q. LP first tries to see if an existing method can account for the step. To do this, LP computes the characteristic tuple, henceforth CT, for each of the lines p and q.

The CT is a tuple of meta-level characteristics of the equation, consisting of facts such as the number of occurrences of the unknown, the type of function symbols occurring in the equations, e.g. trigonometric, whether the equation is a single equation or a

disjunction etc. For example, the CT of

$$\cos(x) + 2.\cos(2.x) + \cos(3.x) = 0$$

is

[occ(multiple,3), trig, eq, functor(+,2)],

meaning that there are multiple occurrences of the unknown, 3 in all. The expression is trigonometric, it is an equation (rather than a disjunction), and the dominating functor on the lhs is "+" of arity 2.

LP then looks for a method that can transform the CT of p into the CT of q.* Use of the CT to constrain search is an example of the technique of meta-level inference that is used extensively in the PRESS, IMPRESS ([Bundy and Sterling 81]), and LP projects.

If it finds a method, it attempts to use it to transform p into q (a method may not apply even if its CT and other control information indicate that it is suitable). If the method is successful, LP records that the step from p to q was performed by that method, and proceeds to the step from q to the next line. Otherwise, it tries to find another possible method. If no more possible methods can be found, LP conjectures a rewrite rule that would explain the step.

The rewrite rule is obtained by removing common terms** from p and q and equating the remaining terms. Neves, [Neves 78], uses a similar technique. For example, consider the lines (i) and (ii) in the worked example shown in section 1.2. Both lines contain the additive term $2.\cos(2.x)$ and the right hand side of both is 0. Deleting these and equating the remainder produces the conjecture

$$\cos(x) + \cos(3.x) = 2.\cos(2.x).\cos(x).$$

LP asks the user if the conjecture is correct, and if it is, the user is asked to provide the general rule. In this case, the user gives LP the rule

$$\cos(A) + \cos(B) \Rightarrow 2.\cos((A+B)/2).\cos((A-B)/2).$$

*This information is part of the control information of the methods.

**LP uses its knowledge of properties of functions to decide whether terms are common or not. For example, A can be deleted from the equations $A + B = C$ and $D + A = C$, to produce the conjecture $B = D$. However, A can not be deleted from the pair $\log_A B = C$ and $\log_B A = C$, as the two arguments of the logarithmic function do not commute.

If the rewrite rule has the same characteristics as an existing rule set*, the new rule is added to that set, otherwise LP puts it in a special set of miscellaneous rules.

Given that LP is learning from one trial, there seems to be no easy way for the program to correctly generalize rules of this sophistication. Unlike Neves, [Neves 78], it is no longer sufficient just to replace numbers with variables. However, we do not worry about this problem. As mentioned above, we are not particularly interested in the learning of rules, we are more concerned with the learning of methods and schemas.

3.2. Creating New Methods

Once every step has been processed, LP examines its analysis to see if any new methods need to be created. New methods are created to explain the application of new rules. Suppose a new rule R is applied at line i , to produce line $i + 1$. LP first finds the preconditions, P , of the method M applied at line $i + 1$, to give line $i + 2$. It then finds which of the members of P are satisfied at line i . Call these preconditions S . The set difference $P \setminus S$ is the set of preconditions not satisfied at line i , but satisfied at $i + 1$, call this set U .**

If U is non-empty, LP assumes that the purpose of applying R is to satisfy U so that M can be applied. The set U is called the major effects of the rule.

If LP finds that all the preconditions of M were satisfied, i.e. U is empty, it looks for another explanation. One possibility is that the rule is used to manipulate the equation so that M can be applied, although no new preconditions are satisfied. This kind of behaviour occurs because the methods are not STRIPS type operators (see section II).

The above analysis is performed for each application of a new rule in the worked example, causing LP to create a new method. This method applies the rule and then method M . The preconditions of the method are S , plus any preconditions of rule. The postconditions of the method are those of M .

3.3. The Division of the Worked Example

The next step involves dividing the worked examples into sections. This allows LP to

*the rules for each method have certain characteristics, e.g. Collection rules reduce the number of occurrences of the unknown.

** U must be satisfied at $i + 1$ or M could not be applied.

construct new macros and schemas. The first step is to give the worked example a type. At present, LP has three types.

If the worked example contains an application of Change of Unknown (see section II) type is chunk, otherwise, if it contains a Factorization step its type is factor. If neither of these steps occur, the type is normal.

The worked example can now be divided into sections. If the example has type chunk, it is divided into three sections. The first section contains the steps leading up to the Change of Unknown step, the second section contains the steps that solve the changed variable equation, the final section contains the steps involved in the solution of the substitution equation.

If the example has type factor, and the Factorization step produces n factors, the example will contain $n + 1$ sections. The first section contains the steps leading up to the application of Factorization. The remaining sections each contain the steps involved in solving one factor.

If the example has type normal, all the steps go into a single section.

3.3.1. Macro Formation

Each section consists of a sequence of steps. LP now creates a macro operator from the sequence. The process is similar to that for creating new methods, except that the purpose of the sequence can be described differently. Thus macros produced from the first section of a factor type example are marked as being useful for transforming an equation to a state where Factorization can be applied. The preconditions of the macro are the preconditions of the first method in the sequence, and the postconditions are the postconditions of the last method in the sequence, plus preconditions derived from the macros purpose. For example, a macro useful for Factorization would have the extra postcondition that the resulting equation is of the form

$$E_1 \cdot E_2 \cdot \dots \cdot E_n = 0,$$

so that Factorization can be applied.

The new macros are like MACROPS in STRIPS, [Fikes et al 72]. They have a similar status to normal methods, and LP can detect the use of a new method in a worked example. Macros are executed in the obvious way, each method in the sequence is executed in turn. If one of the methods fails, the macro fails.

3.3.2. Schema Formation

Now most of the work has been done, and the schema can be created easily. The schema contains four main parts. These are

- the generating equation,
- the unknown,
- the type of the worked example,
- and the body of the schema.

The body is a list consisting of all the methods used in the worked example. Each step is tagged with the conditions that it is used to satisfy, i.e. the major effects, plus any conditions that must also be maintained. The steps are divided into the divisions that were found above.

The schema is used to solve new equations.

IV. SOLVING NEW EQUATIONS

When LP is given an equation it first tries to find a schema that seems to be relevant. A schema is relevant if the equation that produced it has the same CT as the current equation. If one is found, LP tries to apply the steps listed in the schema. We first discuss the case where the schema is unsectioned, i.e. the worked example was of type normal.

Suppose that the current line in the schema suggests the application of method M. LP uses the following procedure:

1. Try to apply method M. If this succeeds, continue with the next step in the schema.
2. Otherwise, try to find another method that has the same major effects as M, and apply it. If this succeeds, continue with the next line in the schema.
3. If the two steps above both fail, try to find a method that does not undo already satisfied conditions, and apply it. If this succeeds, go to 1.
4. If none of the above steps have succeeded, and if M has no major effects, omit the step entirely, and proceed with the next schema step.

If none of these attempts are successful, LP tries to solve the equation without the schema, in a similar way to PRESS.

If the schema is divided, i.e. the worked example was not of type normal, there is one addition to the above account. Each division in the schema has a purpose. For example, the first section of a factor type schema consists of the steps that transform the equation so that Factorization can be applied. At each step LP checks for fortuitous circumstances. For example, if LP is executing the first section of a factor type schema, it checks to see if the Factorization can be applied to the current equation. If it can, LP assumes that the purpose of the section has been achieved, and ignores the remaining steps in the section. It then proceeds

to the next section.

The schema acts as a simple plan that can be executed in a flexible way.

4.1. Learning by Doing

Neves, [Neves 78], has written a program ALEX that also learns by examining worked examples. He calls this approach learning by example. ALEX can also learn during the equation solving stage, using the learning by doing technique. We have found it preferable to centralize the learning process. New schemas, macros, methods and rules are formed only when LP is analyzing a worked example. However, when LP solves an equation it produces a trace of the solution path. This trace is just a list of the equations produced during the solution process and can be used as a worked example. In this way LP can produce new macros and schemas to help it solve future problems.

4.2. Generating New Problems

Like LEX, [Mitchell et al 81, Mitchell et al 82], LP is able to create new problems for it to solve. After LP has completed its examination of the worked example, it produces a set of equations that differ in only one feature from the generating equation. The simplest change is the replacing of one numerical coefficient by another. Another consists of replacing a function by a sibling function, e.g. replacing $\sin$ by $\cos$. The problem generator allows LP to test the applicability of the schema. At present, LP simply notes if the schema allowed it to solve the test problem or not. A future version could incorporate a concept-learning mechanism that would enable modification of the schema.

V. RESULTS

LP has learned several new techniques from worked examples. One result is that LP has been able to solve difficult equations after examining much easier examples. For instance, after LP has been given worked examples for the equations

$$\sin(x) + \sin(2.x) = 0,$$

$$\sin(4.x) = \cos(2.x), \text{ and}$$

$$\cos(x) + \cos(3.x) = 0 \quad (\text{vi})$$

it is able to solve

$$\sin(2.x) + \sin(3.x) + \sin(5.x) = 0. \quad (\text{vii})$$

In the second equation the coefficient of x in the $\sin$ term* must be twice that in the $\cos$ term,

*The number 4 in our example.

but is otherwise arbitrary. In the first and third equations there are no restrictions on the choice of coefficients. The worked examples for the first three equations contain 7 or 8 lines. The solution for equation (vii) contains 14 major steps, and includes a variant of equation (vi) as a subproblem.

VI. CONCLUSIONS

In this paper we have described LP, a program that learns equation-solving techniques from worked examples. LP uses concepts from planning, analyzing strategic information in terms of satisfying preconditions.

This approach seems to be applicable to other domains. Symbolic integration seems to be an obvious choice. LEX, [Mitchell et al 81, Mitchell et al 82] learns heuristics for symbolic integration. LEX solves integrals by applying a sequence of operators. Each operator transforms the integral into another (hopefully) simpler one. This process is very similar to equation solving. It appears that LP could easily be adapted to learn how to perform symbolic integration, although at present this idea has not been tested. LEX refines heuristics, using several examples, whereas LP learns strategies from one example. Thus the two approaches are somewhat complementary.

Using concepts from planning, the technique of learning from examples seems to work well in the algebra domain, even when the equations are much harder than those considered by earlier workers, [Cotton et al 81, Neves 78].

ACKNOWLEDGEMENTS

This work is supported by S.E.R.C. grant GR/C/20826, and a S.E.R.C. studentship to the author. I would like to thank Alan Bundy, Lawrence Byrd, Jane Hesketh, Richard O'Keefe, Mary Angela Papalaskaris, Dave Plummer, Leon Sterling, Maarten van Someren, and Lincoln Wallen for their help with this work.

REFERENCES

- Brazdil, P. Experimental Learning Model, pages 46-50. Society for the Study of Artificial Intelligence and Simulation of Behaviour, 1978.
- Bundy, A. and Silver, B. A critical survey of rule learning programs, pages 150-157. European Conference on Artificial Intelligence, 1982.
- Bundy, A. and Sterling L.S. Meta-level Inference in Algebra. Research Paper 164, Dept. of Artificial Intelligence, Edinburgh, September 1981.
- Bundy, A. and Welham, B. Using meta-level inference for selective application of multiple rewrite rules in algebraic manipulation. Artificial Intelligence, 1981, 16(2), pp 189-212.
- Cotton, J., Byrd, L. and Bundy, A. How can Algebra steps be learned by students with only arithmetic skills. Working Paper 84, Dept. of Artificial Intelligence, Edinburgh, 1981.
- Fikes, R.E., Hart, P.E. and Nilsson, N.J. Learning and executing generalized robot plans. Artificial Intelligence, 1972, 3, 251-288.
- Langley, P. Strategy Acquisition Governed by Experimentation, pages 171-176. European Conference on Artificial Intelligence, 1982.
- Mitchell, T.M., Utgoff, P. E., Nudel, B. and Banerji, R. Learning problem-solving heuristics through practice, pages 127-134. International Joint Conference on Artificial Intelligence, 1981.
- Mitchell, T.M., Utgoff, P. E. and Banerji, R. Learning by Experimentation: Acquiring and modifying problem-solving heuristics. Tioga Press, 1982.
- Neves, D.M. A computer program that learns algebraic procedures by examining examples and working problems in a textbook, pages 191-195. Canadian Society for Computational studies of Intelligence, 1978.
- Sterling, L., Bundy, A., Byrd, L., O'Keefe, R., and Silver, B. Solving Symbolic Equations with PRESS, pages 109-116. Springer Verlag, 1982.
- Winston, P. Learning structural descriptions from examples. McGraw Hill, 1975.
- Young, R.M., Plotkin, G.D. and Linz, R.F. Analysis of an extended concept-learning task, pages 285. International Joint Conference on Artificial Intelligence, 1977.

ADJUSTING BIAS IN CONCEPT LEARNING*

Paul E. Utgoff**

Department of Computer Science
Rutgers University
New Brunswick, New Jersey 08903

ABSTRACT

We approach concept learning as a heuristic search through a space of concepts for a concept that satisfies the learning task at hand. The heuristics represent bias that the concept learning program employs when forming an inductive generalization. We present a model of bias adjustment and report our experience with an implementation of the model.

Key Terms: Inductive concept learning, heuristic, bias, candidate elimination, description language, assimilation, constraint back-propagation.

I. ROLE OF BIAS

A major objective in research on inductive concept learning is creation of a program that can accept training data, apply knowledge, and form inductive hypotheses of the concept, all without human intervention. The learning program searches a space of hypotheses for those that are consistent with the observed examples, and which classify the unobserved instances as indicated by heuristics. The heuristics, which we call *bias*, determine the inductive generalizations that the program will form, given some set of training examples. The initial bias may be appropriate for one learning task, yet inappropriate for another.

We see two basic approaches to using heuristics for inductive concept learning by example. The first approach is to use a relatively large space of hypotheses, and to employ bias to influence how the space is searched. An example of this approach is Michalski's STAR methodology [Michalski 83]. Michalski defines a large hypothesis space via a comprehensive description language. The program uses bias provided by hand to determine whether one consistent hypothesis is to be

preferred to another. Often, the bias is specified as a meta-statement about descriptions, e.g. preference for a simpler description. The second approach is to use a relatively small space of hypotheses, and to employ no bias in searching the space of hypotheses for consistent descriptions. An example of this approach is Mitchell's Candidate Elimination Algorithm [Mitchell 77]. His algorithm uses a restricted space of hypotheses to limit those that the program can consider. The program prunes every hypothesis that has been refuted by a training instance.

Whichever approach is adopted, the common fundamental problem is to obtain and use bias so that the learning program guesses the classification of unobserved instances on the basis of appropriate heuristics rather than simple chance. In most research to date, the search for bias is done by hand. A particular bias is encoded, the program is tested, performance noted, new bias formulated, and the cycle repeated until acceptable performance is achieved.

Automation of the search for appropriate bias is a major problem to be solved. Vere's counterfactual technique [Vere 80] uses a biased hypothesis space, and an unbiased search. When no consistent description can be found, the set difference operator is then employed to construct a new description using those that are already in the space. For example, a new description C can be defined as equal to A-B, where A and B are existing descriptions. The new description C is then added to the space of hypotheses that can be searched by the program.

Lenat's EURISKO [Lenat 83] addresses the same problem. The goal of EURISKO is not only to discover interesting concepts, but also to learn heuristics, or bias, that will lead a program to discover interesting concepts. Unlike Lenat's AM, EURISKO alters the bias that drives the discovery procedure.

II. AN APPROACH TO ADJUSTING BIAS

We represent bias as a restricted search space of concepts that we call the *concept description language*. We use "language" in a formal sense; a set of legal sentences in some given formalism.

* This work was supported by National Science Foundation Grant GMCS80-08889, Rutgers University Laboratory for Computer Science Research, and Siemens Research and Technology Laboratories.

** Current address is Siemens Research and Technology Laboratories, 105 College Road East, Princeton, New Jersey 08540.

As shown in figure 1, we use Mitchell's Candidate Elimination Algorithm [Mitchell 77] to maintain the set of all concept descriptions in the concept description language that are consistent with the training instances. That set is called a *version space*. When the trainer supplies a positive instance, all concept descriptions that exclude the instance are refuted and removed from the version space. Similarly, when the trainer supplies a negative instance, all concept descriptions that include the instance are refuted and removed from the version space.

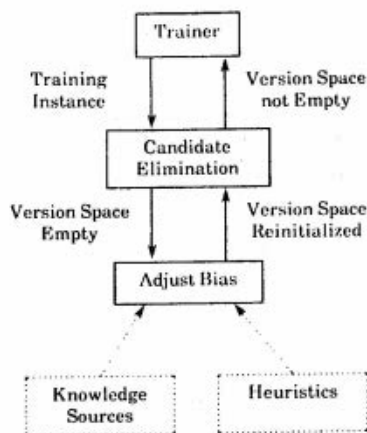


Figure 1 - Learning Element

Figure 2 shows the relationship of the version space, the concept description language, and the formalism for the concept description language. As the version space shrinks, the complementary set of refuted hypotheses grows. If the version space becomes empty, then all descriptions in the concept description language have been refuted. A refuted concept description language indicates a refuted bias, and is sufficient justification for adjusting the bias. We do not yet know stronger justifications. The role of bias adjustment is to enlarge the concept description language so that the version space becomes non-empty and search can continue. It may be that certain concepts are not describable within a particular formalism. If so, shift to a more appropriate formalism may be necessary.

We approach bias adjustment as a three step process:

1. Determine a membership specification for the new concept.
2. Translate the membership specification into a description that uses the formalism of the concept description language.
3. Assimilate the new concept description into the hierarchy of the concept description language.

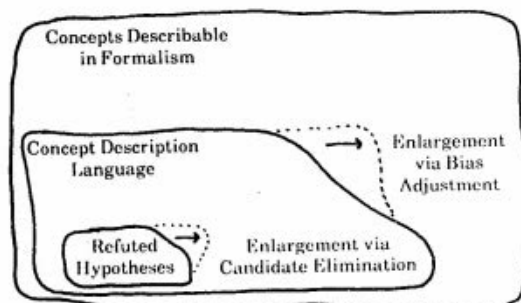


Figure 2 - Model of Bias

A *membership specification* is itself a concept description, but is expressed in an unbiased language so that the heuristics for determining the specification are not compromised by what is expressible. If the membership specification is expressed in an alternate formalism then we require the ability to translate the membership specification into a concept description in the formalism of the concept description language. The translation step may require creation of new descriptive terms that correspond to membership specifications that are not describable in the current concept description language. A new concept must then be assimilated into the concept description language. *Assimilation* is the process of inserting a new concept description into the hierarchy of descriptions in the concept description language. Assimilation is necessary so that we can use the *match* predicate to evaluate whether one concept is more specific or equal to another.

III. AN EXAMPLE

We have implemented a program that models bias adjustment as part of the LEX program [Mitchell 83]. LEX is a concept learning program that learns heuristics that suggest whether a given operator *should* be applied to a given problem state. For each operator there is a heuristic that is represented as the concept "set of states to which this operator should be applied."

LEX currently learns heuristics in the domain of integral calculus. The program is initially given a set of problem-solving operators. Each operator has a domain of applicability, a rewrite rule, and a range of producible states. For each operator, the *legal* domain of applicability describes states to which the operator *can* be applied. For each operator, the *heuristic* domain of applicability describes states to which the operator *should* be applied.

LEX solves a problem by producing a state that contains no integral. The solution tree is then criticized. Each operator application along the minimum solution path is labelled as a positive instance showing a state to which the given operator should be applied. Each operator application that leads away from the minimum

solution path is labelled as a negative instance showing a state to which the given operator should not be applied. The training instances are then used to update the heuristics being learned for the operators. For each operator, Mitchell's Candidate Elimination Algorithm is used to maintain a version space of all candidate versions of the heuristic that have not yet been refuted. If a version space becomes empty, then the bias adjustment module is invoked.

3.1. A Heuristic for Membership

Deciding the membership of a new concept is the essence of induction. Which instances should be included, and which should be excluded? We employ a heuristic:

The domain of an operator sequence that leads to a solution should be describable in the concept description language.

We choose this heuristic because of our a priori knowledge that the operator heuristic we hope to learn will be the disjunction of all useful sequences that start with that operator. To describe the disjunction, we can start by being able to describe the disjuncts. To compute a specification of the domain of an operator sequence that produces a state in a given range, in this case the set of solved problems, we apply a deduction procedure that we call *constraint back-propagation*.

Constraint back-propagation is a procedure for deducing the domain of a macrooperator that produces some constrained range of states. Consider an operator sequence of length 1 that uses operator Op_1 , as shown in Figure 3. If we constrain the range of the Op_1 to some set of states $R \subseteq \text{Range}(Op_1)$, and then apply the inverse operator Op_1^{-1} to R , then we deduce a constrained set $D \subseteq \text{Domain}(Op_1)$ such that $Op_1(D) = R$. For example, an operator $r \rightarrow [r-2]$ has the set of real numbers as its domain and also as its range. If the domain is constrained to the set of integers, then the range is constrained to the set of even integers.

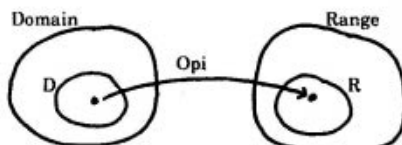


Figure 3 - Constraint Propagation

If we know that Op_1 can produce a state that is a solved problem, we can compute $Op_1^{-1}(\text{Intersect}(\text{SolvedProblems}, \text{Range}(Op_1)))$ to deduce a constrained set D . D is then the set of states to which application of Op_1 will produce a solved problem. For operator sequences of length greater than 1, the deduction step is simply applied sequentially to the result of the previous deduction.

3.2. A Translation Method

A nested expression of intersections and inverse-operator applications, as shown above, is a specification of concept membership, but is not useful as a concept description because operator applications are required to evaluate whether an instance matches the description. The nested expression must be translated from a membership specification into a concept description that uses the formalism of the concept description language. Our translation method is to evaluate the membership specification. To let this work, we have defined a description of the set of solved problems, an intersection function, and inverse operator definitions. New concepts are created and assimilated during evaluation of the membership specification.

When an operator only restructures a state via a simple rewrite rule, e.g. $a \cdot b \rightarrow b \cdot a$, creation of new concepts is not required because the range of the operator is expressible by a description in the concept description language. When an operator requires a computation, e.g. multiplication by 2, the range may be describable in the operator language, but not the concept description language. For example, consider an operator $r \rightarrow [r-2]$. The $[r-2]$ indicates that, as part of applying the operator, the bound value of r should be multiplied by 2, and the result used in the rewritten state. If an operator is to be applied to a set of states, as with constraint back-propagation, then correct specification of the operator's domain and range becomes very important. Application of $r \rightarrow [r-2]$ to the set of integers k produces a description $[k-2]$. This is a description in the operator language, but not the concept description language.

Our method of translating a description implied by a computation $y = f(x)$ to a term in the concept description language is to first create a definition $(y | (\lambda(y) (\text{match } f^{-1}(y) \ x)))$ and then search for a term with the same definition. If a term with that definition is not found, then a new term is created and defined as per the constructed definition. For example, the definition that corresponds to $[k-2]$ is $(y | (\lambda(y) (\text{match } y/2 \ 'k)))$, which we could associate with a new term "even integer."

3.3. An Assimilation Method

Our present method is imprecise. In general, if we are to assimilate $z = (y | (\lambda(y) (\text{match } f^{-1}(y) \ x)))$, then we make z a specialization of y . This method is correct only in the sense that no false match relation is asserted. For example, our method may assert that "even integers" is a specialization of "real numbers", but fail to assert that "even integers" is also a specialization of "integers". At the same time, the method does indicate y as a most general bound on where z should be assimilated. If the set of possible assimilations is small, we would expect that a mechanical procedure could prune these by empirical refutation, but we have not pursued this. Below we show an example of deducing the correct assimilation.

3.4. Experiments

We present three experiments with the implementation. Several problems came to light.

3.4.1. Experiment #1

As in [Utgoff 82], LEX found a solution for $\int \cos^7(x) dx$ similar to that shown in Figure 4. LEX was then given $\int \cos^5(x) dx$, for which the same solution sequence, and $\int \cos^6(x) dx$ for which the same solution sequence did not work because the polynomial in the integrand was not an integer. Because the concept description language did not contain a description that included $\int \cos^7(x) dx$ and $\int \cos^5(x) dx$ and excluded $\int \cos^6(x) dx$, the heuristic for Op51 could not be described. Bias adjustment was invoked, and constraint back-propagation was expected to proceed as shown in Figure 4. The implementation was successful for the last three propagation steps through Op-152, Op-150, and Op-151. Back-propagation through Op-113 failed because our formalism, a modified context-free grammar, does not permit specifying that whatever matches f' be the derivative of whatever matches f . This example shows that the formalism of the concept description language also implies bias. The ability to identify the bias that is associated with a given formalism is an important open problem.

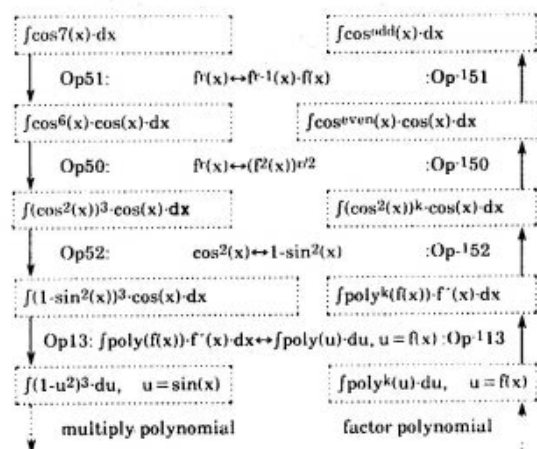


Figure 4 - Solution and propagation for $\int \cos^7(x) dx$

Back-propagation of the set of integers k through the exponent in Op-150 caused a new description $\{x | (\lambda(x) \text{ match } x/2 \text{ 'k'})\}$ to be constructed; the set of even integers. Because the operator could divide any real exponent by 2, the constructed set description was assimilated as a specialization of the set of real numbers. It would have been better to

* The constraint that f' be the derivative of f is encoded as a test within the operator definition, not within the domain description itself.

assimilate the set of even integers as a specialization of the set of integers. Precise assimilation could have been determined with the following reasoning: An alternate equivalent definition of even integers is $\{y | (\exists x \in \text{Integers } y=2x)\}$. Because x and 2 are integers, and because we know that integers are closed under multiplication, we prove that every y is also an integer. The critical piece of knowledge for the proof is that integers are closed under multiplication. We have not implemented a mechanism for executing such proofs. Proposing and then proving subset relationships is an open problem of great interest. The backward constraint propagation method is useful for deducing which conjectures are worth proving.

3.4.2. Experiment #2

In this experiment, the set of real numbers "r" was to be back-propagated through Op22 as shown in Figure 5. The result of the back-propagation step was $r_4(r_3/r_4)$. Because this set was describable by an existing set description, it was not necessary to add any new descriptions to the search space.

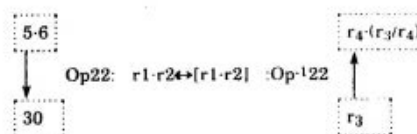
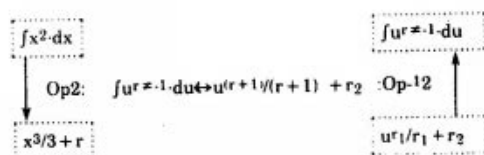


Figure 5 - Evaluating Product with Op22

Nevertheless, the back-propagation failed because the deduced set $r_4(r_3/r_4)$ did not satisfy a safety check requiring that a deduced set, in this case $r_4(r_3/r_4)$, match the state that occurred in the original solution sequence, in this case 5-6. Although r_4 matched 5, r_3/r_4 did not match 6. The concept description language did not include knowledge that r_3/r_4 is equivalent to the set of rational numbers, and that 6 is an instance of a rational number. The inverse operator had inserted an implicit computation; division.

3.4.3. Experiment #3

A problem $\int x^2 dx$ was given to LEX, and was solved by Op2 as shown in Figure 6. Other instances were also provided that caused the bias adjustment module to be invoked. The program needed to back-propagate the set of real numbers "r" through the exponent in Op-12. The code created a description $\{x | (\lambda(x) \text{ match } x+1 \text{ 'r'})\}$, and assimilated it as a specialization of r . This produced a description with infinite recursion of the form "x is a real number if x+1 is a real number". If the program could have applied knowledge that the real numbers are closed under addition, in this case by 1, then it would have proven that adding 1 to an x will not alter whether that x is a real number.

Figure 6 - Solution and propagation for $\int x^2 \cdot dx$

It is important to consider not only where, but whether to assimilate a new description. To circumvent superfluous descriptions, we added an ad hoc procedure for computing closures; if the description D_{new} is intended to be a specialization of an existing description D_{old} , but D_{new} is more general or equal to D_{old} , then D_{new} is superfluous.

IV. SUMMARY

We suggested that heuristics used for induction should be found by the learning program itself, not by hand. We presented a model of a concept learning program that adjusts its own bias, thereby altering the inductive generalizations it will make. We implemented an example of the model by adding a bias adjustment module to the existing LEX concept learning program. Experience with the implementation uncovered examples where the formalism of the concept description language was weak, or additional knowledge was needed to determine whether or where to assimilate a newly constructed description. Where these problems were not a factor, the program was able to automatically adjust its own bias.

ACKNOWLEDGEMENTS

I thank Tom Mitchell, Rich Keller, and Donna Nagel for discussions of ideas presented here. I thank Jack Mostow and Bruce Ladendorf for helpful comments on a draft.

REFERENCES

- Lenat, D. B., "The Role of Heuristics in Learning by Discovery," *Machine Learning*, Michalski, Carbonell, Mitchell (eds.), Tioga, Palo Alto, 1983, pp. 243-306.
- Michalski, R. S., "A Theory and Methodology of Inductive Learning," *Machine Learning*, Michalski, Carbonell, Mitchell (eds.), Tioga, Palo Alto, 1983, pp. 83-134.
- Mitchell, T. M., "Version Spaces: A Candidate Elimination Approach to Rule Learning," *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Mass., 1977, pp. 305-310.
- Mitchell, T. M., Utgoff, P. E., and Banerji, R. B., "Learning by Experimentation: Acquiring and Refining Problem-Solving Heuristics," *Machine Learning*, Michalski, Carbonell, Mitchell (eds.), Palo Alto, Tioga, 1983, pp. 163-190.
- Utgoff, P. E. and Mitchell, T. M., "Acquisition of Appropriate Bias for Inductive Concept Learning," *Proceedings of the Second National Conference on Artificial Intelligence*, Pittsburgh, August 1982, pp. 414-417.
- Vere, S. A., "Multilevel Counterfactuals for Generalizations of Relational Concepts and Productions," *Artificial Intelligence*, Vol. 14, No. 2, September 1980, pp. 138-164.

Operationalizing Advice: A Problem-Solving Model

Jack Mostow

USC Information Sciences Institute
4676 Admiralty Way, Marina del Rey, CA. 90291¹

Abstract

A problem with learning by taking advice arises when the advice is *non-operational*, i.e., expressed in terms of data or actions unavailable to the learner. For example, the advice "avoid taking points" in the card game Hearts is non-operational because the game does not give players an explicit choice about whether to take points. *Operationalization* is the process of converting such advice into an executable (perhaps heuristic) procedure. A model of this process has been implemented in a problem-solver that takes a piece of advice for an agent performing a task and reformulates it in terms of the agent's capabilities by applying transformations from its knowledge base.

The program uses means-ends analysis and planning in an abstraction space to identify non-operational aspects of advice and find transformation sequences that eliminate them. Rather than using a hand-coded difference table, the program analyzes the transformations in order to identify transformation sequences likely to be useful in solving a given problem. Thus new transformations can be added without modifying the problem-solver itself.

Key terms: learning by taking advice, operationalization, program transformation, means-ends analysis, planning in a hierarchy of abstraction spaces

1. Introduction

The idea of learning by taking advice has been around for some time [McCarthy 68], but the ability to accept and follow advice poses several difficult problems. This paper focusses on the problem of handling advice that is *non-operational*, i.e., expressed in terms of data or actions unavailable to the learner.

My dissertation [Mostow 81] [Dietterich et al 82] showed how a transformational approach could capture the process of translating heuristic advice into an effective procedural form. It modelled operationalization as a series of transformations leading from the initial advice statement to an executable procedure. For example, the Hearts advice "avoid taking points in a trick" was transformed into the procedure "play a low card."

¹This research was supported by DARPA Contract MDA-903-81-C-0335. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government. I am grateful to Jeff Barnett, Martin Feather, Matthew Morgenstern, Bill Swartout, and especially Don Cohen for their comments, to the Gist project at ISI for its intellectual influence, and to my dissertation committee (Rick Hayes-Roth, Allen Newell, Jaime Carbonell, and Bob Balzer) for manifold contributions to this research.

This model was incorporated in a program called FOO,² which was used to operationalize a dozen pieces of advice (for Hearts and a music composition task) by applying sequences of transformation rules, some equivalence-preserving, others heuristic. The sequences averaged several dozen steps, but typically each sequence was based on one high-level operationalization strategy; the rest of the steps "jittered" the advice to match the form required to apply the strategy [Fickas 82].

FOO lacked a problem-solver; the user selected transformations by hand and stopped when an operational procedure was produced. This paper describes FOO's successor, a problem-solver called BAR.³ BAR accepts an expression and a pattern; it must rewrite the expression to match the pattern by applying a sequence of transformations from its knowledge base. Automating what was done by hand in FOO raises several issues:

1. *Model of operability:* how can operability be decided automatically?
2. *Goal structure:* given a problem to solve, what subgoals need to be achieved along the way?
3. *Contact:* given a subgoal to achieve, which transformation sequences are likely to help?
4. *Selection:* given several plans for achieving a goal, which is best?

To provide a basis for discussing these issues, Section 2 describes the problem space in which BAR operates; the model of operability is part of this space. Section 3 shows how BAR uses means-ends analysis to identify key subgoals, and Section 4 shows how BAR's abstract plan generator addresses the contact problem. Section 5 presents an extended example to illustrate BAR's strategies for reducing various kinds of differences between an expression and a pattern. Section 6 presents BAR's rather rudimentary techniques for selecting among alternative plans. Section 7 describes some remaining problems. Section 8 evaluates BAR's approaches to the issues, and concludes by putting this work in perspective.

2. The Problem Space

This section describes BAR's problem space [Newell 79] for operationalization, consisting of problem states (initial and transformed advice), goals (patterns), and operators (transformation rules). The problem space represents BAR's knowledge about the task domain, operability, and transformations. BAR's control component is described later.

²For "First Order Operationalizer."

³For "Basically Automatic Reformulator."

2.1. The States

Like FOO, BAR represents advice in a language similar to applicative LISP: there are no assignment statements, the only variables are quantifier variables, and the only functions with side effects are actions that change the state of the environment. A complete definition of this language is outside the scope of this paper: the constructs relevant to this paper include:

- constants
- function applications of the form $(f e_1 \dots e_n)$
- quantification in the form $(Q x S e)$, where Q is a quantifier and e is an expression in which the variable x is quantified over the set S

For example, the initial advice "avoid taking points in a trick" is encoded as (Avoid (Take-points ME) (Trick)), where the constant ME denotes the advice recipient and the other symbols are functions.

2.2. The Goals

BAR considers a procedure to be operational for an agent if it is expressed entirely in terms of the data and actions available to the agent. In this model, the agent interacts with its environment (which may contain other, autonomous agents) via a two-way channel: it can sense observable data, and it can act in certain ways to change the state of the environment. The model assumes that the agent can remember any condition or event it could observe in the past, and derive any computable value of observed or remembered data.

This model of operability leads immediately to two types of goals:

1. Find a procedure to evaluate a given expression using available data.
2. Find a procedure to achieve a given condition using available actions.

In the course of solving such problems, BAR generates subgoals of the form

3. Transform an expression to match the left side of a specified rule.

These goals could be represented formally as the goal of proving (constructively) that

$$\exists e' \exists P(e') \wedge R(e, e')$$

where e is the initial expression, P is a predicate that describes the desired syntactic form of the transformed expression e' , and R specifies the desired semantic relation between e and e' . For example, for type (1) e should be semantically equivalent to e' , whereas for type (2) e' should be a procedure for achieving e . At present, BAR pretends that all its transformations are equivalence-preserving; since they are not, it sometimes produces heuristic solutions. All three types of goals are represented in BAR as transforming an expression to match a pattern expressed in a pattern language.

BAR's pattern language includes the following constructs:

- *Literals*: a literal expression matches itself.

- *Variables*: an expression E matches a pattern variable $?v$, binding $?v$ to E , if $?v$ is unbound; otherwise E matches $?v$ only if E matches $?v$'s binding.

- *Sequences*: a sequence $(f e_1 \dots e_n)$ (normally a function application) matches the pattern $(\text{patt}_0 \dots \text{patt}_n)$ if f matches patt_0 and e_i matches patt_i for $1 \leq i \leq n$.

- *Ellipsis*: the segment pattern $"..."$ matches zero or more consecutive elements of a sequence.

- *Kleene star*: the segment pattern patt^* matches zero or more consecutive sequence elements all of which match patt .

- *Tests*: E matches the pattern $@P$ if $P(E)$ is true. Notice that P is a predicate on syntactic expressions, not on their referents. The tests in the language include *IsAchievable*, *IsComputable*, *IsPredicate*, *IsConstant*, *IsQuantifier*, etc. Tests act like variables, i.e., matching E to $@P$ binds $@P$ to E , so multiple instances of a test in a pattern must be matched by the same expression. Tests can be bound to different expressions by giving them different suffixes. Thus the expression $(= 3\ 4)$ matches the pattern $(= @\text{IsConstant}1\ @\text{IsConstant}2)$ but not the pattern $(= @\text{IsConstant}\ @\text{IsConstant})$. BAR automatically generates such suffixes to distinguish among different instances of a test inside a Kleene star: this allows $(= 3\ 4)$ to match $(@\text{IsComputable}\ @\text{IsConstant}')$.

- *Containment*: an expression matches the pattern $[... \text{patt} ...]$ if it matches patt or contains a subexpression that matches patt . This is a weak version of what it means to be expressed in terms of something; for example, the expression $(f\ x\ y)$ matches $[...y...]$ even if f ignores its second argument. However, BAR's current set of transformations and the way it uses them have so far prevented this deficiency from causing any vacuous solutions.

- *Independence*: an expression matches the pattern $[\sim \dots v \dots]$ if it contains no references to the variable v .

- *Conjunction*: an expression matches the pattern $[\text{patt}_1 \dots \text{patt}_n]$ if it matches both patt_1 and $[\dots \text{patt}_n \dots]$, and similarly for $[\text{patt}\ \sim \dots v \dots]$. More general conjunctive patterns have not been needed yet.

2.3. Model of Operability

BAR's knowledge about operability includes a description of the advice recipient's capabilities in a given environment. For example, BAR knows that the predicate *Has(player, card)* is evaluable when *player* = ME, but not in general, since *player* ME (the learner) can see its own cards but not its opponents'. This is implemented by defining *Has-ME(card)* = *Has(ME, card)* and attaching the property *IsComputable* to the function *Has-ME* but not to *Has*. Similarly, the fact that *player* ME chooses its own card but not its opponents' cards is represented by attaching the property *IsControllable* to *Card-played-by-ME*, defined as *Card-played-by(ME)*, but not to *Card-played-by*.

General knowledge about what is operational is encoded in BAR as a set of facts defining the class of operational procedures. For example, the fact that the application of a computable function to evaluable arguments is evaluable is represented as

(*@IsComputable @IsEvaluable*)
 —matches→ *@IsEvaluable* (1)

Similarly, the fact that an evaluable constraint on a controllable variable is considered operational is represented as

(*@IsConstraint (@IsEvaluable...@IsControllable...)*)
 —matches→ *@IsAchievable* (2)

Rule 2 embodies the heuristic assumption that the value of a controllable variable can be chosen to satisfy a given constraint. For example, it says that the constraint (Require (Low (Card-played-by-ME))) is achievable, but this is false if player ME has no low cards.

2.4. The Operators

BAR's operators are transformations that rewrite expressions in this language. A transformation rule *lhs* → *rhs* can be used to rewrite the advice or to replace a sub-expression of it.

Some of the transformations encode knowledge about domain concepts, which BAR represents as functions. For example, the concept "avoid" is represented by the function

Avoid(event,scenario) =
 (Prohibit (During scenario event)) (3)

That is, "don't let event occur during scenario." Each definition corresponds to two possible transformations: *unfolding* an instance (*f e₁ ... e_n*) of a function *f(x₁ ... x_n)* by substituting *e₁ ... e_n* for *x₁ ... x_n* throughout *f*'s definition, and *folding* an expression that matches this definition into an instance of *f* [Darlington 76].

Other transformations can represent knowledge about a whole class of functions. For example, BAR lacks a definition for the relation *During* but knows that it is reflexive, and has rules for transforming reflexive relations. One such rule is

(*R (f x₁ ... x_n) (f y₁ ... y_n)*) → (*And (= x₁ y₁) ... (= x_n y_n)*) (4)

This transformation rule rewrites a reflexive relation *R* between two expressions (*f x₁ ... x_n*) and (*f y₁ ... y_n*) by equating their corresponding arguments, which is a useful simplification in the absence of a definition for *R*. Rule 4 is shown here in an informal notation for readability; rules are actually implemented as procedures using the Interlisp Pattern Match Compiler [Teitelman 78]. Rule 4's right side logically implies its left side, but the converse need not be true. Thus depending on where it is used, Rule 4 can make heuristic (non-equivalence-preserving) transformations.

3. Overview of Problem-Solver

BAR uses means-ends analysis [Newell 69] to find a sequence of transformations that rewrites a given expression to match a given pattern.

BAR's top-level procedure, called *Transform*, takes an expression and a pattern as input. For example, the problem of operationalizing "avoid taking points" is given to BAR by calling *Transform* with (Avoid (Take-points ME) (Trick)) as the expression and @IsAchievable as the pattern.

Transform calls *Reduce* to reduce differences between the expression and the pattern. *Reduce* identifies the grossest difference between them, looks for plans to eliminate it, selects a plan, and tries to apply it. If this generates subproblems (rewriting the expression to match the left side of a proposed rule), *Reduce* recursively invokes *Transform* to solve them. After each transformation, BAR tries to simplify the new expression. Simplifying the expression typically renders the current plan obsolete and causes BAR to replan. *Transform* calls *Reduce* repeatedly until either the rewritten expression matches the pattern (success) or *Reduce* is unable to make further reductions (failure).

Reduce knows about several kinds of differences, listed below in the order it tries to reduce them:

1. Semantic difference between expression (*f ...*) and pattern (*P ...*): *f* doesn't match *P*
2. Extraction: *f* doesn't match *P*, but (*f ...*) contains a sub-expression that might
3. Recursion: sub-expression doesn't match corresponding subpattern
4. Deletion: sub-expression must be deleted to match pattern [*~...?v...*]
5. Expansion: expression must mention a specified function to match pattern [*...f...*]
6. Restructuring: arguments must be rearranged to match pattern

A plan is simply a series of proposed transformations to apply: it is abstract in that getting the expression into the form required by a step in the plan may require applying additional transformations along the way. The next section shows how BAR finds abstract plans for reducing semantic differences. The treatment of other kinds of differences is illustrated in the Section 5, and BAR's mechanisms for selecting among alternative plans are described in Section 6.

4. Planning with Operator Abstractions

To find plans for reducing semantic differences, BAR solves an abstracted version of the problem. The purpose of the abstract search is to find a high-level plan for rewriting an expression (*f ...*) to match a pattern (*P ...*), that is, to find a sequence of rules that reformulates an instance of *f* as an instance of some *g* that matches *P*. This is accomplished by abstracting the original problem space essentially as follows:

- Abstract a state (*f ...*) as *f*
- Abstract a goal (*P ...*) as *P*
- Abstract a rule (*L ...*) → (*R ...*) as *L* → *R*

BAR then searches for a sequence of abstracted rules leading from *f* to *P*; this sequence constitutes an abstract plan for rewriting (*f ...*) to match (*P ...*), i.e., carrying out the plan generally requires additional transformations to fully match the left side of each step.

Somewhat more precisely, BAR searches for a path from f to P (without repeated symbols) consisting of the following kinds of steps:

- $f \rightarrow \text{unfold} \rightarrow g$, where f 's definition has the form $(g \dots)$
- $f \rightarrow \text{fold} \rightarrow g$, where g 's definition has the form $(f \dots)$
- $L \rightarrow \text{rule} \rightarrow R$, where rule has the form $(L \dots) \rightarrow (R \dots)$
- $f \rightarrow \text{matches} \rightarrow @P$, where f satisfies the test P^4
- $f \rightarrow \text{fact} \rightarrow @P$, where fact has the form $(f \dots) \rightarrow \text{matches} \rightarrow @P$

To find such plans, BAR uses straightforward depth-first search up to a preset depth limit, currently 10. The recursive search procedure $\text{FindRephrasePlan}(\text{expression}, \text{pattern}, \text{depth})$ can be described as follows:

```

if expression matches pattern
then return the one-step plan
    expression  $\rightarrow$  matches  $\rightarrow$  pattern
else if depth = 0 then fail
else if there is an allowable step
    expression  $\rightarrow$  next such that
        FindRephrasePlan(next, pattern, depth-1) returns
        a plan
    then return CONS(step, plan)
else fail.

```

To prune the search, FindRephrasePlan uses memo functions [Marsh 70] in several ways. First, it remembers successful calls; if it has earlier found a path from g to P , and encounters g while searching for a path from f to P , it will use it to form a path to P . Second, it remembers unsuccessful calls and their associated depth limit. If $\text{FindRephrasePlan}(g, P, n)$ fails, subsequent calls of the form $\text{FindRephrasePlan}(g, P, k)$ with $k \leq n$ will fail immediately. BAR further prunes the search by testing whether a proposed step is allowable; it rejects any step that repeats an earlier state in the current path, or has proved impossible to carry out in the past.

BAR does not stop after finding one plan to reduce a given difference; it tries to cover the set of possible plans by finding all first steps that can be extended into a plan. BAR does not bother to generate more than one plan with the same first step, since there is no point in finding all such plans if the first step fails. If BAR gets stuck partway through a plan, it tries replanning from the current expression. If this fails, BAR backs up to the original expression and tries a different first step. This arrangement reduces the search for plans at the cost of replanning whenever the chosen plan strays from the correct path after the first step. To generate only one path with a given first step, FindRephrasePlan rejects the first step if a plan that starts with it has already been found for reducing the current difference.

In the case at hand, FindRephrasePlan finds one plan for eliminating the semantic difference between (Avoid (Take-points ME) (Trick)) and the pattern $@IsAchievable$:

```

Avoid  $\rightarrow$  unfold  $\rightarrow$ 
Prohibit  $\rightarrow$  matches  $\rightarrow$ 
@IsConstraint  $\rightarrow$  Fact 2  $\rightarrow$ 
@IsAchievable

```

(5)

⁴To make this abstraction work, BAR's tests are usually defined so that if the expression $(f \dots)$ satisfies a test, e.g., $IsPredicate$, so does the function f .

Actually, this description is over-simplified -- a plan can contain more information than a series of atomic symbols connected by rules. FindRephrasePlan tries to preserve as much information as possible about the expression resulting from each step. If a rule in a plan can be applied without subgoaling, FindRephrasePlan does so. If not, it tries to force a match between the expression and the left side of the rule, and then substitute the resulting bindings in the right side of the rule to produce a partial instantiation. These bindings represent subproblems to be solved. Only as a last resort does BAR abstract an expression as its topmost symbol.

Although BAR applies its transformation rules procedurally, a procedural rule representation makes it difficult to identify the subgoals required to match the left side of a rule in a plan, or to predict what the eventual result of applying it will look like. For this kind of reasoning, BAR uses rule knowledge expressed in the pattern language. For example, Rule 4 is described as

$(@IsReflexive (?f \dots) (?f \dots)) \rightarrow (And (= ?x ?y))$

This is an abstraction: it lacks information contained in the procedural representation, but captures several facts useful in deciding when to try Rule 4:

- Rule 4 rewrites a reflexive predicate as a conjunction.
- The two arguments of the predicate must be instances of the same function.
- Rule 4 eliminates the reflexive predicate and the function.

Thus plan 5 actually looks like this:

```

(Avoid (Take-points ME) (Trick))
 $\rightarrow$  unfold  $\rightarrow$ 
(Prohibit (During (Trick) (Take-points ME)))
 $\rightarrow$  matches  $\rightarrow$ 
(@IsConstraint [@IsEvaluable...@IsControllable...])
 $\rightarrow$  Fact 2  $\rightarrow$ 
@IsAchievable

```

(6)

Executing this plan will require solving the subproblem of rewriting (During (Trick) (Take-points ME)) to match $[@IsEvaluable...@IsControllable...]$. This subproblem takes several dozen steps to solve, and illustrates the benefit of planning in the abstracted problem space [Sacerdoti 74]: it prunes the search by trying only those transformations that lead to a solution in the abstraction space. Otherwise BAR could waste lots of time solving such subproblems along the way to a dead end.

5. Reduction Strategies Illustrated

BAR's strategies for reducing each kind of difference are now illustrated by means of an extended example: the operationalization of "avoid taking points." This takes about 60 transformation steps, but the reasoning can be loosely paraphrased as follows:

1. I want to avoid taking points in the trick.
2. To take points, I would have to take a trick with points.
3. To take the trick, I would have to play the highest card in the suit led.

4. So I should play a card lower than somebody else's.
5. But I can't always tell whether my card will be lower than somebody else's.
6. So I should simply play a low card, and hope for the best.

The problem is given to BAR by calling Transform with (Avoid (Take-points ME) (Trick)) as the expression and @IsAchievable as the pattern to match. Reduce identifies the semantic difference between Avoid and @IsAchievable as the grossest difference between the two, and FindRephrasePlan finds plan 6 above. BAR looks for all possible plans before choosing which one to apply, but in this case it can only find one, so it proceeds to execute it.

The first step of the plan unfolds the definition (3) of Avoid, rewriting the initial advice as

(Prohibit (During (Trick) (Take-points ME))) (7)

Although Prohibit satisfies the test IsConstraint, this expression does not match the left side of Fact 2, because

(During (Trick) (Take-points ME)) (8)

is not an evaluable constraint on a variable controlled by ME. When the current expression does not match the next step in a plan, Reduce invokes Transform to rewrite the expression to match the left side of the rule. This idea is not new -- GPS invoked itself to satisfy the preconditions of an operator it wanted to apply [Newell 60]. Thus Reduce now calls Transform to rewrite expression 7 to match (@IsConstraint : [:@IsEvaluable ...@IsControllable...]), and Transform calls Reduce to reduce the differences between them.

Given an expression ($f(e_1 \dots e_n)$) and a pattern ($P \text{ patt}_1 \dots \text{patt}_n$) where f matches P , Reduce uses recursion to reduce the differences between them by Transforming each e_i to match the corresponding subpattern patt_i . Thus Reduce now calls Transform with expression 8 and the pattern [:@IsEvaluable...@IsControllable...].

By unfolding the definitions of Trick and Take-points and applying Rule 4, BAR eventually reduces expression 8 to

(And (Trick-has-points)
(= (Card-played-by ME) (Winning-card))) (9)

That is, for player ME to take points, the trick must have points and player ME must play the highest card in the suit led.

At this point, BAR is partway through a plan to rewrite the expression as an instance of the computable function And, and then apply Fact 1: to match the left side of the rule, each conjunct must match @IsEvaluable. To simplify this example, Trick-has-points is marked IsComputable to represent the existence of (heuristic) methods for predicting whether a trick will have points. (BAR has not been used to derive such methods but this could be done.) However,

(= (Card-played-by ME) (Winning-card)) (10)

is unevaluable because Winning-card cannot be predicted. To make this expression evaluable, BAR decides to apply

(= (The x S Px) c) $\rightarrow$ (And (In c S) Pc) (11)

It turns out that (Card-played-by ME) cannot be transformed to match (The x S Px) given BAR's current set of concept definitions.

When Reduce cannot transform the arguments of an expression to match the arguments of a pattern, it looks for rules that might restructure the expression by moving information between arguments. It then tries to transform the restructured expression to match the pattern. A rule is considered a restructuring rule if it has the form $P \dots \rightarrow P \dots$, i.e., leaves the function of an expression unchanged. One such rule transposes the arguments of a symmetric function:

(@IsSymmetric ?x ?y) $\rightarrow$ (@IsSymmetric ?y ?x) (12)

Applying Rule 12 to expression 10 produces

(= (Winning-card) (Card-played-by ME)) (13)

To apply Rule 11, BAR unfolds (Winning-card), producing

(= (The ?c (Cards-played-in-suit-led)
(Forall ?x (Cards-played-in-suit-led)
(Not (Higher ?x ?c))))
(Card-played-by ME)) (14)

The result of applying Rule 11 reduces to

(And (In-suit-led (Card-played-by-ME))
(Forall ?x (Cards-played-in-suit-led)
(Not (Higher ?x (Card-played-by-ME))))) (15)

That is, in order for ME to take the trick, no other player can play a higher card in the suit led.

BAR considers the first conjunct of expression 15 to be evaluable, but the second conjunct is quantified over (Cards-played-in-suit-led), which player ME cannot generally predict. When an expression cannot be transformed to match a pattern but contains an argument that can, BAR looks for rules that might extract the argument. A rule is considered if its right side is embedded in its left side, like

(@IsQuantifierPredicate ?x ?S [:@IsPredicate ...?x...])
 $\rightarrow$ @IsPredicate (16)

Rule 16 extracts a predicate from the scope of a quantifier provided the predicate is independent of the quantifier variable. It suggests a plan for evaluating expression 15: eliminate the quantifier variable ?x from the predicate

(Not (Higher ?x (Card-played-by-ME))) (17)

Then extract the predicate and make it evaluable.

To delete a variable from an expression, BAR looks for rules that have a pattern variable in an argument position on the left side but not on the right side. One such rule is abstracted as

(@IsComparative ?e ?v) $\rightarrow$ (@IsPredicate ?e) (18)

Rule 18 approximates a binary ordering relation as the corresponding unary predicate, thereby deleting ?v. BAR decides to use Rule 18 to delete ?x from

(Higher ?x (Card-played-by-ME)) (19)

However, ?x must be the second argument of the binary relation for this plan to work, so BAR first rewrites expression 19 as (Lower (Card-played-by-ME) ?x) and then applies Rule 18, reducing expression 19 to (Low (Card-played-by-ME)). That is, to make my card lower than somebody else's, I should play a low card and hope for the best.

The final result of Transforming expression 8 to match @IsEvaluable is the expression

(And (Trick-has-points)
(In-suit-led (Card-played-by-ME))
(Not (Low (Card-played-by-ME)))) (20)

Since Card-played-by-ME is a choice variable controlled by player ME, expression 20 also happens to match the pattern [...@IsControllable...] in Fact 2. Otherwise, Reduce would try to expand expression 20 in terms of some controllable function. To transform an expression to match [...P...], BAR uses a search that resembles FindRephrasePlan but allows steps that go from an expression to one of its subexpressions.

At this point the overall expression is

```
(Prohibit (And (Trick-has-points)
  (In-suit-led (Card-played-by-ME))
  (Not (Low (Card-played-by-ME))))) (21)
```

This expression matches the left side of Fact 2.

```
(@IsConstraint (@IsEvaluable...@IsControllable...))
  —matches→ @IsAchievable
```

@IsAchievable was the original goal, so BAR halts. Thus the result of operationalizing "avoid taking points" can be freely paraphrased as "play a low card if the trick has points."

6. Selection

Since BAR deals with problems requiring dozens of steps to solve, and often finds several plans for reducing a given difference, it faces a combinatorial search problem that will worsen as more rules and concepts are added. It uses several techniques to deal with this problem.

BAR rates the plans it finds according to their estimated chance of success, and tries them in order until it finds one that succeeds. The rating depends on the number of steps in the plan that require additional problem-solving and a numerical strength for each rule. Heuristic rules like Rule 18 are penalized so as to try correctness-preserving plans first.

Alternatively, BAR provides a mode called UserSelectPlan that allows the user to guide the search. In this mode, whenever Reduce finds multiple plans it asks the user to select among them, but whenever it finds only one plan it tries it without asking. This reduces the question-answering burden on the user but leads to extra search in cases where a single plan is found and fails.

BAR caches the results of all calls to Transform and Reduce in the course of solving a given problem [Lenat 79] [Marsh 70]. This allows a subproblem solution found in one branch of the search to be used in another, and prevents a failed subproblem from being re-attempted. Caching is also used to avoid the kind of looping that occurs when a problem generates itself as its own subproblem. When a problem is first attempted, a failure result is cached; if in the course of solving the problem, a subproblem identical to it is encountered, the subproblem will fail without preventing the solution of the original problem by other means.

Finally, BAR keeps track of the number of successful and unsuccessful attempts to apply each rule. This information is used both to order the path-finding search and to rate plans once they are found, allowing BAR to tune its feasibility estimates even during the course of a single problem. This simple form of learning generalizes experience gained in the course of problem-solving, and allows BAR to tune its own performance within and between problems. It is compatible with user guidance: in UserSelectPlan mode, the plan ratings have no effect, but tuning increases the chances that the first plan found with a given initial

step will be the most feasible plan starting with that step. This cuts down on the replanning required when the adopted plan gets stuck after the first step. Of course such statistical learning is quite weak; learning to discriminate between successful and unsuccessful attempts to apply a rule would be much stronger. However, predicting whether a rule can be applied to a given expression requires rather deep analysis, and it may be less work to try it and see.

As of this writing the "avoid taking points" example has been operationalized only in UserSelectPlan mode. Even though the right plan is frequently the only one found or is scored higher than the alternatives, and wrong paths often reach dead ends quickly, BAR's rating scheme is not strong enough to prevent an explosive search, and is fragile in the face of added rules.

7. Remaining Problems

BAR can be evaluated by how well it decides operability, models the goal structure of operationalization, makes contact between problems and methods, and selects among competing methods.

As implemented, BAR's model of operability ignores preconditions and state dependencies -- it does not handle procedures that can be executed only at certain times or in certain situations. For example, the procedure to avoid taking points implicitly assumes that player ME can legally play a low card. Also, the circumstances under which Trick-has-points is computable are not explicitly represented. The model assumes that the environment is external to the agent operating in it: it does not address the problem of representing and reasoning about states of knowledge. Although BAR uses heuristic methods like Rule 18 that lead to fallible procedures, it does not consider their relative reliability. To analyze the logical status of a transformation, BAR would have to keep track of the type of goal being pursued and the context of the sub-expression to be transformed. Also, the model assumes that all computation is free.

A more realistic model would acknowledge that operability is relative: some methods are more reliable than others, some computations are more expensive than others, and the operability of a procedure depends on the feasibility of executing it and the probability that doing so will produce the desired result. These improvements are needed to handle operationalization goals of the form "make this procedure cheaper to execute," "make this estimate more accurate," or "make this plan succeed more often." Some such problems were solved in FOO, for example the problem of refining a heuristic search procedure [Mostow 82], but the goals were implicit in the hand-selected sequence of transformations.

BAR uses a rather syntactic goal structure: given an expression and a pattern, its top-level goal is to change the top-most symbol of the expression to match the top-most symbol of the pattern. It addresses the contact problem by searching for abstract plans to get from one to the other. This planning narrows the set of attempted transformations to those for which some abstract path to the solution exists. However, this scheme generates vacuous plans when the top-most symbol contains little semantic content, e.g., "And" or "Exists." Moreover, this goal structure appears unintuitive when the important reasoning is buried in one of the subproblems. The planner might be improved by explicitly identifying the obstacles to the evaluation of an expression. For example, the fact that BAR lacks a definition for During would motivate its deletion from expression 8.

To constrain the search, BAR assumes that an expression (f ...) can be transformed to match a pattern (g ...) without rewriting the expression in the form (g ...) along the way, if it can be done at all. It is easy to construct examples that violate this assumption. For example, to transform (Union A B) into (Union B A) if one didn't know Union was commutative, one might have to unfold (Union A B) as (SetOf x Universe (Or (In x A) (In x B))), transpose the arguments of Or, and refold into (Union B A). BAR cannot find proofs of this simple "bridge" form so familiar to algebra students, although none have been needed in the few examples attempted thus far. Allowing BAR to consider such plans would explode its search space unless the conditions for considering them were strongly restricted.

BAR's selection mechanism is rather weak, since it relies on crude numerical estimates of rule feasibility. Tuning these estimates based on experience provides limited improvement at best. Also, BAR searches depth-first, although it tries plans in order of estimated feasibility. A breadth-first strategy could find short transformation sequences faster, but would require more memory. Also, implementing it would be complicated by the recursive nature of the problem-solver. A more promising approach is to constrain the output of the plan generator based on the identified non-operational aspects of the current expression.

8. Conclusion

BAR is still in a relatively early stage of development; so far it contains only 60 of FOO's 230 rules and 36 of FOO's 112 concept definitions, and FOO itself came nowhere close to covering the space of operationalization problems. In the absence of additional knowledge or improved control strategies, BAR's syntactic approach can be expected to doom it to combinatorial explosion in the face of added rules and concepts. Thus BAR should not be viewed as a practical tool for operationalization, but as a vehicle for investigating the knowledge, inference methods, and goal structure needed by such a tool. Already it has inspired the several improvements proposed in the previous section.

BAR addresses only one of many problems involved in taking advice. Other key problems include interpreting informally expressed advice, acquiring domain concepts, developing a catalog of transformation rules, and integrating multiple pieces of advice [Mostow 79b]. Practical application of the work described here will be easiest where these other problems are minimized, i.e., where the advice, domain knowledge, and a small but adequate set of transformations have already been formalized.

A promising candidate for such an application arises in the course of transforming a formal closed system specification [Feather 83] into a compilable program. Such a specification defines acceptable behavior for a system of interacting components, and describes the data and actions available to each component; the problem is to implement each component subject to these restrictions. Many of the required transformations have already been formalized [London & Feather 82]. An especially attractive aspect of this application is that a useful solution need not solve the control problem completely: automating even some of the implementation decisions could significantly amplify human software productivity [Fickas 82].

References

- [Darlington 76] J. Darlington and R. M. Burstall, "A system which automatically improves programs," *Acta Informatica* 6, 1976, 41-60.
- [Dietterich et al 82] T. G. Dietterich, Bob London, K. Clarkson, and G. Dromey, "Mostow's Operationalizer," in P. R. Cohen and E. A. Feigenbaum, Volume 3 (eds.), *Handbook of Artificial Intelligence*, Stanford Computer Science Department, Stanford, CA, 1982. In section on Learning and Inductive Inference, available as STAN-CS-82-913/HPP-82-10.
- [Feather 83] M. S. Feather, *Closed System Specifications*, 1983. In preparation.
- [Fickas 82] S. Fickas, *Automating the Transformational Development of Software*, Ph.D. thesis, University of California at Irvine, 1982.
- [Lenat 79] D. B. Lenat, F. Hayes-Roth, and P. Klahr, "Cognitive economy in artificial intelligence systems," in *IJCAI-6*, pp. 531-536, Tokyo, Japan, 1979.
- [London & Feather 82] London, P.E. & Feather, M.S., "Implementing specification freedoms," *Science of Computer Programming*, (2), 1982, 91-131.
- [Marsh 70] D. Marsh, "Memo functions, the Graph Traverser, and a simple control situation," in B. Meltzer and D. Michie (eds.), *Machine Intelligence* 5, pp. 281-300, American Elsevier, New York, 1970.
- [McCarthy 68] J. McCarthy, "The advice taker," in M. Minsky (ed.), *Semantic Information Processing*, pp. 403-410, MIT Press, Cambridge, MA, 1968.
- [Mostow 79a] D. J. Mostow and F. Hayes-Roth, "Operationalizing heuristics: some AI methods for assisting AI programming," in *IJCAI-5*, pp. 601-609, Tokyo, Japan, 1979.
- [Mostow 79b] D. J. Mostow and F. Hayes-Roth, *Machine-aided heuristic programming: A paradigm for knowledge engineering*, The Rand Corporation, Santa Monica, CA, Technical Report Rand-N-1007-NSF, 1979.
- [Mostow 81] D. J. Mostow, *Mechanical Transformation of Task Heuristics into Operational Procedures*, Ph.D. thesis, Carnegie-Mellon University, 1981. Technical Report CMU-CS-81-113.
- [Mostow 82] D. J. Mostow, "Learning by being told: Machine transformation of advice into a heuristic search procedure," in J. G. Carbonell, R. S. Michalski, and T. M. Mitchell (eds.), *Machine Learning*, Palo Alto, CA: Tioga Publishing Company, 1982.
- [Newell 60] A. Newell, J. Shaw, H. Simon, "Report on a general problem-solving program for a computer," in *Proceedings of the International Conference on Information Processing*, pp. 256-264, UNESCO, Paris, 1960.
- [Newell 69] A. Newell, "Heuristic programming: Ill-structured problems," in J. Aronofsky (ed.), *Progress in Operations Research*, pp. 363-414, Wiley, New York, 1969.
- [Newell 79] A. Newell, *Reasoning, problem solving and decision processes: the problem space as a fundamental category*, Carnegie-Mellon University Computer Science Department, Pittsburgh, PA, Technical Report, June 1979.
- [Sacerdoti 74] E. D. Sacerdoti, "Planning in a hierarchy of abstraction spaces," *Artificial Intelligence* 5, 1974, 115-135.
- [Teitelman 78] Teitelman, W., *Interlisp Reference Manual*, Xerox Palo Alto Research Center, 1978.

GOAL DIRECTED LEARNING*

Tom M. Mitchell and Richard M. Keller

Computer Science Department
Rutgers University
New Brunswick, NJ 08903

1 Introduction

Much research on learning has focussed on methods for empirical generalization: for learning descriptions of general concepts from training examples. This generalization problem is usually formulated so that the only information available to the learner is the set of input training instances, and some a priori biases for preferring certain generalizations over others. We argue here that this kind of formulation of the generalization problem is unrealistic and overly difficult—that when humans face such generalization problems, they typically have some goal, or purpose for attempting to learn, a particular concept. Our main point is that explicit knowledge of this learning goal, and of the context in which it was formulated, can provide very strong guidance for learning.

The longer version of this paper will compare two recent learning systems that use explicit knowledge of their learning goal as the basis for driving learning. One of these is the LEX2 system [Mitchell 82, Keller 83, Mitchell 83], which learns problem-solving heuristics in integral calculus. The learning goal of LEX2 is to learn "heuristics that recommend problem solving steps which lead to solutions". LEX2 interprets each of its positive training instances in terms of this learning goal, in order to determine which features of the instance are essential, and which are irrelevant to the heuristic to be learned. In LEX2, we have found that introducing explicit knowledge of the learning goal leads to two important new capabilities: (1) the ability to form a justifiable generalization by analyzing a single training instance [Mitchell 82, Keller 83, Mitchell 83], and (2) the ability to discover new terms, such as "odd integer", which are essential for describing heuristics in the program's task domain of integral calculus [Utgoff 82].

In comparison to LEX2, its predecessor, LEX [Mitchell 81], had no explicit knowledge of its learning goal. In giving LEX2 knowledge of its learning goal, we have changed in a fundamental way the nature of the generalization problem. In particular, we have told the system the concept (e.g., "problem solving steps that lead to solutions") that it is to learn! Without explicit knowledge of its learning goal, LEX was forced to make inductive leaps to guess the classification of unobserved training instances. With knowledge of this learning goal, LEX2 begins by knowing *in principle* the classification of every possible training instance. But while the learning goal

constitutes a *valid* definition of the concept to be learned, it is not a very *useful* definition. In particular, to determine whether any particular problem solving step is a "problem solving step which leads to a solution", one must first attempt to find a solution! Since learned heuristics are meant to guide the problem solver in finding solutions, this method for evaluating whether a potential search step should be attempted is prohibitively expensive. Thus, the learning problem in LEX2 can be viewed as a problem of reformulating a valid definition of the concept to be learned, into a form in which it is more useful for recognizing instances of the concept.

A second system which uses explicit knowledge of its learning goal is the BAR system [Mostow 83] (a successor to the FOO system [Mostow 82]) which learns effective strategies for playing the card game Hearts. BAR starts with knowledge of its learning goal (for example to learn "which cards to play to avoid taking points"). As with LEX2, this initial piece of advice is not particularly useful, since it gives no information as to how the recommendation should be implemented. Similarly, a piece of advice which requires complete knowledge of another player's hand (e.g. "lead with a card having higher point value than any card of the same suit in another player's hand") is not directly useful since this kind of information is not typically available to the card player. The system must reexpress such advice into a form which the player can use more directly. For example, there are many possible ways in which the advice to "avoid taking points" could be successfully realized by a Hearts-player. Consider the following operational forms:

- Always play your lowest card in the suit led.
- Underplay another card in the current hand.
- Use probabilistic methods to determine the best card to play.
- Search through a space of possible scenarios in which one might envision taking points. Avoid any such scenarios.

Mostow's program can derive each of these forms from the initial, non-usable advice about avoiding tricks with points. In contrast to LEX2, BAR reasons directly from its learning goal to the more operational form of its knowledge, without considering training examples to guide this process.

*This is an extended abstract of a forthcoming paper to be presented at the Second International Machine Learning Workshop. This work is supported by the National Science Foundation under grant MCS80-08889, and by the National Institutes of Health under grant RR-64309.

Goal-directed learning is a form of learning by being told the learning goal defines the concept to be learned, and the learning problem is one of reexpressing this concept in a more useful form. While the learner's goal could come from outside the learning system, we argue that for many kinds of learning, the learning goal could instead be formulated by the system itself. Given the capability to monitor its own performance, and to formulate learning tasks, a problem-solving system should be able to compose learning goals such as "learn heuristics for task X, so that the heuristics suggest problem-solving steps leading to solutions." In such cases, goal-directed learning corresponds to learning by being told *by oneself*. The Eurisko program [Lenat 82] provides one example of a system that monitors aspects of its own performance in order to formulate certain types of useful learning tasks. We suggest that such learning tasks are naturally suited to goal-directed learning.

References

- [Keller 83] Keller, R.
Learning by Re-Expressing Concepts for Efficient Recognition.
In *Third National Conference on Artificial Intelligence*, to appear. Washington, D.C., August, 1983.
- [Lenat 82] Lenat, D. B.
The role of heuristics in learning by discovery: three case studies.
In Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (editors), *Machine Learning*. Tioga Press, Palo Alto, C.A., 1982.
- [Mitchell 81] Mitchell, T. M., Utgoff, P. E., Nudel, B. and Banerji, R.
Learning problem-solving heuristics through practice.
In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 127-134. Vancouver, August, 1981.
- [Mitchell 82] Mitchell, T. M.
Toward Combining Empirical and Analytic Methods for Learning Heuristics.
In Elithorn, A. and Banerji, R. (editors), *Human and Artificial Intelligence*, Erlbaum, 1982.
- [Mitchell 83] Mitchell, T.
Learning and Problem Solving.
In *Proceedings of IJCAI-8*, to appear. Karlsruhe, Germany, August, 1983.
- [Mostow 82] Mostow, D. J.
Transforming declarative advice into effective procedures: a heuristic search example.
In Michalski, R. S., Carbonell, J. G. and Mitchell, T. M. (editors), *Machine Learning*. Tioga Press, Palo Alto, C.A., 1982.
- [Mostow 83] Mostow, D.J.
Operationalizing Advice: A Problem-Solving Model.
In *Proceedings of the Second International Workshop on Machine Learning*, pages . Urbana, Illinois, June, 1983.
- [Utgoff 82] Utgoff, P. E. and Mitchell, T. M.
Acquisition of Appropriate Bias for Inductive Concept Learning.
In *Proceedings of the 1982 National Conference on Artificial Intelligence*. Pittsburgh August, 1982.

PROGRAM SYNTHESIS AS A THEORY FORMATION
TASK: PROBLEM REPRESENTATIONS AND
SOLUTION METHODS*

Saul Amarel**
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

EXTENDED ABSTRACT

The long-term objective of this research is to understand theory formation processes and to develop effective methods for mechanizing them. It has been clear for some time that we need to make progress in this area of AI in order to develop applications that will have serious impact on the "doing of science." Recent work shows that advances in theory formation are of a more fundamental significance for AI, especially in developmental issues related to problem solving. Theory formation capabilities are necessary for problem solving systems that can improve their performance by learning from experience; in particular, systems that can shift the representation of the problem on which they are working via the effective use of new knowledge discovered during their problem solving activity [See Amarel, 1982].

The specific focus of our work has been on the interactions between problem representations and methods for searching/building solutions in theory formation problems, and also on processes for moving between alternative problem representations. We are exploring these issues in the context of a specific program formation (synthesis) task in the mathematical domain of finite partially ordered structures. This is an extremely rich domain with many applications to computing. In particular, we are considering the following problem: given a language of programs, and the specification of a function (such as Infimum) in the form of a table of associations between data elements in the domain; build a program that defines the function.

The problem that we face here is how to search in the given language of programs for a program that satisfies the constraints imposed by the given set of associations. This can be regarded as a constraint satisfaction problem, where the constraints are presented in a data space, and the possible solutions are presented in a second space -- the program space; and furthermore, the initial body of available knowledge is not in a form that permits useful links to be readily established between the two spaces. An effective solution method requires the development of strong links between the two spaces, and the

establishment of a strategy for good coordination between the search/reasoning activities in these spaces. Problems of this type have been studied by Simon and Lea [1974].

In previous work, we developed two approaches for solving our program formation problem. The first approach [See Amarel, 1982] started by specifying a grammar for the language of programs, which provided rich facilities for a hierarchical description of the structure of programs. The choice of the grammar was essentially arbitrary. We then assumed that certain types of regularities exist between structured descriptions of programs obtained in the grammar and the functional behavior of these programs; and we formulated an evaluation function which ranked candidate programs by their degree of partial success in satisfying the given functional constraint of the problem, i.e. the table of data associations. Using the grammar, the evaluation function and the assumptions of regularity, we formulated a "Generate-and-Test" procedure, which was guided by a heuristic hill-climbing search strategy in program space. No reasoning in data space took place here. The problem data was used mainly for testing a candidate program/hypothesis, and only very weakly for a priori control of program generation. Even though a moderate level of performance was obtained by this approach, it became clear very soon that performance was highly sensitive to the choice of program grammar. Small changes in the way of specifying the grammar had strong effects on overall system performance. This experience convinced us of the centrality of the problem of representation in the formation process; and it led us to the development of a second approach for handling the problem.

The second approach [See Amarel, 1971] is based on a model of program space which enables us to reason about programs in terms of their functional characteristics. We constructed such a model in the form of a modified algebra of relations whose terms correspond to structural descriptions of programs. Given the model, it was possible to formulate a strategy of search in problem space which is more goal-directed than in our first approach. The search involves obtaining estimates of "distance" in program space between candidate programs and the goal program, and applying "distance reducing" moves in program space that are chosen on the basis of algebraic properties of the model. In this approach, a considerable amount of specific knowledge about program space had to be used -- to constrain the choice of program

*This research was supported in part by the Division of Research Resources, NIH, Grant RR00643 to the Rutgers Research Resource on Computer in Biomedicine.

**Visiting Research Fellow at the AI Center, SRI International, Menlo Park, California, during the Spring of 1983.

grammar, and to provide a more solid basis for the assumptions of structure-function regularity that are needed in the problem solving process. This approach resulted in a more robust (and an improved level of) performance in solving our problem by improving the representation of program space relative to the given task. Again, no direct reasoning in data space was involved.

In recent work we are concentrating on a third approach to program formation, where the process is strongly guided by the detailed analysis of individual data associations. In contrast to the previous two approaches where the main thrust of reasoning is from candidate programs to data associations, the present approach is mainly data driven; but it is strongly guided by the program grammar and by the algebraic properties of the model of program space. Given a data association, a derivation problem is solved to obtain a "computational path" between the two elements of the association. The program grammar provides the reduction schemes and the moves for solving the derivation problem. The key problem that we face here is how to combine the "computational paths" of several data associations in order to infer a single program that can be seen as generating them; and how to merge programs that provide partial solutions to our program (by satisfying subdomains of the function) into a program that represents a global solution for the entire domain. Algebraic properties of program space can help in this merging process.

By shifting the reasoning towards the data space, several interesting domain-dependent problems of concept specialization and generalization have been encountered. In general, the present approach appears more flexible and more general than our previous two approaches.

Our recent work in this area has resulted in the identification of several basic issues in AI and in theory formation. Clearly, the degree of *dependence* between the conditions (or the goals) of a problem is of fundamental importance for the choice of a problem solving method. More research is needed on methods for handling conjunctive goal situations, not only for planning or design problems but also for theory formation problems. Also, it became evident that the notion of problem formulation in theory formation is somehow fluid and open-ended. While our present formulation of the theory formation task is oriented to the finding of a definition for a *given* concept in terms of a given language, this represents a special case of theory formation. In our case, the concept is well-specified from outside the system in terms of a concrete domain which we are asked to express in the language of the theory. More generally, theory formation involves both the choice of a (sub)domain to express within a language of theory, and the finding of an expression for defining/explaining the (sub)domain in the theoretical terms of the language. The interplay between domain choice and the formation of a theory for the domain is an interesting and difficult problem, which we have started to explore.

REFERENCES

- Amarel, S., "On the Automatic Formation of A Computer Program which Represents a Theory," in *Self Organizing Systems - 1962*, Yovits, Jacobi, Goldstein. (Editors), Spartan Books, 1962.
- Amarel, S., "Representations and Modeling in Problems of Program Formation," in *Machine Intelligence 6*, Meltzer and Michie (Editors), University of Edinburgh Press, 1971.
- Amarel, S., "Expert Behavior and Problem Representation," in *Artificial and Human Intelligence*, Elithorn and Banerji (Editors), North-Holland Publishing Co. (to appear); also available as Technical Report CBM-TR-126, LCSR, Rutgers University, 1982.
- Simon, H.A. and Lea G., "Problem Solving and Rule Induction: A Unified View," in *Knowledge and Cognition*, Gregg (Editor), Lawrence Erlbaum, 1974.

Mechanisms for Qualitative and Quantitative Discovery¹

Pat Langley
Jan Zytkow
Herbert A. Simon
Gary L. Bradshaw
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213 USA

ABSTRACT

Scientific discovery is a complex activity involving many different components. Our interest in discovery has led us to construct four AI systems that address different facets of the scientific process. BACON.6 focuses on discovering empirical laws that summarize numerical data. This program includes heuristics for finding numerical relations in noisy data, postulating intrinsic properties such as mass and specific heat, and noting common divisors. GLAUBER is concerned with discovering laws of qualitative structure, such as the hypothesis that acids react with alkalis to form salts. It includes heuristics for noting recurring patterns, for defining classes based on those patterns, and for combining classes that have many elements in common. STAHL focuses on determining the components of substances involved in reactions, and has been used to model the reasoning that led to the phlogiston theory. DALTON is concerned with formulating molecular models that explain reactions such as those occurring in chemistry and particle physics. It carries out a search through the space of possible models, using information it has acquired in other contexts to direct its search down promising paths. While each of these discovery systems is interesting in its own right, we are also exploring ways in which the systems can interact to help direct each other's search process.

Key Terms: scientific discovery, quantitative laws, qualitative laws, chemical reactions, classification, theory of acids and bases, phlogiston theory, atomic theory, molecular models

Introduction

Science is a complex endeavor, involving such diverse activities as designing experiments, inventing new measuring devices, discovering empirical laws, formulating explanations, and testing hypotheses. Until recently, philosophers and historians of science have focused on the criteria for verification and testing of hypotheses, which were amenable to logical analysis. The discovery process was largely ignored, since there seemed no logically valid methods for approaching this elusive activity. However, in recent years, researchers in computer science and cognitive psychology have developed powerful new approaches to modeling thought. The most central ideas of this new methodology are the notion of *search* through a problem space, and the notion of *heuristics* for directing that search. Although these concepts were originally developed with respect to problem solving, they are equally applicable to explaining the mechanisms that lie behind scientific discovery.

In this paper we examine scientific discovery in the light of such notions as search and heuristics. Of course, there are many different types of discovery, ranging from finding numerical laws, to classifying similar objects and events, to formulating structural models to explain regularities. In the following pages, we address each of these forms of discovery. Our approach has been to consider some period from the history of science, and then to construct an AI system capable of rediscovering laws that were first found during that period. Thus, while we have not attempted to replicate the historical discovery process in detail, our ideas have been influenced by the historical data. We have organized the paper around four discovery programs that we have developed in this manner. In each case, we discuss the historical context, describe the system in terms of the heuristics it employs, and consider the manner in which these heuristics let the system discover the historical laws. Most of our examples are drawn from the history of chemistry, but none of the systems are limited to this domain. In closing, we discuss the relations between the various programs, and examine the potential for constructing an integrated discovery system that includes these programs as components.

Discovering Quantitative Empirical Laws

At the very end of the 18th century and the beginning of the 19th, three fundamental discoveries were made that shaped the directions of chemical research for several generations thereafter. The first of these was Proust's (1799) statement of the law of constant proportions. Proust conducted a painstaking analysis of chemical compounds, finding that the ratio of the combining weights of the constituent elements was always constant for a particular compound. The second fundamental advance was Dalton's (1804) introduction of the law of multiple proportions. This law asserts that when two elements combine to form several different compounds, the ratios of their combining weights are always small integer multiples of one another. The third advance was Gay-Lussac's (1809) discovery of the law of combining volumes for gaseous reactions, which states that gases combine with each other in very simple ratios by volume.

These three discoveries provided the foundation for a quantitative theory of chemical reactions, and ultimately led to the determination of the atomic weights of the elements. To the extent that the work of Dalton and Gay-Lussac was motivated by the atomic hypothesis, these discoveries were not products of pure data-driven induction but were partly guided by theory. However, we shall see that data-driven induction is sufficiently powerful to attain the laws without making use of a full-blown atomic hypothesis.

¹ Reprint requests should be sent to Pat Langley, The Robotics Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213. This work was supported by Contract N00014-82-0168 from the Office of Naval Research.

Finding Numerical Relations in Noisy Data

We have explored the process of quantitative discovery through BACON.6, the sixth in a line of programs named after Sir Francis Bacon (1561-1626). Given a set of independent variables, BACON.6 varies one of the terms, looking for relations between that term and some dependent variable. Once a functional relation has been found, the parameters in that function are given the status of dependent terms at a higher level of description. When the system varies the next independent term, it looks for a relation between that variable and the new higher level terms. This process continues, with BACON.6 recursing to higher levels of description, until all the independent terms have been incorporated into a complex quantitative relationship. We will not discuss this process in any great detail, since it has been described for earlier versions of BACON elsewhere [Langley, 1981, Langley, 1982].

Unlike previous versions, the current system is capable of dealing with significant amounts of noise in its data. The program uses a differencing technique to find the best polynomial function for relating two terms. However, it also considers polynomial relations between *functions* of these terms, so that relations such as $y^3 = ax^2 + bx + c$, and $\sin(y) = a \log(x) + b$ can be found. The differencing method accepts any relation that accounts for more than a user-specified percentage of the variance in the data. When this requirement for explained variance is high, BACON.6 behaves much like its predecessors: if the data are noise-free, it generates only a single hypothesis; however, if the data are noisy, it fails to find any relation at all. In contrast, when the setting is low, the system tends to generate a number of alternate hypotheses, whether the data are noisy or not.

In cases where a number of competing hypotheses have been generated, BACON.6 must have some way to order these hypotheses. In addition to the explained variance, the system takes into account the *complexity* of each law, measuring this by the number of terms that make up the polynomial expression. The user can specify the exact role played by the two criteria, but in our experiments with the system, we have found that the *ratio* of explained variance to complexity gives good results. Once the score for each hypothesis has been calculated, a threshold score is computed, and only those laws exceeding the threshold are retained. BACON.6 also takes the *generality* of each law into account. For example, if two laws are found to describe the relation between y and z when $x = 1.0$, but one of these laws does not fit well when $x = 2.0$, then that law will be rejected. In this way, the system ensures that only hypotheses holding across a broad range of data are retained.

Postulating Intrinsic Properties

While the above heuristics are useful for discovering relations between numerical terms, they cannot be used to relate *nominal* or symbolic independent terms to numeric dependent variables, and this is precisely the situation in which the early chemists found themselves. For instance, the independent terms in Proust's, Dalton's, and Gay-Lussac's chemical experiments were the elements or compounds involved, while the dependent terms were numerical measures such as weight or volume. In such cases, BACON.6 defines *intrinsic properties* that take on numeric values, and then associates these properties with the nominal terms.

ELEMENT ₁	ELEMENT ₂	COMP.	W ₁	W ₂	W ₂ /W ₁
NITROGEN	OXYGEN	NO	1.0	1.14	1.14
NITROGEN	OXYGEN	NO	2.0	2.28	1.14
NITROGEN	OXYGEN	NO	3.0	3.42	1.14
NITROGEN	OXYGEN	N ₂ O	1.0	0.57	0.57
NITROGEN	OXYGEN	N ₂ O	2.0	1.14	0.57
NITROGEN	OXYGEN	N ₂ O	3.0	1.71	0.57
NITROGEN	OXYGEN	NO ₂	1.0	2.28	2.28
NITROGEN	OXYGEN	NO ₂	2.0	4.56	2.28
NITROGEN	OXYGEN	NO ₂	3.0	6.84	2.28

Table 1. Determining the combining weights for reactions.

Let us consider the role of intrinsic properties in BACON.6's rediscovery of the early chemical laws. Given control over the substances entering and resulting from a reaction, as well as the weight of the first substance that is used, the system gathers data like those shown in Table 1. Upon varying the amount of oxygen used to form nitric oxide (NO), the program discovers that the two weights w_1 and w_2 are linearly related with a slope of 1.14 and an intercept of zero. Upon varying the output of the reaction, BACON.6 then examines the weight relations for the compound nitrous oxide (N₂O). In this case, the law is also linear, but the slope has changed to 0.57. A similar result is obtained when the system examines the values for nitrogen dioxide, and in this case the slope is 2.29.

The slopes that BACON.6 finds in these experiments are closely related to the weight ratios found by Proust. Having found these ratios, the program summarizes its results at a higher level of description, as shown in Table 2, and treats these summaries as data. The table also includes the results obtained for two reactions of oxygen and carbon. In this case, the system finds three nominal independent terms and a single numeric dependent variable, so it defines an intrinsic property (say p) whose values are associated with the three nominal values under which they occur. Thus, the value of p for the triple nitrogen/oxygen/nitric oxide would be set to 1.14, the value for nitrogen/oxygen/nitrous oxide would be 0.57, and the value for nitrogen/oxygen/nitrogen dioxide would be 2.29. As stated, these intrinsic values simply store an already known fact, and in this sense they are tautological. However, they can be retrieved in future experiments involving the same chemicals, and used to make predictions or to discover new empirical laws.

Noting Common Divisors

As we have seen, Proust's insight about combining weights laid the groundwork for Dalton's law of multiple proportions. This law stated that in cases where two elements combine to form *different* compounds, the ratios of their combining weights were always small integer multiples of one another. BACON.6 includes a heuristic which lets it discover just such a relation in the data from Table 2. This heuristic operates whenever the system defines a new intrinsic property, and examines the values of the new property to see if they (or their inverses) have a common divisor. This technique is especially useful when intrinsic values are associated with multiple nominal values, as often occurs in chemistry. We have described both the intrinsic property and common divisor methods at length in earlier papers [Bradshaw, 1980, Langley, 1983].

In this case, BACON notes that 1.14, 0.57, and 2.28 have the common divisor 0.57, and would replace these intrinsic values with their corresponding integers 2, 1, and 4. In addition, the program defines a higher level intrinsic property based on the divisors it finds in different situations, and associates the divisors with those cases. Thus, the common divisor 0.57 would be associated with the nitrogen/oxygen pair, while the divisor 1.33 would be associated with carbon and oxygen. These relations are formally equivalent to Dalton's law of multiple proportions. BACON takes a similar path in discovering Gay-Lussac's common divisors for combining volumes, and has even arrived at the correct relative atomic weights of hydrogen, oxygen, and nitrogen from data similar to those in Table 1. Thus, BACON's discovery mechanisms account for many of the quantitative laws found by chemists in the early 19th Century.

ELEMENT ₁	ELEMENT ₂	COMP.	W_2/W_1	P	$W_2/W_1 P$
NITROGEN	OXYGEN	NO	1.14	2	0.57
NITROGEN	OXYGEN	N ₂ O	0.57	1	0.57
NITROGEN	OXYGEN	NO ₂	2.28	4	0.57
CARBON	OXYGEN	CO	1.33	1	1.33
CARBON	OXYGEN	CO ₂	2.66	2	1.33

Table 2. Noting common divisors for chemical reactions.

Expectation-Driven Discovery

The discovery heuristics we examined above can all be characterized as *data-driven* techniques. In other words, although they search a predefined space of hypotheses, the directions followed during that search are largely controlled by the data that are observed. However, the current version of BACON (along with its immediate predecessor, BACON.5) also includes *expectation-driven* discovery heuristics. These range from very simple rules for expecting the same form of law to hold across different contexts, to much more sophisticated assumptions about symmetric laws. The second of these is employed in cases involving two objects, each having the same set of associated variables. For instance, in a collision between two balls, one can measure the initial and final velocities for each object. In such situations, BACON.6 first finds a partial law relating the terms associated with one of the objects, and then assumes that the final law will have a symmetric form. Using this technique, the system has rediscovered Snell's law of refraction, Black's specific heat law, and conservation of momentum. BACON's expectation-driven methods have been described in greater detail in an earlier article, so we will not elaborate on them here [Langley, 1982].

Such heuristics simplify BACON's task in two important respects. First, they allow the system to consider a few alternate hypotheses in a top-down manner, rather than forcing it to systematically search the entire hypothesis space in a bottom-up fashion. Second, these heuristics greatly reduce the data the system must collect in order to determine laws. Since the program expects laws of only a few alternate forms to occur, it need gather only enough observations to determine parameter values and to see which of these fares best. In addition, BACON's expectation-driven methods simplify the discovery process with very little loss in generality. This results from the fact that they obtain their expectations not from domain-specific knowledge, but from relations that have been found by the data-driven methods. This interaction between data-driven and expectation-driven techniques lets BACON take advantage of its early discoveries to direct its search at later stages.

Comments

In the preceding pages, we have described BACON's inputs, outputs, and the heuristics it employs to generate one from the other. By itself, BACON provides an interesting and useful account of the discovery of quantitative empirical laws. However, the system leaves some important questions unanswered. For example, how does BACON decide which variables to employ in the experiments it runs? Similarly, how are the laws generated by the system put to use? In the current version, the representation of data is provided by the programmer, and the laws are simply printed on a terminal screen. One can imagine a version of BACON with an improved user interface, serving as a scientist's aide in analyzing data, and fulfilling a useful function while still requiring its user to design its input and interpret its output. This is one direction in which the system might be extended, and it should probably be done.

However, if one's goal is to understand the nature of scientific discovery, then a more serious answer to the above questions is required. The introduction of expectation-driven heuristics provides a partial answer to the second question, stating that laws found in one context are used to reduce the search for laws in other contexts. However, we know from the history of science that laws also lead eventually to theories and explanations, and BACON has little to say about such aspects of discovery. This suggests that we will find answers to both questions only by studying other facets of the discovery process. Although constructing AI models of these components would undoubtedly be interesting even in isolation, the true advantage will come from exploring the interrelations between different forms of discovery. Our long-range goal, then, should be to understand components of the discovery process whose outputs can be used as BACON's inputs, and to uncover other components which can employ BACON's outputs as their inputs. In the remainder of the paper, we focus on three different models of discovery that we have constructed to this end, and we close with some speculations on possible interactions between the various systems.

Discovering Qualitative Empirical Laws

Although BACON's heuristics are quite useful for discovering quantitative relations, they cannot be applied when only qualitative information is available.² For example, let us consider the state of chemistry before scientists discovered theories involving quantitative measures such as weight and volume. By the 17th and 18th Centuries, chemists had made considerable progress in classifying substances on the basis of qualitative properties. During this period, researchers focused on features such as the taste and texture of substances, as well as their interactions with other substances. Thus, they knew that the substance we now call hydrochloric acid had a sour taste, and that it combined with ammonia to form NH₄Cl (though the structure of this compound was not known). Moreover, they knew that sulfuric acid also tasted sour, and that it also combined with ammonia to form (NH₄)₂SO₄. From such facts as these, the early chemists defined classes such as *acids*, *alkalis*, and *salts*, and formulated laws involving these terms, such as "acids taste sour" and "acids react with alkalis to form salts". Eventually, they came to view both alkalis and metals as special cases of the more abstract concept of a *base*, and arrived at the more general law that "acids react with bases to form salts".

²The intrinsic property heuristic does let one associate numbers with symbols, but some numbers must be present at the outset.

A Qualitative Discovery System

In our efforts to understand the process by which chemists arrived at the notions of acids and bases, and by which they developed theories involving these terms, we constructed a second discovery system that accepts qualitative facts as input. We have named the program GLAUBER, after the 17th Century German chemist who played an important role in developing the concepts of acids and bases. Before considering the program's heuristics, we should examine its representation of data, since this representation is closely linked to its discovery methods.

GLAUBER represents facts using a simple frame-like structure. Each fact contains a *predicate* that specifies the type of fact it is, along with one or more attribute-value pairs. For example, the fact that HCl reacts with NH_3 to form NH_4Cl would be stored as (reacts inputs (HCl NH_3) outputs (NH_4Cl)). Here the predicate is *reacts*, the attributes are *inputs* and *outputs*, and the lists (HCl NH_3) and (NH_4Cl) are the values for these attributes. GLAUBER knows that the order of symbols in a value list does not matter, so that (reacts inputs (NH_3 HCl) outputs (NH_4Cl)) would be considered identical to the above fact. The knowledge that HCl tastes sour would be stored as (has-quality object (HCl) taste (sour)). In this case the values are enclosed in parentheses for consistency with other predicates (such as *reacts*), which may have multiple symbols as values. The reason for not representing this fact in the simpler form (sour object (HCl)) will become apparent shortly.

Like BACON, the GLAUBER program is largely data-driven, in that the data it observes play a major role in determining the hypotheses it generates. GLAUBER's first heuristic searches for recurring patterns in these data; it may be stated as:

**If two facts have the same predicate,
and have the same value for the same attribute,
then create a pattern identical to these facts, but in
which differing values are replaced by class names**

For instance, given the two facts (reacts inputs (HCl NH_3) outputs (NH_4Cl)) and (reacts inputs (H_2SO_4 NH_3) outputs ((NH_4) $_2\text{SO}_4$)), the system would note that the symbol NH_3 occurred in the value of the *inputs* property in both cases. As a result, it would define two classes (say class-1 and class-2), and create an abstract pattern in which the class names replace the differing symbols, but which retains the constant term NH_3 . In this case, the pattern would be (reacts inputs (class-1 NH_3) outputs (class-2)). Given this strategy, GLAUBER must represent the taste of an object as a value rather than as a predicate, if it is ever to note recurring tastes.

As soon as the system creates such a pattern, a second heuristic comes into play, which is responsible for assigning appropriate substances to the classes that have been defined. It may be paraphrased:

**If you have just created a pattern,
then find all facts that fit this pattern,
and add the values which differ in those facts
to the appropriate classes.**

This rule iterates through the list of known facts, checking each one to see if it obeys the new pattern. This occurs when a fact has the same predicate as the pattern, and when all non-class symbols are the same. Minimally, the two facts that first led to the pattern's creation will match, and the symbols that differ in these facts will be added as members of the class mentioned in the pattern. In our example, the chemicals HCl and H_2SO_4 would be stored as members of class-1, while NH_4Cl and ((NH_4) $_2\text{SO}_4$) would be stored with class-2. Since the substance NH_3 occurs in both inputs, it would not be added to any class at this time.

Table 3 presents a set of facts that were generally known by chemists in the 17th Century. This set includes information about reactions between chemicals, as well as information about the tastes of individual substances. When GLAUBER was presented with this set of propositions, it arrived at a class of sour-tasting substances, a class of bitter-tasting ones, a class of salty ones, and some 12 classes defined in terms of reactions with other chemicals.

```
(reacts inputs (HCl NaOH) outputs (NaCl))
(reacts inputs (HCl KOH) outputs (KCl))
(reacts inputs (HCl  $\text{NH}_3$ ) outputs ( $\text{NH}_4\text{Cl}$ ))
(reacts inputs ( $\text{HNO}_3$  NaOH) outputs ( $\text{NaNO}_3$ ))
(reacts inputs ( $\text{HNO}_3$  KOH) outputs ( $\text{KNO}_3$ ))
(reacts inputs ( $\text{HNO}_3$   $\text{NH}_3$ ) outputs (( $\text{NH}_4$ ) $\text{NO}_3$ ))
(reacts inputs ( $\text{H}_2\text{SO}_4$  NaOH) outputs ( $\text{Na}_2\text{SO}_4$ ))
(reacts inputs ( $\text{H}_2\text{SO}_4$  KOH) outputs ( $\text{K}_2\text{SO}_4$ ))
(reacts inputs ( $\text{H}_2\text{SO}_4$   $\text{NH}_3$ ) outputs (( $\text{NH}_4$ ) $_2\text{SO}_4$ ))
(has-quality object (HCl) taste (sour))
(has-quality object ( $\text{HNO}_3$ ) taste (sour))
(has-quality object ( $\text{H}_2\text{SO}_4$ ) taste (sour))
(has-quality object (NaOH) taste (bitter))
(has-quality object (KOH) taste (bitter))
(has-quality object ( $\text{NH}_3$ ) taste (bitter))
(has-quality object (NaCl) taste (salty))
(has-quality object (KCl) taste (salty))
(has-quality object ( $\text{NH}_4\text{Cl}$ ) taste (salty))
(has-quality object ( $\text{NaNO}_3$ ) taste (salty))
(has-quality object ( $\text{KNO}_3$ ) taste (salty))
(has-quality object (( $\text{NH}_4$ ) $\text{NO}_3$ ) taste (salty))
(has-quality object ( $\text{Na}_2\text{SO}_4$ ) taste (salty))
(has-quality object ( $\text{K}_2\text{SO}_4$ ) taste (salty))
(has-quality object (( $\text{NH}_4$ ) $_2\text{SO}_4$ ) taste (salty))
```

Table 3. Chemical facts presented to GLAUBER.

If the early chemists had stopped at this point, their developing science might have remained a mass of isolated regularities. However, they also noted that certain patterns tended to occur together, and GLAUBER uses a third heuristic to notice such co-occurrences:

**If two classes A and B
have enough members in common,
then combine them into a single class C,
and store the patterns for A and B with C.**

This rule would note, for example, that the class of HCl-reacting substances has the same members as the class of HNO_3 -reacting substances, and would define a new class (let us call it the class of *alkalis*) that subsumes both of the original classes. The more specific classes would be abandoned, but their associated patterns would be stored with the alkali concept.

Thus, the alkali class would have two associated patterns - one stating that alkalis tend to react with HCl, and another stating that they also tend to react with HNO_3 . After making this combination, the same heuristic would note that the alkali class and the class of substances that react with H_2SO_4 also have identical members. The process would be repeated, this time replacing alkalis and the H_2SO_4 -reacting class with another class (say *alkalis'*). This new class would have three associated patterns, one referring to reactions with HCl, another to reactions with HNO_3 , and a third to reactions with H_2SO_4 . GLAUBER continues to apply the combination heuristic until all classes with enough members in common have been combined. The exact meaning of the phrase "enough in common" depends

on the setting of a system parameter; however, we have kept it close to 100 percent to avoid the combination of all classes into one superclass.

Note that when two classes with non-identical members are combined, GLAUBER is effectively deciding to incorporate a pattern into the definition of a concept even though that pattern does not hold for all examples of the concept. Thus, if most alkalis tasted bitter – but not all of them – the system would still add this as one of the main characteristics of alkalis, even though exceptions to the rule exist. This is an important characteristic of GLAUBER's approach to discovery, since there are few concepts in any branch of science that do not include some exceptions. In order to avoid overgeneralizations, the system stores the names of substances obeying each particular pattern.

A closely related heuristic applies when the members of one class appear to be a subset of the members of another. In such cases, the subset class is retained, but its associated patterns are copied over to the superset class and stored with it as well. With respect to the acid/alkali data shown in Table 3, this situation occurs for various classes of salts. For instance, GLAUBER defines one class consisting of substances that have salty tastes, and another composed of compounds that result from reactions with HCl. Since the second of these classes is a subset of the first, the latter's pattern is copied (with the first class's name being substituted for the second) and added to the first class's set of associated patterns. Table 3 presents the three main classes that GLAUBER settles on using its two heuristics for combining classes, along with their members and associated patterns; these classes are very similar to the 18th Century chemists' notions of acids, alkalis, and salts.

ACIDS: {HCl, HNO ₃ , H ₂ SO ₄ }
(reacts inputs (acid NaOH) outputs (salt))
(reacts inputs (acid KOH) outputs (salt))
(reacts inputs (acid NH ₃) outputs (salt))
(has-quality object (acid) taste (sour))
ALKALIS: {NaOH, KOH, NH ₃ }
(reacts inputs (HCl alkali) outputs (salt))
(reacts inputs (HNO ₃ alkali) outputs (salt))
(reacts inputs (H ₂ SO ₄ alkali) outputs (salt))
(has-quality object (alkali) taste (bitter))
SALTS: {NaCl, KCl, NH ₄ Cl, NaNO ₃ , KNO ₃ , NH ₄ NO ₃ , Na ₂ SO ₄ , K ₂ SO ₄ , (NH ₄) ₂ SO ₄ }
(reacts inputs (acid NaOH) outputs (salt))
(reacts inputs (acid KOH) outputs (salt))
(reacts inputs (acid NH ₃) outputs (salt))
(reacts inputs (HCl alkali) outputs (salt))
(reacts inputs (HNO ₃ alkali) outputs (salt))
(reacts inputs (H ₂ SO ₄ alkali) outputs (salt))
(has-quality object (salt) taste (salty))

Table 4. Classes retained by GLAUBER after combination.

At this point, GLAUBER has arrived at a set of classes that accurately summarize the data, but its patterns are still fairly specific. Thus, it has defined a class of substances with sour, bitter, and salty tastes, but it does not yet have a general law relating acids, alkalis, and salts. To arrive at such a law, GLAUBER must employ a final heuristic that is very similar to BACON's rule for recursing to higher levels of description; it may be paraphrased:

**If you have found all the patterns that you can,
and combined all the classes that you can,
then treat the known patterns as facts,
and apply the other heuristics to them.**

This rule leads the program to examine the patterns shown in Table 4 and to treat them as if they were data, thus generating new classes and patterns. For instance, given these data, the first heuristic discussed above would note the recurring symbols *alkali* and *salt* in the "facts" (reacts inputs (HCl alkali) outputs (salt)) and (reacts inputs (HNO₃ alkali) outputs (salt)). As a result, it would define a new class (say alkali-reactors) and use it in the higher level pattern (reacts inputs (alkali-reactor alkali) outputs (salt)). After this, the second heuristic would add the chemicals HCl, HNO₃, and H₂SO₄ to this class, since each of them occurs in a second level fact that fits the pattern. Later, after other higher level patterns and classes have been defined, our third heuristic would note that the class of alkali-reactors has the same members as the class of acids, and combine the two categories. When this process is complete, GLAUBER is left with basically the same classes as shown in Table 4, except that each has the associated pattern "acids react with alkalis to form salts".³

When GLAUBER is presented with facts about metals (such as their texture and reactions) as well as information about alkalis, the system follows a somewhat more complex, but basically similar, path. It formulates a class of HCl-reacting substances, consisting of both metals and alkalis, very early on, as well as identical classes based on reactions with the other acids. The class of alkalis (as distinct from metals) is originally defined entirely in terms of taste, while metals are defined in terms of texture. Since both of these classes are subsets of the various acid-reacting classes, their associated patterns are copied over to the supersets. After combining overlapping classes and recursing multiple times, GLAUBER arrives at concepts for alkalis, metals, and bases which are similar to those defined by the early chemists.

Limitations and Extensions

In its present form, GLAUBER has a number of important limitations, and these should be remedied in future versions of the system. For one thing, the existing representation of qualitative laws is ambiguous. For instance, the pattern (reacts inputs (HCl class-1) outputs (class-2)) contains implicit quantifiers on the terms class-1 and class-2, but it is not clear whether these are universal or existential. The following interpretation seems to be a useful one. Whenever GLAUBER generalizes by replacing constant terms with class names, a number of alternate patterns should be produced. In each of these, one of the class names is universally quantified, with the remaining classes being existentially quantified (with the universal quantifier occurring first). As a special case, if only one class in created, it is universally quantified.

Thus, if N classes are introduced, N alternate patterns are created. For instance, in the above example, two statements would result: "for all members X of class-1, there exists a member Y of class-2, such that HCl reacts with X to form Y", and "for all members Y of class-2, there exists a member X of class-1

³We have used the names "acid", "alkali", and "salt" in the table for the sake of clarity, though of course GLAUBER does not generate them.

such that HCl reacts with X to form Y ". Upon comparison with the data, an extended version of GLAUBER might decide that only one of these statements was accurate, or it might decide that both were useful summaries of the observed phenomena. The ambiguity of the current representation has not yet caused difficulties because both interpretations hold for the theory of acids and bases. However, one can imagine domains in which only one interpretation describes the data, and the system's representation of laws and its decision processes should be extended to deal with these cases.

A second limitation is that the existing version fails to find recurring relations between *conjunctions* of facts. For instance, suppose one observed (as Mendel did) that when certain green garden peas were self-fertilized, they produced only green offspring, but that when other green peas were self-fertilized, they produced both green and yellow children. In this case, we would like GLAUBER to divide the green peas into two classes based not on their own directly observable features (since these are identical), but based on the features of their offspring. Thus, in looking for patterns, the system would have to examine not only single facts, but pairs of facts, triples of facts, and so forth. Such a strategy, though much more expensive than the current one, would enable the program to note that some green peas have only green offspring, while others have mixed offspring, and to classify them on this basis.

A final difficulty with GLAUBER relates to the system's search through the space of patterns and classification schemes. The current version completes a run with a number of classes, some having many associated patterns and others having only a few or even one attached pattern (such as the sub-class of salts that are formed from reactions involving HCl). From the early chemists' viewpoint, some of these classes were clearly more important than others, while the GLAUBER treats all classes as equal. Since the system has so far only been tested on carefully crafted data, this strategy has not yet caused any difficulties. However, if the program is ever presented with large set of facts (some of which contain minor or even spurious regularities), it will require some method for focusing attention on some classes in preference to others. In future versions of GLAUBER, we plan to use the number of patterns associated with a class as one measure of importance, since this seems to have been used by chemists in focusing on the concepts of acids, alkalis, and salts rather than on the outputs of reactions involving HCl , NH_3 , and so forth.

Another relevant factor relates to the *empirical content* of a pattern. For instance, consider the statements "acids taste sour" and "acids dissolve metals". The first of these refers to a single property of acids, while the second relates acids to metals. If three substances are known to be acids, while four substances are known to be metals, then the first statement predicts only three facts, while the latter statement predicts twelve different facts. Presumably, patterns that make many predictions should be given greater weight than those making few predictions. Thus, one can imagine measuring the importance of a class as a linear function of its associated patterns, with the empirical content of each pattern acting as coefficients. The details of this scheme remain to be elaborated, but the basic idea of focusing on classes that account for the most data seems to be a plausible approach.

Determining Components of Substances

The forms of discovery we have examined in earlier sections were all concerned with finding empirical laws that summarized observations. However, an equally important form of discovery is concerned with finding *explanations* of phenomena. The distinction between description and explanation is not clear-cut, but the laws generated by BACON and GLAUBER definitely seem to lie at the description end of the spectrum, while such constructs as the phlogiston theory, the atomic theory, and the kinetic theory of gases lie more toward the explanatory end. In our attempt to understand the nature of explanatory discovery, we have implemented two programs that attempt to determine the underlying structure of reactions. In contrast to BACON and GLAUBER, these systems employ knowledge about the nature of reactions to aid in the explanation process, so they are not as general as the earlier empirical discovery programs. In this section we describe STAHL, a system that attempts to determine the components involved in a set of reactions. In the following section we consider DALTON, a program that uses information about the components of substances to construct molecular models of reactions.

Heuristics for Determining Components

A primary goal of the early chemists was to determine the components of various substances, and information about the reactions between substances proved quite useful in this regard. We have implemented a system that infers such components from a set of data about reactions. The program is named STAHL, after G. E. Stahl (1660 - 1734), who played an important role in formulating the phlogiston theory, which was primarily concerned with the components of combustible bodies. Like GLAUBER, this program accepts qualitative information as input, and generates qualitative statements as output. In fact, this information can be represented in the same format, with the input being statements like (reacts inputs (hydrogen oxygen) outputs (water)) and the output being statements like (composed-of substance (water) components (hydrogen oxygen)). However, since STAHL's conclusions relate to the internal structure of substances, they can be viewed as explanations rather than simple descriptive summaries.

As with the earlier systems, STAHL's approach to discovery can be summarized as a set of heuristics. The most basic of these heuristics deals with simple synthesis and decomposition reactions, and lets the system infer the components of a compound. It can be stated:

If A and B react to form C,
or if C decomposes into A and B,
then infer that C is composed of A and B.

An obvious example of this rule's use involves determining the components of water. Given the information that hydrogen reacts with oxygen to form water, STAHL would infer that the latter substance is composed of the first two. Note that STAHL does not draw any conclusions about the *amount* of hydrogen and oxygen contributing to water, but only that they contribute something. Of course, the rule is not limited to reactions involving pairs of elements, but can also deal with cases in which three or more substances unite to form a compound.

If all chemical reactions were of the simple form shown above, STAHL's task would be simple indeed. However, more complex reactions are common in chemistry, so STAHL includes additional heuristics for dealing with these more complex situations. The purpose of these rules is to simplify such reactions, with the goal of transforming them into a form that can be matched by the synthesis/decomposition rule shown above. One such rule is responsible for "canceling" out substances occurring on both sides of a reaction; this "reduction" rule can be paraphrased:

**If A occurs on both sides of a reaction,
then remove A from the reaction.**

This heuristic leads directly to a simplified version of a reaction. For instance, if STAHL is told that "A, B, and C react to form D and C", this rule would apply, giving the simplified reaction "A and B react to form D". This revised relation would then be used by the synthesis/decomposition rule to infer that D is composed of A and B. One can imagine cases in which this approach would lead to errors, as when different amounts of a substance are observed before and after a reaction. However, one can also imagine more conservative versions of the heuristic that require equal amounts of the canceled substance to occur on each side.

STAHL incorporates a third heuristic that initially leads to more complex statements of reactions, but may make it possible for the reduction rule to apply. This heuristic draws on information about the components of a substance that have been inferred earlier; it can be stated:

**If A occurs in a reaction,
and A is composed of B and C,
then replace A with B and C.**

For instance, the system may know that X is composed of Y and Z, and that "X reacts with W to form V and Z". In this case, the substitution rule would let us rewrite the second relation as "Y, Z, and W react to form V and Z". Given this formulation, the reduction rule would lead to "Y and W react to form V", and the synthesis/decomposition rule would lead to the conclusion that V is composed of Y and W. As before, this rule is not restricted to substances composed of two elements, but works equally well for more complex structures.

Two final heuristics let the system determine when two substances that were originally thought to be different are in fact identical. The first of these rules may be stated:

**If A is composed of B and C,
and A is composed of B and D,
then identify C with D.**

This heuristic applies when one learns that a compound can be decomposed in two different ways, but in which those decompositions differ by a only single substance. The second rule is very similar, except that it applies when two apparently different compounds are found to have the same components. It can be paraphrased:

**If A is composed of C and D,
and B is composed of C and D,
then identify A with B.**

The history of chemistry abounds with cases in which a new substance was discovered in two different contexts, was originally thought to be two distinct substances, and was eventually combined into a single concept. We will see an example of such identification shortly.

One of the interesting features of STAHL is the manner in which its heuristics interact. Note that the substitution rule requires knowledge of a substance's composition, so that some inferences about composition must be made before it can be

used. However, we have also seen that complex reactions must be rewritten by the reduction and substitution rules before some composition inferences can be made. This interdependence leads to a "bootstrapping" effect, in which inferences made by one of the rules enables further inferences to be made, these allow additional inferences, and so forth, until as many conclusions as possible have been drawn. This process generally begins with one or more simple reactions, but after this the particular path taken depends on the data available to the system. Below we consider a simple example of STAHL's behavior in relation to the phlogiston theory.

Discovering the Phlogiston Theory

The theory of phlogiston originated early in the 18th Century, and after undergoing several transformations, was widely accepted until the 1780's. This theory adopted the ancient view that fire, heat, and light are different manifestations of a common principle that leaves a body during combustion. Therefore, any reaction involving combustion was viewed as a decomposition; for instance, burning coal was interpreted as decomposing into the matter of fire (another term for phlogiston) and ash.⁴ Early phlogisticians were not able to isolate phlogiston, but the disengagement of fire during combustion seemed to be a good observational reason for admitting the disengagement of a substance from the burning body. Later, as the notion of phlogiston proved useful in explaining many additional reactions, the existence of this substance was supported by a substantial body of evidence.

After they began to study combustion within closed vessels, chemists realized that air was necessary for combustion to occur. However, they did not assume that air changed its chemical identity during this process. Rather, they decided that air played an auxiliary role, similar to that played by water in reactions involving acids, alkalis, and salts. Thus, even starting with empirically more complete descriptions of combustion, such as "in the presence of air, carbon burns to release phlogiston and to form ash", they employed the reduction heuristic to remove air and simplify the relation. Given such data, STAHL makes similar "errors" in reasoning, so that it provides a simple explanation of the process by which chemists developed phlogiston-based models of combustion reactions. Such confusions are common in the history of chemistry, and a similar error led the followers of Lavoisier (around 1810) to believe that sodium was a compound of soda and hydrogen.

Let us examine the path taken by STAHL in arriving at results that are very similar to the phlogiston theory. Suppose we present the system with two facts: (reacts inputs (coal air) outputs (matter-of-fire ash air)) and (reacts inputs (calx-of-iron coal air) outputs (iron ash air)).⁵ One may question the exact representation of these facts, but clearly something very much like them was believed during the period in which the phlogiston theory was developed. Given this information, STAHL would immediately apply its reduction heuristic, giving the revised reactions (reacts inputs (coal) outputs (matter-of-fire ash)) and (reacts inputs (calx-of-iron coal) outputs (iron ash)). The first of

⁴Several decades later, in the second half of the 18th Century, fixed air (carbon dioxide) was discovered, and recognized as the product of burning coal in place of ash.

⁵Calx of iron was the current name for iron oxide; we have used the original terminology because the modern term is based on the oxygen theory developed by Lavoisier.

these revisions, combined with the synthesis/decomposition rule, leads directly to the inference that coal is composed of matter-of-fire (or phlogiston) and ash, which was one tenet of the early phlogiston theory. Having arrived at this conclusion, STAHL would apply the substitution rule, generating the expanded relation (reacts inputs (calx-of-iron ash matter-of-fire) outputs (iron ash)). At this point, the reduction rule is used to remove ash from both sides of the equation, giving (reacts inputs (calx-of-iron matter-of-fire) outputs (iron)). Finally, the synthesis/decomposition heuristic leads STAHL to infer that iron is a compound composed of calx-of-iron and the matter of fire.

Next, let us consider how STAHL employs its identification heuristics with respect to the phlogiston theory. Suppose we give the system the following additional data: (reacts inputs (iron sulfuric-acid water) outputs (vitriol-of-iron inflammable-air water)) and (reacts inputs (calx-of-iron sulfuric-acid water) outputs (vitriol-of-iron water)).⁶ Given these facts, STAHL removes the water from both reactions using the reduction rule. This sufficiently simplifies the second reaction so that one can apply the synthesis heuristic, inferring that vitriol-of-iron is composed of calx-of-iron and sulfuric-acid. This fact is substituted into the first reaction, giving (reacts inputs (iron sulfuric-acid) outputs (calx-of-iron sulfuric-acid inflammable-air)). After eliminating sulfuric-acid from both sides of this expression, STAHL infers that iron consists of calx-of-iron and inflammable air. However, the system knows from the other reactions described earlier that iron can also be decomposed into calx-of-iron and phlogiston. Using the first of its identification heuristics, the system infers that inflammable-air and phlogiston are identical. Both the reasoning and conclusions of STAHL in this example are the same as those of Cavendish and other phlogiston theorists during the 1760's.

Constructing Molecular Models

Once the substances involved in a reaction have been determined, one can begin to construct molecular models of the reaction, in which one specifies the number of times each substance occurs in the transformation. After John Dalton set forth the general form of his atomic hypothesis in 1808, the design of such molecular models became of central importance to chemists. Dalton employed his *rule of greatest simplicity* to apply the atomic theory to specific cases, and though this heuristic worked in many cases, it led him to incorrect conclusions in others. For instance, it led him to conclude that water was composed of a single hydrogen atom and a single oxygen atom. In contrast, Avogadro (1811) employed Gay-Lussac's data on combining volumes, along with the assumption that equal volumes of gases contained equal numbers of particles. Using this information, he inferred diatomic models for hydrogen and oxygen and a different structure for water. Although we accept Avogadro's hypothesis today, it was rejected by his contemporaries, since they believed that different atoms of the same element would repel, rather than attract, each other. The explanation for diatomic molecules did not appear until well into the 20th Century, when the theory of covalent bonds was developed.

⁶The formula for vitriol of iron is FeSO_4 , while the modern name for inflammable air is hydrogen.

Searching the Space of Models

In order to further understand the process by which the early chemists arrived at such models, we implemented another discovery system that constructs molecular models of reactions. We have named this program DALTON, since Dalton was the first proponent of an atomic theory that explicitly used structural models. DALTON knows that two quantities are important in any reaction – these are number of molecules of each substance that take part (in the simplest form of the reaction), and the number of particles in each molecule.⁷ Suppose the system is told that the element hydrogen reacts with the element oxygen to form the compound water, and is asked to construct a model of the components of this process. In this case, the program must determine the number of hydrogen, oxygen, and water molecules, and the internal structure of each type of molecule. The system operates by starting with a very abstract model in which no commitments are made, and successively refines this model as it proceeds. Let us examine what happens at each stage in the search through the space of models.

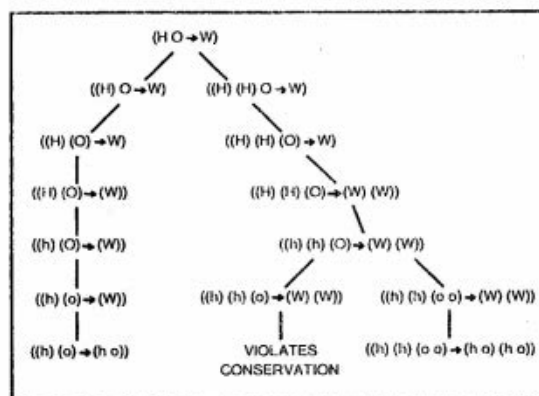


Figure 1. DALTON's search for a model of the water reaction.

Starting with an abstract model of the form $(H O \rightarrow W)$, the program first considers the number of hydrogen molecules involved. Lacking any theoretical bias, the system chooses the simplest hypothesis and assumes a single hydrogen molecule is required. If this choice later causes difficulty, the model-builder can back up and try another path. Similar initial choices are made for oxygen and water, so that the partially specified model includes one molecule each. This is represented by the proposition $((H) (O) \rightarrow (W))$, in which each molecule is enclosed in parentheses.

Now DALTON must determine the internal structure of each type of molecule, and it decides to assume initially that both hydrogen and oxygen consist of a single elementary particle (say h and o), giving the model $((h) (o) \rightarrow (W))$. At this point, the program invokes the strong theoretical assumption that the total number of particles in any reaction is conserved. This assumption is stated in terms of a condition-action rule that checks to see if the model can be finalized in such a way that

⁷Thus, the program begins with a better notion of the true situation that did its namesake, since Dalton did not make the distinction between atoms and molecules.

conservation is obeyed. If this is possible, DALTON outputs the completed model and halts, but if the conservation principle cannot be satisfied, the system backs up and considers other possibilities. In this case, the conservation heuristic tells DALTON that the water molecule must be composed of h and o particles, and that the final model must have the form ((h) (o) → (h o)). This model is equivalent to the one originally formulated by the human Dalton. Figure 1 presents some of the paths available in the space of molecular models. In arriving at the monatomic model just described, DALTON takes the left path, and since this leads to an acceptable solution, no backtracking is required.

Directing the Search Process

As we have seen, DALTON's basic strategy is to carry out a depth-first search through the space of models, ordering the search so that simple models are considered first. However, when enough of the model has been specified, a theory-driven heuristic (implementing the conservation assumption) takes over and finalizes the model. DALTON can also employ theory-driven methods at other stages in its search process, and these methods can alter the system's behavior in significant ways. Thus, in the above run, the system had no theoretical biases other than a belief in conservation of particles and a desire to construct as simple a model as possible. However, if we give DALTON some additional information about the water reaction, its behavior changes drastically. Avogadro was aware of Guy-Lussac's results, and firmly believed that the combining volumes he observed were related to the number of molecules involved in the reaction. To model this knowledge, we can add the heuristic:

If you want to know the number of molecules of X
that are involved in a reaction,
and the combining volume of X was V,
then consider only multiples of V as possibilities.

Given this assumption (and knowledge of the combining volumes), our program would instead postulate two molecules of hydrogen and water (and if this was later found to be unsatisfactory, four and then six), while it would retain the assumption of one oxygen molecule. Thus, at the third level in the search tree, DALTON would have the partially specified model ((H) (H) (O) → (W) (W)).

At this point the system would move to consider the internal structure of the hydrogen molecule, assuming it was composed of a single particle (say h), and would make a similar assumption for oxygen. However, for the resulting model, ((h) (h) (o) → (W) (W)), there exists no decomposition of water in terms of h and o that satisfies the conservation assumption, so the program would back up and consider some other alternative. DALTON would next hypothesize the oxygen molecule as composed of two particles, and since this does allow conservation to be satisfied, a final model would be constructed in which oxygen is diatomic and hydrogen is monatomic: ((h) (h) (o o) → (h o) (h o)). (These two search paths are shown on the right side of Figure 1.) While this model differs from the modern-day one, it is consistent with Guy-Lussac's data and encounters difficulty only when other reactions are considered. For example, the monatomic assumption does not work for the ammonia reaction.

Like most of the programs we have described, DALTON is stated as a production system. In default mode, the system uses a few simple rules to formulate simpler models first, and more complicated ones as necessary. However, if new condition-action rules are added to the system, they take precedence over

the default rules and can direct search down paths that might otherwise not be considered. Thus, one can insert a rule that would match if the combining volume of some substance is known, and use this information to determine the number of molecules used for that substance in the model. The conservation assumption is implemented in a similar fashion, so that it generates a molecular structure of a reaction's output that uses all particles occurring in the input.

Once DALTON has generated a successful model for a reaction, it converts this knowledge into productions. For instance, having arrived at the diatomic explanation of water given above, the program would store one rule concerning the molecules of hydrogen involved, another for oxygen molecules, and a third for the water molecules. If the system is asked to explain the water reaction at a later date, it will be able to recall the number of molecules without search. DALTON also constructs productions describing the internal structure of various molecules, and while this knowledge is useful in re-explaining the water reaction, it is useful in other cases as well. For instance, when asked to model the ammonia reaction, the system would immediately guess that hydrogen was monatomic, based on the success of this assumption in its model of water. None of the models incorporating this assumption satisfy conservation, so the system would back up, hypothesize that hydrogen is instead diatomic, and eventually arrive at the correct model for ammonia.

However, DALTON must also update its model of the water reaction. Since the system knows that its monatomic hydrogen rule was responsible for leading it astray, it removes this rule from memory and replaces it with a diatomic rule for hydrogen. It then retrieves the reaction that led it to construct the monatomic production, and checks to see if the replacement rule works here as well. In this case it does, but had it run into difficulty, the process would have been repeated, with DALTON considering ever more complicated molecular structures (up to a limit), until both the water and the ammonia reaction had been successfully explained by a single rule.

Extensions of the Approach

Although DALTON's methods are concerned with reactions, they are not limited to the chemical domain. For example, the field of elementary particle physics is also concerned with reactions, and with the formulation of "molecular" models to explain those reactions. The most widely accepted theory in this domain accounts for the internal structure of protons, neutrons, and other hadrons in terms of a small set of hypothesized particles called "quarks". In its present form, DALTON cannot discover the quark theory, but two relatively simple extensions should enable the system to arrive at the basic tenets of this framework.

First, the current version of DALTON assumes that molecules of elements (such as hydrogen and oxygen) are composed of a single type of particle (such as hydrogen atoms or oxygen atoms), and this assumption considerably simplifies the search process in the chemical domain. However, there are no directly observable "elements" in the field of particle physics, and in order to explain particle interactions, one must postulate more complex models of hadrons. For example, the basic proton "molecule" is viewed as composed not of three proton "atoms", but as composed of two u quarks and one d quark. In order to regenerate the quark theory, DALTON must be modified to search the larger space of models in which such decompositions can occur.

An equally important aspect of the quantum theory involves the conservation of mass and of the various quantum numbers, such as spin and electric charge. In order to generate these features of the theory, DALTON must attempt to explain quantitative attributes of directly observable substances in terms of attributes associated with inferred substances such as quarks. Presumably, these constraints can be stated as theory-driven heuristics, much as the conservation of particles assumption is implemented in the current version.

A less obvious application of DALTON involves the field of classical genetics. For example, the hereditary rules for garden peas, first enumerated by Gregor Mendel in 1866, can be viewed as reactions in which characteristics of the parents are transformed into characteristics of the offspring. Given the first extension described above, along with a suitable replacement for the conservation assumption (since this does not apply in reproductive systems), DALTON should be able to arrive at the two-trait model originally formulated by Mendel. Suppose that the system is provided with genotypic statements of the result of inbreeding and crossbreeding, which might be induced by another discovery system (like GLAUBER) from phenotypic descriptions of these reactions.

For example, if we let G stand for green peas that produce only green offspring, Y stand for yellow peas that produce only yellow offspring, and G' stand for green peas that produce mixed offspring, then four basic reactions suffice to describe Mendel's observations: $G \rightarrow G$, $Y \rightarrow Y$, $G Y \rightarrow G'$, and $G' G' \rightarrow G' G'$. Given these reactions, an extended version of DALTON should be able to infer that two primitive traits (say g and y) are required, and to decide that the genotype G can be modeled by the "molecular" pair ($g g$), that Y can be modeled by the pair ($y y$), and that G' can be modeled by the pair ($g y$). As we envision it, the system's explanation of these reactions would not involve the notion of dominance, nor would it predict the proportions in which the various genotypes are observed, but it would account for the basic qualitative relations between parents and offspring.

Toward an Integrated Discovery System

While each of the systems described above accounts for an important part of the discovery process, a more profitable long-range goal would be to combine these programs into a single, integrated discovery system. One advantage of this approach is that it would increase our understanding of the relations between the various forms of discovery. Another benefit is that it would decrease the degrees of freedom in the individual programs, since the outputs of some would be forced to conform to the inputs of others. In this section, we consider the potential for constructing such a unified system, by exploring the possible interactions between pairs of the individual programs described earlier.⁸

GLAUBER and BACON

Let us begin by examining the potential connections between our two empirical discovery systems, GLAUBER and BACON. Since the first of these deals entirely with symbolic data and the second generates numerical laws, there seems no manner in which BACON's output could be used by GLAUBER. However, the possibilities in the other direction appear much better. The

reader will recall that GLAUBER produces two forms of symbolic knowledge – abstract laws and hierarchies of classes used in those laws – while BACON employs symbolic values for some of its independent terms. One can imagine BACON making two uses of GLAUBER's output. First, it might employ the qualitative laws generated by the latter to design its own experiments, rather than rely on its programmer for this information. For example, suppose BACON were told by GLAUBER that nitric oxide, nitrous oxide, and nitrogen dioxide were all substances that resulted from reactions between nitrogen and oxygen. Given knowledge of this class of compounds, an extended version of BACON might design an experiment in which the substances entering a reaction (oxygen and nitrogen) were held constant, while the output of the reaction was varied. If quantitative variables such as the weights of the substances were examined, the resulting experiment would lead BACON to Dalton's law of multiple proportions (as described in an earlier section).

The second use of GLAUBER's output relates to BACON's generalization process. As it is currently implemented, BACON initially associates intrinsic values with all potentially relevant symbolic conditions, and generalizes by removing conditions whenever it finds that a set of intrinsic values are useful in a new context. However, the availability of the classification hierarchy generated by GLAUBER presents an alternate generalization method. Rather than removing conditions entirely, one can generalize by replacing the symbolic value in a condition with the name of a class containing that value. For instance, suppose BACON has stored a set of intrinsic values, and one of the conditions for retrieval is that one of the substances entering the reaction is HCl. Now suppose that the system finds these intrinsic values useful when the substance is H_2SO_4 instead of HCl. Rather than inferring that this condition is irrelevant, BACON might decide that the intrinsic values should be retrieved whenever an *acid* is involved in the reaction (provided that the concept of acid has previously been defined by GLAUBER). This is a more conservative approach to generalization, and would allow BACON to express a larger class of hypotheses than it currently can. Of course, the system could eventually decide to remove this condition entirely, should the intrinsic values prove useful for non-acids as well.

GLAUBER and STAHL

We have seen that GLAUBER treats the symbols *reacts*, *inputs*, and *outputs* in the same manner that it treats other symbols, while STAHL gives them very special consideration due to its knowledge about reactions. Nevertheless, both systems are capable of dealing with information about particular reactions, so there is considerable room for interaction. Let us first consider the path from GLAUBER to STAHL. Although in our examples of GLAUBER's operation, we have assumed data like (reacts inputs (hydrogen oxygen) outputs (water)), note that substances like hydrogen, oxygen, and water are not really primitive concepts. Rather, these names represent *classes* of objects that share certain recurring features and enter into similar relations. Given instances of substances such as these, GLAUBER would have no difficulty in formulating these classes and finding patterns such as (reacts inputs (hydrogen oxygen) outputs (water)). In this revised scheme, GLAUBER would generate the patterns (which we originally viewed as data) that STAHL requires as input. Similarly, STAHL can also accept higher level patterns as input, and make inferences about the structures of higher level classes. For instance, given the pattern (reacts inputs (acid base) output (salt)), the system would decide that salts are composed of acids and bases.

⁸We will not discuss the relation between BACON and STAHL, or that between GLAUBER and DALTON, since no obvious interactions present themselves.

Interactions can occur in the other direction as well. STAHL's major outputs are conclusions about the components of substances, and there is nothing to prevent this information from being represented in a form that can be inspected by GLAUBER. For example, suppose that STAHL determined that HCl was composed of H and Cl, while H_2SO_4 was composed of H and SO_4 . These inferences might be represented as (has-components object (HCl) components (H Cl)) and (has-components object (H_2SO_4) components (H SO_4)). Given this information, GLAUBER would create a class containing the substances HCl and H_2SO_4 , which it might then combine with the acid class. This would lead to the pattern (has-components object (acid) components (H acid-components)), which is equivalent to stating that acids have hydrogen as one of their components. Thus, useful knowledge could flow in both directions between these two systems.

STAHL and DALTON

As we have seen, STAHL focuses on determining the components of chemical substances, while DALTON is concerned with the number of particles involved in a reaction and with their internal structure. Thus, STAHL can be viewed as laying the groundwork for a detailed structural model, with DALTON being responsible for finalizing the model. In fact, DALTON employs knowledge about the components of a substance in testing its conservation assumption, and it is natural to assume that this information comes from STAHL. However, since DALTON's outputs are actually more detailed versions of the models produced by STAHL, it is unlikely that any communication from DALTON to STAHL would be useful.

BACON and DALTON

Although we have seen how DALTON's search through the space of molecular models can be altered by theoretical assumptions stated as productions, we have not discussed the origin of the data matched by those rules. For instance, Avogadro's heuristic relied on knowledge of the combining volumes of gases. As the reader will recall, such information can be generated by BACON's common divisor heuristic, and it is interesting to think that BACON's data-driven heuristics could lead to theory-driven behavior in a system such as DALTON. The origin of DALTON's theory-driven heuristics is more difficult to explain, and we have no plausible account for them at present. Since DALTON's outputs are molecular models and BACON's inputs are numerical data, we foresee no major information passing from the former to the latter, though it is conceivable that DALTON's models might lead to predictions of empirical laws which BACON could then test.

Conclusions

In this paper, we have examined four AI programs that address different facets of the discovery process. The details of the individual systems should by now be clear, and we hope the reader has gained some insight into our long-range plans through our discussion of the possibilities for an integrated discovery system. Even within our expanded framework, many important aspects of discovery have been omitted, such as the formulation of mechanistic explanations, and the role of structural analogies [Winston, 1980, Gentner, 1982]. Thus, even our goal of an integrated discovery system involving BACON, GLAUBER, STAHL, and DALTON is limited in some important

respects. However, limiting one's focus of attention is a venerable and useful tradition in the history of science, and there will be ample time to incorporate these additional facets of discovery after we better understand the relations between the four existing systems.

Before closing, we should mention some of the earlier AI research on discovery, and its relation to the programs we have described. BACON is most closely related to Gerwin's [1974] heuristic curve-fitting system, though BACON can deal with more complex functions and employs a different set of heuristics. GLAUBER bears some similarity to Brown's [1973] system for discovering kinship relations, though this focused on eliminating redundancies from a set of facts, rather than on forming abstract laws. GLAUBER is somewhat more closely related to Michalski and Stepp's [1981] conceptual clustering program, which also generates classification schemes based on qualitative data, though the details of the two systems differ. DALTON's search for molecular models can be viewed as a simplified version of DENDRAL's [Feigenbaum, 1971] search for organic compounds to explain mass spectrograms, though again the details of these programs are quite different. We are not aware of any obvious analogy in the AI literature to STAHL's inference rules, so this system appears to incorporate a unique approach to scientific discovery. Thus, while most of our discovery systems are related to earlier work in the area, they also differ in significant ways, and unlike most earlier programs, show potential for being combined into a more complete, integrated theory of discovery. This approach is perhaps most akin to Lenat's [1977] account of mathematical discovery, in which a variety of heuristics interact in complex ways to generate an ever richer set of concepts.

In conclusion, we now have four independent AI discovery systems. The first of these, BACON, focuses on the discovery of quantitative empirical laws, and has been extensively tested in a number of different domains. Of the other systems, GLAUBER is most similar to BACON in that it relies on very general, data-driven heuristics and generates summaries of the qualitative data it observes. STAHL and DALTON incorporate more domain-specific knowledge, since they employ knowledge of reactions, but this does not mean they are limited to the chemical domain. These systems can be viewed as producing simple explanations, since they construct models of the data they receive as input. The three new systems are not nearly such finished products as BACON, and they should be extended in directions that we have already discussed. Once this has been accomplished, we will be ready to attempt combining the programs into an integrated discovery system that involves significant interactions between its components. Whether or not we succeed in this effort, we feel that it will lead us to both a deeper and broader understanding of the process of scientific discovery.

References

- Bradshaw, G., Langley, P., and Simon, H. A. BACON.4: The discovery of intrinsic properties. *Proceedings of the Third National Conference of the Canadian Society for Computational Studies of Intelligence*, 1980, 19-25.
- Brown, J. S. Steps toward automatic theory formation. *Proceedings of the Third International Joint Conference on Artificial Intelligence*, 1973, 20-23.
- Feigenbaum, E. A., Buchanan, B. G., and Lederberg, J. On generality and problem solving: A case study using the DENDRAL program. In *Machine Intelligence 6*, Edinburgh: Edinburgh University Press, 1971.
- Gentner, D. A structure-mapping approach to analogy and metaphor. *Proceedings of the International Conference on Cybernetics and Society*, 1982, 75-79.
- Gerwin, D. G. Information processing, data inferences, and scientific generalization. *Behavioral Science*, 1974, 19, 314-325.
- Langley, P. Data-driven discovery of physical laws. *Cognitive Science*, 1981, 5, 31-54.
- Langley, P., Bradshaw, G., and Simon, H. A. Data-driven and expectation-driven discovery of empirical laws. *Proceedings of the Fourth National Conference of the Canadian Society for Computational Studies of Intelligence*, 1982, 137-143.
- Langley, P., Bradshaw, G., and Simon, H. A. Rediscovering chemistry with the BACON system. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence View*, Palo Alto, CA: Tioga Press, 1983.
- Lenat, D. B. Automated theory formation in mathematics. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, 1977, 833-842.
- Michalski, R. S. and Stepp, R. E. An application of AI techniques to structuring objects into an optimal conceptual hierarchy. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, 1981, 460-465.
- Winston, P. H. Learning and reasoning by analogy. *Communications of the ACM*, 1980, 23, 689-703.

COGNITIVE ECONOMY In a Fluid Task Environment

Douglas B. Lenat
C.S. Dept., Stanford U.
Stanford, Ca.

Frederick Hayes-Roth
Teknowledge, Inc.
Palo Alto, Ca.

Philip Klahr
The Rand Corporation
Santa Monica, Ca.

ABSTRACT

Intelligent systems can explore only tiny subsets of their potential external and conceptual worlds. To increase their effective capacities, they must develop efficient forms of representation, access, and operation. If forced to survive in a changing task environment, most inferential systems could benefit from *learning* about that environment and their own behavior. For example, they could exploit new schemata or different slot-names to simplify and restructure their knowledge. In this paper we describe how Eurisko (a discovery program) automatically extends its schematized representation of knowledge, by defining appropriate new slots dynamically.

1. Introduction

Computer programs, no less than biological creatures, must perform in an environment: an externally imposed set of demands, pressures, opportunities, and regularities. *Cognitive economy* is the degree to which a program is adapted to its environment, the extent to which its internal capabilities (structures and processes) accurately and efficiently reflect its environmental niche. If the environment changes frequently and radically, then the program can benefit from monitoring those changes and adapting itself to them dynamically. This is cost effective when the penalty for failure to adapt exceeds the sum of the following costs: (i) The continuous overhead for monitoring to detect changes, (ii) The onetime cost of providing a mechanism by which self-modification can occur, and (iii) The occasional expense of employing that mechanism to respond to a change which has been observed. This is expressed as Equation 1, below.

There are two evident directions for exploring this phenomenon:

(i) Make the preceding equation less mushy. Investigate what features of the performer (program), task environment, changes in that environment, etc., make the "monitoring & self-modification" process cost effective. Clearly, in some cases the penalty for non-adaptation is high enough; as examples spanning several time scales consider biological evolution (millenia), the educational system of a culture (years), the business meetings of a corporation (months), the immune system of an organism (hours), and the nervous system of an organism (milliseconds). In other cases, the above equation tips *against* adaptation as being cognitively economical; to date, almost all computer programs (and cognitive models in general) have been constructed to deal with a fixed task, or at least with a fixed task environment [Newell & Simon 1972]. But the magnitude of the phenomena we seek to model continues to grow, and we begin to feel the confines of any single, unchanging model we might hypothesize. In the years ahead, our models --- be they in Lisp, equations, or prose --- must become increasingly responsive. In the conclusions of this paper, we provide some further thoughts on this research direction.

(ii) Operationalize the vague phrase "monitoring & self-modification." In the case of computer models, how might they select (or *discover*) timely new knowledge, new control algorithms, new knowledge representations? What are the difficulties encountered?

This paper focuses on direction (ii). Section 2 assumes that programs can monitor their runtime environment, and it presents techniques by which they can modify themselves to heighten their fitness to it. Section 3 describes how one particular program, Eurisko, dynamically defines useful new slots automatically, at appropriate times. Appendix 1 discusses cognitive economy in existing programs; Appendix 2 presents some of the assumptions we are making in this paper; Appendix 3 illustrates some specific techniques for having programs dynamically monitor their task environment.

$$(1) \quad \begin{array}{ccccccc} \text{Penalty} & & \text{Overhead} & & \text{One-time cost} & & \text{Occasional cost} \\ \text{for} & & \text{in} & + & \text{of getting} & + & \text{of responding} \\ \text{not adapting} & > & \text{monitoring} & & \text{self-modification} & & \text{to observed changes} \end{array}$$

Equation 1. When it is cost-effective to have software that models itself, monitors its own execution, thereby notices changes in its runtime environment, and reacts to such changes by modifying itself.

2. Dynamic Self-Modification

Summary: We illustrate various ways in which a program might utilize knowledge gleaned dynamically: selecting (or perhaps even discovering) new data structures, representations, and algorithms, which seem better suited to the current runtime environment, the current user, the current problem, etc.

In order for a program to dynamically modify its own processing, it must examine, integrate and apply knowledge of the task domain, knowledge about programming in general, knowledge about the actual runtime environment, and knowledge gathered from observing its own processing behavior. Appendix 4 sketches ways in which dynamic program performance could be self-monitored. This section draws attention to the possibility of applying such usage data to modifying --- and in the extreme re-synthesizing --- the program.

Tasks can be specified anywhere from a pure "What" (desired input/output behavior) to a pure "How" (precise algorithms, data structures, representations, formats). It has been observed [Feigenbaum 1977] that one of the goals of AI research is to push our ability from the How end toward the What end of that spectrum. This often takes the form of designing a new veryⁿ⁺¹ high level language: i.e., higher than any that hitherto existed, one which takes over more of the program implementation. We can see this as far back as assemblers replacing machine language, and more recently in KRL [Bobrow and Winograd 1977], MRS [Genesereth 1981], LOOPS [Stefik *et al* 1982] and other extensions to Lisp.

Assuming, then, that the user desires merely to specify What, an intelligent language (automatic code synthesizer) must decide How. Upon what can it base its design? Previous efforts have employed knowledge about program transformations [Burstall and Darlington 1977] [Balzer *et al.* 1977], programming techniques and task-specific knowledge [Green and Barstow 1978], and --- if the input language is at all complex --- information to enable the successful translation of the input specification into a sufficiently precise specification of What [Lenat 1975]. Some efforts attempt to guide the decision-making during code synthesis by algorithmic analysis [Kant 1977] or even by aesthetics [Cohen 1979][Kahn 1979] [Lenat 1976]. But all these methods are *static*: they feed off a fixed base of initially-supplied knowledge (heuristics, constraints, suggestions, facts). The user supplies a specification for the desired program (via a formal spec, a dialogue, etc.), and the knowledge base is employed to produce that target.

What might it mean for a program to modify itself dynamically? The simplest kind of adaptive abilities along this line would be to *select* a data structure, a representation, or an algorithm from a set of well-known choices --- and to reserve the right and the ability to change that decision later. One could gather knowledge about the relative strengths and weaknesses of each choice [Kant 1977]. This knowledge could be in the form of rules which ask not only about static data, but which sample the runtime environment of the program as well. See Figure 1.

"If one conjunct appears to be false more often than any other,
Then make it the first one tested in the conjunction"

"If each person will use the program only once, briefly,
Then it is not worth building a model of each such user"

"If a function is used frequently and changed rarely,
Then it should be compiled."

"If items always appear to be accessed in the same order,
Then a sequential data structure is probably adequate for them."

Figure 1: Environment-driven selection rules

Such design questions could in most cases be answered initially (i.e., during program synthesis) but our suggestion of automating the gathering of such data is a further step toward the "What" of artificial intelligence.

There is another possibility, much more exotic than the previous one, which deserves attention: the automated discovery --- in "real time" --- of new types of data structures and representations which would be useful for the program in its current environment.

Defining a new data structure is not as difficult as it may first appear: a data structure can be thought of abstractly in terms of a set of operations one wants to be able to perform on it (e.g., First, Last, Next, and Previous, for lists). The set of desired operations need not be minimal nor concern itself with which operations are most important, etc. Such efficiency considerations are used, of course, in *selecting* an appropriate data structure, and in *implementing* it. The run-time usage of the currently-employed data structure can be sampled, and that can guide the search for an ever more appropriate implementation (e.g., in one usage, it might be very useful for Last to be efficient, and Interlist's TCONCing might be chosen as an implementation; in another environment, it might be useful for Next and Previous to be efficient, and a doubly-linked implementation would be best).

The two points here are (i) the implementation of a data structure may depend upon how it is commonly used in a particular spot in a program, by a particular user, and (ii) a new kind of data structure may be defined abstractly merely by selecting a set of operations; this choice, too, can be made by examining the run-time data as it accumulates (e.g., the final rule in Fig. 1.)

Defining a new *representation* is not quite so neat. In fact, humans have only managed to find a handful of representations to date (graphs, wffs, images, text, etc.) In lieu of any constructive suggestions along that line, we shall focus on *extending* a given representation: This appears to be intermediate in difficulty between selecting a representation and discovering a new one.

2.1. Extending a Schematized Representation

Summary: The Eurisko program extends its schematized representation by defining new types of slots. It uses a very simple grammar for defining new slots from old ones (legal moves), and a corpus of heuristic rules for guiding and constraining that process (plausible moves).

For a schematized representation, "extension" could occur by defining new types of slots. The Eurisko program (an extension of the AM program [Lenat 1976]; for more details see [Lenat 1983a,b]) has this capability, and we shall briefly describe how this mechanism was developed, step by step.

First, we isolated several ways in which new slots are defined from old ones:

STAR (e.g., Ancestor =_{df} Parent*, which means a Parent of a Parent of... of a Parent of me; the general case should be aParent);

UNION (e.g., Parent =_{df} Father $\cup$ Mother);

COMPOSITION (e.g., First-Cousin =_{df} ChildoSiblingoParent; i.e., any child of any sibling of either of my parents);

DIFFERENCE (e.g., Remote-ancestor =_{df} Ancestor - Parent).

These four operators (* , $\cup$, o , -) which define new types of slots are called *slot-combiners*. Other slot-combiners will be mentioned later.

Next, we added to Eurisko's knowledge base a concept for each type of slot, and a concept for each type of slot-combiner. Figure 2 shows some of the Generalizations concept (a slot) and the Star concept (a slot-combiner). Note in particular the way in which Generalizations is defined as Genl' --- i.e., immediate generalizations, their immediate generalizations, and so on.

GENERALIZATIONS

ISA: Slot
 SPEC: Extreme-Generalizations, Genl, Proper-Generalizations
 WORTH: 875
 AVG-SIZE: 7
 INVERSE: Specializations
 ANALOGS: Supersets, Relaxations, Enlargements
 SLOT-COMBINER: Star
 BUILT-FROM: Genl (i.e., Immediate-Generalizations)

STAR

ISA: Slot-combiner
 WORTH: 488
 ANALOGS: Repeat, Closure
 COMBINER-FOR: Generalizations, Specializations, Ancestor, ...
 SLOT-TYPE: Star
 DEFN: $\lambda (s) \text{ (SUBST } s \text{ for SLOT in } [\lambda (c) \text{ (CONS } c \text{ (self (GET } c \text{ slot))})])}$

Figure 2: Parts of the Generalizations and Star concepts

Finally, we modified the way in which slot entries are accessed. To illustrate this, we choose a simple task in mathematics, whose paraphrase is, "What are all the generalizations of the concept 'primes'?" The traditional way in which (GET 'PRIMES 'GENERALIZATIONS) would work is to examine the property list of PRIMES, looking for the attribute GENERALIZATIONS; if found, the entries listed there would be returned. If, as in Fig. 3, there is no such attribute, the call upon GET will return NIL (the empty list).

PRIMES

ISA: SetOfNumbers
 DEFN: $\lambda (n) \text{ (for } i \text{ from } 2 \text{ to } n-1 \text{ (NOT (DIVIDES } i \text{ } n)))}$
 GENL: Numbers
 SPECIALIZATIONS: OddPrimes, PalindromePrimes
 WORTH: 744
 EXS: (2 3 5 7 11 13 17 19 23 29 31)

Figure 3: Part of the Primes concept. Note that it has an attribute called GENL, which specifies its immediate generalization, namely Numbers, but it has no GENERALIZATIONS attribute.

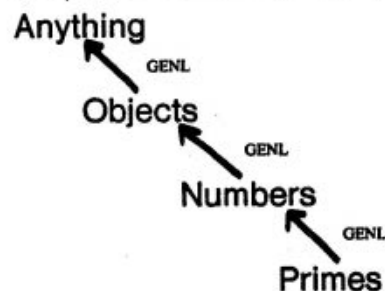


Figure 4: Network of concepts from Primes upwards, along Genl links.

We modified the way in which any retrieval request of the form (GET C F) operates. In case the F attribute of C has no entries (or doesn't exist), Eurisko examines the definition of F and --- if one exists --- tries to use it to compute the entries that could legally be stored on the F attribute of C. More precisely, before quitting we try to (GET F 'DEFN), and if that succeeds we apply it to C. Let's continue the example of (GET 'PRIMES 'GENERALIZATIONS). As we see in Fig. 3 above, there are no entries recorded on the GENERALIZATIONS slot of the frame for Primes. So GET now calls itself recursively: our original call is replaced by ((GET 'GENERALIZATIONS 'DEFN) 'PRIMES). But as Fig. 2 shows, there is no slot labelled DEFN on the concept for GENERALIZATIONS. So we recur one more time. By now our call has become

((GET 'DEFN 'DEFN) 'GENERALIZATIONS) 'PRIMES). I.e., get the expression stored in the Defn slot of the Defn concept; call it Δ . Δ should exist, and should be a lambda-expression (a function) which takes the name of a slot S as argument, and (when evaluated) computes a function which tells how to read slot S for any concept. Eurisko applies Δ to Generalizations, and that yields as a result a new function, a lambda-expression, call it Γ . Γ tells how to find generalizations of any concept. In particular, when Eurisko applies Γ to Primes, the result is a list of the legal entries for the Generalizations slot of Primes.

DEFN

ISA: Slot
 SPEC: Necessary-Defn,
 Suff-Defn,
 Operational-Defn
 WORTH: 975
 AVG-SIZE: 1.3
 ANALOGS: Statement, Algorithm
 DEFN: $\lambda (x)$ (APPLY*
 (GET x 'Slot-Combiner) 'Defn)
 (GET x 'Built-From))
 GENL: Predicate
 MyISA: EuriskoUnit
 MyCreator: Lenat

Figure 5: Part of the DEFN concept

Luckily, DEFN does have a DEFN slot, as we see from Figure 5, so we end the recursion (rather than entering an infinite loop at this point). Applying the entry stored there (in the DEFN slot of DEFN) to the argument "GENERALIZATIONS," we see our original call transformed into

```
(GET (GET 'GENERALIZATIONS 'SLOT-COMBINER) 'DEFN)
(GET 'GENERALIZATIONS 'BUILT-FROM))
PRIMES]
```

We see from Fig. 2 that the slot-combiner of Generalizations is "Star," and the argument (old slot) from which it is built is "Genl." So the entire call shrinks into

```
((GET 'STAR 'DEFN) 'GENL) 'PRIMES).
```

The Star concept has an entry for its Defn slot, as we see in Figure 2. This turns the preceding call into

```
((A (c) (AND c (CONS c (self (GET c 'GENL)))))) 'PRIMES).
```

In evaluating this expression, the first thing called for is

```
(GET 'PRIMES 'GENL).
```

which (see Figure 3) is just Numbers. That is, the immediate generalization of Primes is Numbers. The expression then calls itself recursively on Numbers, seeking their immediate generalization:

```
(GET 'NUMBERS 'GENL)
```

which we see from Figure 4 is simply Objects. The next recursive call asks for (GET 'OBJECTS 'GENL), which is ANYTHING. Finally, the next recursive call terminates when it is discovered that ANYTHING has no GENL (no immediate generalization.) See Fig. 4. The list constructed and finally returned is therefore (PRIMES NUMBERS OBJECTS ANYTHING). These four items are indeed the valid entries for the GENERALIZATIONS slot of PRIMES, according to the distributed definition of GENERALIZATIONS as GENL*.

Notationally there is no distinction between slots which are "primitive" (values actually stored as attributes on a property list) and slots which are "virtual" (values must be computed using the slot's definition). A heuristic might refer to the Generalizations of Primes without knowing, or caring, whether that initiated a single access or a dizzy chase. Thus we are decoupling, disentangling knowledge (the rules) from the specific implementation of its representation (the choice of which slots are primitive and which are virtual.) This is required because knowledge remains invariant under changes in the environment which effect the worth of various representation schemes. Donald Michie recognized this long ago with his memo functions.

To define a new kind of slot, now, Eurisko merely specifies one of the then-known slot-combiners and a list of the old pre-existing slots from which it is built. Thus a naive user sat down and defined a new slot, by creating a new concept (calling it "DG"), filling its Slot-combiner slot with the entry "Difference", and filling its "Built-from" slot with the arguments "Generalizations Genl." This thereby defined DG as a new kind of slot, one which returned all generalizations of a concept except its immediate generalizations; the call (GET 'PRIMES 'DG) then made sense, and returned the list (PRIMES OBJECTS ANYTHING).

After carrying out the above steps, to explicitly represent slots and slot-combiners in Eurisko, an interesting benefit accrued. With no extension at all, Eurisko was then able to define new kinds of slot-combiners. For instance, it proposed one which took two old slotnames as arguments, f and g, and defined a new slot which was f $\circ$ g $\circ$ f. This could be extremely useful (e.g., in database searches: see Fiksel and Bower [1976]). In particular the crucial slot "Examples" is defined as Spec $\circ$ ExsoSpec* (Exs are immediate examples, much as Genl's are immediate generalizations). The important slot-combiner PLUS (in the sense of regular expressions, not arithmetic) was also discovered by Eurisko. Eurisko defined it as the difference between f* and f. Later, Eurisko decided that Genl* was more worthwhile than Genl*, and (partly for this reason) PLUS itself ultimately achieved a higher Worth than STAR.

2.2. CACHING to reclaim lost efficiency

Summary: "Caching" the results of computations can dramatically improve the performance of many programs. Reasoning can be brought to bear to decide whether to cache, if so what to remember, and (later) whether or not to ignore the cached value and recompute it. Caching reclaims the efficiency that could otherwise be sacrificed by the highly indirect representation-interpreting scheme described in the previous section.

The reader may object that Eurisko's scheme for indirectly interpreting the definition of each slot is a horribly inefficient way to operate, since the slots' definitions change so rarely. It seems wasteful to squander cpu time recomputing (GET 'PRIMES 'GENERALIZATIONS) each time, and it seems absurd to

recompute (GET 'GENERALIZATIONS' DEFN) each time. After at most one or two repetitions, we would want our program to have "learned" an efficient way to get an answer.

Indeed, a general tool for cognitive economy --- software caching --- can be exploited to reclaim most of this efficiency. By a "general" tool, we mean one which is applicable in almost all environments, not merely those being focused upon in this paper (fluid ones.)

Immediately upon computing (GET c f), Eurisko stores the computed value v in the f slot of concept c (a la Michie's memo functions.) Thus, the next time (GET c f) is called, it will find v stored right there, and return with it immediately. After the first time (GET 'PRIMES' GENERALIZATIONS) was called, the list (PRIMES NUMBERS OBJECTS ANYTHING) was stored on the GENERALIZATIONS slot of PRIMES. Recall that Eurisko had to recursively call (GET 'GENERALIZATIONS' DEFN), so the λ -expression that that call returned would be stored on the DEFN slot of the concept called GENERALIZATIONS. When we later call (GET 'DUCK' GENERALIZATIONS), that call will run much faster, since the definition of the Generalizations slot has already been computed and stored away in the "proper place." The definitions of slots are very slowly --- if ever --- changing, hence the recomputation of Defn of Generalizations is quite a rare event. Caching that value *must* be cost-effective in the long run.

In general, we see that caching a value for slot f of concept c is most applicable when the value can be expected to vary quite infrequently. In Eurisko, this gives us the flexibility to redefine a slot's definition if we wish, but (since this change of representation will be rare) the program will run just about as efficiently as if that capability were absent. This is analogous to compiling: so long as the definition of a function doesn't change too often, it's undisputedly worthwhile to compile it. The caching of values for frames' DEFN and GENERALIZATION slots is not in any way special; the value of any virtual slot can be cached.

We saw earlier that notationally there was no need to distinguish an access of a primitively-stored slot from an access of a virtual, computed slot. For instance, a heuristic could refer to "Generalizations of Primes" without knowing or caring whether that initiated a single access or a long rippling search. Now we see that once the value is computed and cached away, there may be no telling it from a primitive one. Thus, with but a small one-time cost, our program runs as quickly as if all slots were primitive. The inefficiency introduced by the interpretive slot-combiner scheme appears to have been eliminated.

Of course, the values that Eurisko caches away may become stale and incorrect after a while. When doing a GET, therefore, when a cached value is encountered, Eurisko must decide whether to accept it, or else to ignore it, recompute it, and store the new value there. One possible answer to this "updating problem" is the following conservative one: whenever a value for a slot of a concept changes, discard any cached values that might *possibly* have depended on it. Arbitrary amounts of reasoning may be brought to bear to decide which values needn't be discarded to maintain consistency of the knowledge base, but of course if too much time is spent in this pursuit the efficiency advantages of caching disappear. Eurisko carries on AM's tradition of living with contradictions, with a plausible but not "correct" knowledge base. Namely, it has some heuristics (see Fig. 7) for deciding whether to ignore or accept any cache it encounters during a GET operation.

We added some extra arguments to GET, parameters which describe a cost/benefit curve: for each amount of resources expended and each degree of precision of the result, this curve specifies precisely how acceptable that resource consumption / accuracy of solution pair is. One extreme of this is to provide an ironclad limit on resources and say "Do the best you can within these time/space/... limits." Another extreme (most common in present-day programs) is to specify an ironclad goal criterion ("Find an answer which is at least *this* good, no matter how long it takes to get.") We are calling attention to the space in between the two extremes.

To bring us back from the lofty heights of generality, let's see in more detail how we got Eurisko to do this. First, we linearized this space: we picked a few significant parameters of it, and defined a description to be merely a list of values for these parameters. When a call on GET was executed, and a cached value was encountered, a formula relating these extra arguments to GET would then determine whether to accept that cache, or to ignore it and recompute the value. Eurisko's parameters (extra arguments to GET) are: cpu time to expend, space to expend, whether the user can be asked about this, how fresh the cache must be, minimum amount of resources which must have been expended at the time the cache was written.

So (GET PRIMES GENERALIZATIONS 200 40 NIL 3 0) would allow any cached value to be accepted if it appeared that recomputing would take more than 200 milliseconds, or would use up more than 40 list cells, or if the value had been put there less than three Cycles ago. (In Eurisko, a "Cycle" is the time it takes to work on the top-rated task on the top-rated agenda.) Otherwise, the cache would be ignored, a newer value would be computed, and it would be stored away. With it would also be recorded the following information: (i) the fact that it was a cached value, not a primitive one. (ii) how long it took to compute, (iii) how many list cells it used up in computing this value, (iv) the current Cycle (number of tasks worked on so far). The above call on GET might result in the following value being stored on the GENERALIZATIONS slot of PRIMES:
(*cache* (PRIMES NUMBERS OBJECTS ANYTHING) 54 6 9).

2.2.1. Storing and Updating Cached Values

The caching process involves storage and updating. For both of these aspects, we can discuss details of when, why, how, and what. The decisions that arise are made with the guidance of heuristic rules, many of which are illustrated below in Figs. 6 and 7. Those lists have omitted many of the very general "common-sense" heuristics, and those which deal with details of Lisp programming. Additionally, they present the rules in English, and freely use informal expressions such as "recently"; in the actual Interlisp encoding they are of necessity operational, hence more precise (and concomitantly a bit less valid).

Just as each heuristic is not always relevant, so to the corpus applies only to some (although we hope many) AI programs, running on large uni- (or few-) processing computers, running in an environment such as Interlisp's, and with a human user watching and interacting. There would be other rules for other machine architectures and languages and man-machine interfaces. In other words, one could say that we have only partially specified the IF-part (left hand sides) of the following heuristic rules. An alternative would be to build up an entire knowledge base of heuristics just to determine the applicability (the domains of relevance) of the following (as a function of machine and language.)

When to cache:

Every time you have a call to a lower order function.
You called a function and it took a long time to return.
The value returned is the same as last time.
Some function F is called with the same value x often.
The amount of time to recompute the value would be high.
The cached values can be used as data to induce upon.
Time is dearer than space.
The value is not so critical you'd always favor recomputing.
This number is high: $\frac{(\text{freq. of usage}) \times (\text{avg. cost to recompute})}{(\text{freq. of change})}$

Where:

There should be a canonical storage place to facilitate lookup.
Instead of an asstetional database (with no structure) we prefer to add structure: namely, slots (including domain-specific ones)

How:

Given a request to compute $F(x)$, point to the canonical place(s) where cached value might be; if it's there, return it; if it's not there, compute it and store it there for next time.
 Caller may constrain you
 Discarding any cache results only in slowdowns, not errors.

What:

Store the returned value
 Store a copy of the call that got this value
 Store important intermediate subgoals worked on to get the value
 Store a description of what situation would make this value stale
 If no cache is stored, store a footnote explaining why

Figure 6: PRINCIPLES FOR STORING CACHES

When:

The current request would have produced a different answer.
Usually: ...would have produced a better answer; e.g.:
 The current request specifies more resources can be brought to bear.
Most frequently: There is no cached answer there, yet.
 Either x or $\text{Defn}(F)$ has changed since $F(x)$ was last cached.

How:

Choose: discard old value, or keep it around to induce or back up.
 Use demons to flag that caches are out of date.
 Just because a cache is stale, don't recompute it (just erase it).
 Keep track of dependencies of one cached value upon another.
 This at least enables timely discarding; at best, intelligent updating.

Figure 7: PRINCIPLES FOR UPDATING CACHES

2.3. The Spectrum of Data Reductions

Caching, as we have described it, is merely the lowest-level of a tier of data reduction processes, techniques for compiling hindsight into usable, more efficient, more procedural forms.

We often refer differently to "caching" depending upon what it is that's being retained: open-ended, inductive searches can be condensed in hindsight (i.e. cached) into heuristics, deductive searches can be cached into much less branchy algorithms, subroutines can be cached into tables of frequently called arguments and resultant values, and variable quantities have their value cached simply by the process of binding. To see this, consider the progression of ever more sophisticated tasks which programs perform [see Fig. 8]: accessing a datum, calculating a value from a

fast algorithm, deducing a result by searching for a solution or proof in a well-defined space, inducing a result by a very open-ended search process. Often there is the opportunity to take experiences at one of these levels of sophistication and abstract them into an entity which is one level lower, yet which appears to contain almost the same power.

Consider first the way in which we reduce induction. After much experience with inducing, we produce a heuristic rule~ which enables the same inferencing to be done in a much more controlled space, in a much more deductive way. Heuristics reduce discovery tasks to mere (!) symbolic manipulation within a well-defined system (as in AM [Lenat 1976]).

Deduction is reduced similarly. By many painful deductions, we gain knowledge of the space we are searching, e.g. by deriving many theorems about it. Those theorems can then be used to vastly simplify our search. Ultimately, the process becomes efficient calculation. A short, optimal algorithm may be found, e.g., which replaces a search for a solution (this has recently occurred with the "n queens" chess problem.)

Now we see that our value-caching process is the next step in this progression. It reduces a calculation (e.g., to traverse a particular access path to gather all Ancestors of a node) by a lookup, by a single access of a datum in memory.

All the data reduction processes in the progression are environment-independent techniques for increasing cognitive economy. Often, an apparent loss due to a specific expressive feature can be rectified by applying some of these kinds of methods. In our case, we opted for interpretive defining of slots, but regained efficiency by caching those slot definitions.

2.4. Contrasts with Psychological Ideas of Economy

When semantic networks first appeared in cognitive psychology, they led to the idea that function should follow form. In particular, since taxonomic knowledge structures were hierarchical, inference processes would presumably follow the same inter-category network paths each time the corresponding relation needed testing. It was for this reason that Collins & Quillian [1969][1972] conjectured that the time required to verify semantic relations would correlate with distance in a corresponding "economical" semantic network. Thus, they showed that people ordinarily verified "a canary is a bird" faster than they could verify either of the two more remote relationships, "a canary is an animal" or "a canary has wings."

Subsequent research, however, showed that the needs of particular retrieval experiences, not *a priori* categorizations, determined which inference paths humans would follow and the associated response times. A simple counterexample to the presumed hierarchical retrieval schemes was found by Rips, Shoben and Smith [1973]. They showed that people access a familiar but so-called "indirect" relation such as "a dog is an animal" much faster than the supposedly immediate but less frequent relation "a

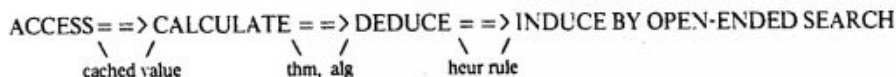


Figure 8: Progressively more sophisticated caching: of values, of algorithms, of heuristics.

dog is a mammal." Using experimental materials and methods, Hayes-Roth & Hayes-Roth [1975] found that human memory seemed plastic and adaptive in many of the same ways we have proposed for intelligent machine programs. Their results led them to "an adaptive model of memory, in which all learned relations are represented directly, with strength proportional to experience. There is also good evidence for redundancy in the network, with multiple routes connecting nodes representing particular pairs of concepts" [1975, p. 519].

In another study, Hayes-Roth and Hayes-Roth (1977) found some evidence that people understand and remember English texts directly in terms of high-level lexical concepts, rather than exclusively as low-level or primitive concepts. They argued that cognitive processes would presumably benefit if they could build directly upon the words or other meaningful units of familiar language. Any system that interpreted and stored data only in terms of primitive units, as is often assumed in language understanding theories, would encounter many slower and costlier searches. Both the empirical evidence and theoretical principles support the idea that human language understanding exploits many kinds of caching.

In short, we have chosen the term "caching" to refer to these types of redundant storage. Caching, to capture and exploit repetition and regularity, will benefit both humans and machines of limited processing capabilities.

3. When to Define a New Kind of Slot

Summary: We have discussed how a new slot might be defined; we consider now how the Eurisko program does decide when and how to define a new kind of slot.

Our basic reply to *When?* is that a slot is defined only when a heuristic rule explicitly suggest it as being useful; that is, heuristics are used as plausible move generators (as opposed to their typical use [Nilsson 1972] as implausible move pruners). Our basic reply to *How?* is that the heuristic which suggests a new slot must provide (constraints for) its definition.

This merely pushes the problem back one layer, namely to the way in which the heuristics know when and how to define new slots. But now we are talking the language of AM and Eurisko, and we can provide examples and general characterizations of the slot-defining heuristics used in Eurisko. Since each type of slot is represented internally as a full-fledged (schematized) concept, the new slots defined can eventually be evaluated (and changed or discarded if they are not utile). Each heuristic is also represented as a full-fledged concept, and its worth can be determined based on the worth of the slots it suggests; thus each slot-suggesting heuristic is occasionally modified (or even discarded). Let's look at some cases:

Sometimes the new type of slot is first proposed purely for aesthetic, analogy-completing, or exploratory reasons. One such heuristic is the following (note that "I" is the identity mapping):

"If R and S are partial inverses of each other, Then it's plausible to define and investigate the following: RoS, SoR, RoS - I, SoR - I"

There are several noneffective slots provided for the above heuristic in Eurisko, such as its reason (in this case, "aesthetic"), its average running time (7 seconds), which other rule(s) created it originally (IsolateDifferences), instances of its use, pointers to slightly more general and more specific heuristic rules, etc.

Note that the heuristic applies when R and S are mathematical functions, heuristics, or slot definitions. Let's examine an example of the latter case, which is of concern to us. This occurred when R was the slot "Generalizations" and S was the slot "Specializations." The heuristic rule concluded that it was aesthetic to define the specializations of the generalizations of a concept (what we would

call "siblings"), and the generalizations of the specializations of a concept (what we would call "inlaws"). Eventually, the Siblings slot is seen to be much more useful than the Inlaws, which is discarded.

The new kind of slot-combiner defined in the previous section [$\lambda(f.g) \text{ fogof}$], was derived purely from symmetry (using a rule similar to this one), and then it was noticed that some of the slots which are most useful in the system (e.g., Isas, Examples) share that common form.

There are other ways in which new types of slots are defined by Eurisko. The definition of a new slot is often based soundly upon need — or the absence of need. For example,

"If many concepts have a large number of entries for their F slot, Then consider creating some new specializations of F."

To make this rule effective (executable), Eurisko monitors the sizes of slots on its units. When applied, the rule leads Eurisko to create several new slots, each having fewer entries on the average than F had, to cover the original F slot. E.g., after noting that the Examples slot was heavily used, it would be plausible to consider creating new slots like Extreme-examples and Typical-examples. An actual case where this rule was used was in the design of three-dimensional VLSI circuits: many units (representing chips) had enormous entries for their Terminals slot; Eurisko fired this rule and produced several new, more specialized slots; one useful such one was XOR-InputTerminals (its value is defined to be a list of sets of terminals, each set having the property that precisely one element can be an input terminal of the device).

A slightly more specialized heuristic says:

"If you frequently perform 'Remove x from f(x)', Then it's plausible to define g as $f - I$ "

Here, I is the identity, so $g(x)$ is simply $f(x)$ with x removed. This led to the definition of the proper ancestor slot ($\text{Genl}(c) \rightarrow c$) from the simpler Genl (immediate generalizations) slot. It also led Eurisko to define the new slot-combiner PLUS from STAR. We envision how it might lead to the definition of many common, useful mathematical functions, including (although Eurisko doesn't actually do this) operations on matrices with main diagonals removed.

The above examples have tried to illustrate the manner in which heuristic rules can generate timely new kinds of slots. A very important point is that the rules are not specific to defining slots; they apply equally well to defining new heuristics, new mathematical functions, new lab procedures, and so on. One of the bedrock philosophies of the Eurisko effort is that a single large core of commonsense knowledge can adequately guide discovery processes operating on all these levels.

4. Conclusions

If forced to survive in a changing world, most inferential systems could benefit from *learning* about their task environment and their own behavior. For example, they can exploit new schemata or different slot-names to simplify and restructure their knowledge. Self-modification would be *cognitively economical*.

If forced to explore large search spaces with some repetitive regularity, a cognitive system could benefit by employing *caching* to save partial results. We described a variety of techniques to implement caching, and explained how caching specific results is one of a spectrum of methods (algorithms, heuristics) that can make trade-offs among precision, speed, certainty, and generality.

Since most environments contain much that is extraneous to the direct solution of a task, a problem solver in such worlds can gain cognitive economy by *expectation-filtering* (see Appendix 4.1.) In

general, intelligent systems need to exploit their knowledge about *typicality* to reduce their cognitive load and increase their attention to important data. In many situations, we believe that expectations can both reduce processing requirements for handling ordinary events as well as simplify the identification of surprising events.

If the tasks are complex, and the resources allotted vary dramatically in magnitude, then *multiple levels of abstraction* (MLA) becomes an *economical* representation (see Appendix 4.2.) In many simple environments, the advantages of MLA are minimal and may not even justify their costs. Thus, routine tasks may warrant the kind of knowledge compilation that would convert an initial, expressive MLA knowledge-base into a fast, integrated, and largely uninterpretable code. Conversely, as task complexity and variability increase, MLA increasingly provides a basis for intelligent and rapid restructuring.

4.1. The Need for Further Research

In the years to come, AI programs will employ greatly expanded knowledge bases and, as a consequence, they will explore increasingly open-ended problem spaces. Already, a few existing systems show signs of having more potentially interesting things to do than they have resources to pursue. In the past decades of intelligent systems research and development, several design concepts have emerged in response to contemporary needs for creating ever larger knowledge bases. For example, many researchers proposed multiple levels of abstraction and automatic property inheritance as keystones of efficiency or "cognitive economy." We believe that the value of such mechanisms derives largely from their usefulness in describing initial knowledge bases. Once an intelligent system begins to explore the consequences of its knowledge and to solve novel problems in a dynamic environment, it needs to adapt its knowledge to achieve faster and more profitable retrievals.

A theory of cognitive economy should explain under what conditions (and why) knowledge needs to be adapted, and should prescribe systematically how to do it. In this paper, focusing upon fluid environments, we have tried to motivate and lay the groundwork for such a theory. We presented Eurisko's automatic defining of plausible, useful new slot types. Many of the design heuristics we mentioned embody some gross features of an eventual theory and suggest numerous paths for future research.

We have tried to show what cognitive economy *is* and *is not*. It *does not* consist of a set of static knowledge-base design principles, such as those proposing taxonomic concept structures with automatic property inheritance. Rather, cognitive economy is a feature of those intelligent systems that learn to solve problems efficiently and consequently realize more of their lifetime potential. As knowledge bases expand and basic software obstacles are overcome, AI systems will increasingly address the same question facing intelligent humans: "What would I most like to accomplish next, and how can I do that economically?" As has always held true with human beings, the answer for AI systems increasingly involves some type of learning.

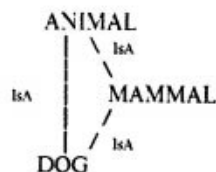


Figure 9. Although the IsA link from Dog to Animal is redundant, it is worth having.

Appendix 1. Cognitive Econ. in Existing Programs

To date, most programs, including our own AI behemoths, have *not* had to cope with runtime environments that changed significantly. Referring back to Equation 1 (in Section 1), the costs for monitoring and building in self-modification abilities have been prohibitive, and the need for adaptiveness has been minimal. The environments have been *complex* but not *changing*. To be "fit" in such worlds, programs have had to employ sophisticated architectures, knowledge bases, control strategies, representations, and expressive languages in which they are specified; but programs have *not* needed to reorganize themselves dynamically, to adaptively alter those sophisticated design components. Even in those situations in which this flexibility must be required in the long run, it has been avoided, because (i) self-modification capabilities are tough to build in: many programs have simple self-reprogramming as their sole task, and (ii) monitoring is an expensive overhead in both space (detailed record-keeping) and time (finding regularities in such histories). This raises the issue of efficiency:

Efficiency

When we build a new, large AI program, we often find ourselves caught in the following tradeoff: On the one hand, we want to develop our initial systems quickly and painlessly, since they are experimental and ephemeral. We want to be able to modify them later with little or no redesign or reprogramming manually. On the other hand, we want our systems to run as efficiently as possible, to minimize the demands for cpu time and primary and secondary memory. We are torn between *expressibility* and *efficiency*.

Many AI researchers *cum* language designers have focused on *expressibility*, the problem of rapidly constructing a working experimental vehicle. This objective is apparent in the goals of EMYCIN [Feigenbaum 1977], KLONE [Brachman 1978], KRL [Bobrow and Winograd 1977], PSI [Green 1977], RITA [Anderson and Gillogly 1976], ROSIE [Waterman et al. 1979], and SAFE [Balzer et al. 1977]. The results are some superlative software (such as the systems just mentioned), and also exemplified by Interlisp's DWIM, File, and Break packages. Four fundamental techniques utilized in highly *expressive* programs are: (i) reliance upon a very-high-level language, (ii) planning and reasoning at multiple levels of abstraction, (iii) inference by pattern-directed knowledge sources, and (iv) minimal, nonredundant representation, as in a canonical generalization/specialization hierarchy.

Both within the body of this paper (Caching, Section 2.2) and within these appendices (Expectation-Filtering and Levels of Abstraction, Appen. 4.1 and 4.2) we present various techniques which do not sacrifice expressibility, yet enable programs to (semi-)automatically improve themselves and thus increase their productivity. One point of this paper is to show that the second goal of AI programming, short-term runtime efficiency, is not sacrificed by these techniques.

The basic source of power is the ability to predict the way that the program will be used in the future, and to tailor the program to expedite such uses. This approach is sometimes used *manually*, as when a program is quickly coded in an expressive but inefficient form, used heavily, and then recoded in a much more optimized form. Thus, e.g., Ray Carhart spent 1977 rewriting DENDRAL (streamlining its CONGEN module) for a minicomputer.

The traditional approach to program optimization has assumed that static analysis of programmer's expectations (whether implicit in his code or explicit in his formal assertions) can identify the significant optimization opportunities. Three types of methods for analyzing program descriptions in this way include: (i) analyzing program flow and structure [Knuth 1968] [Dijkstra 1976], (ii) designing data structures to be appropriate, and (iii) compiling (as in the FORTRAN H compiler) and optimizing transformations of the program (as in [Darlington and Burstall 1973 1977] or [Balzer et al. 1977].)

In a changeable environment, we advocate the use of methods more *dynamic* than these. Rather than improving the static description of a program, we propose that the program modify its own structure to adapt it to its operational environment. Some earlier automatic programming systems [Darlington and Burstall 1973] [Lenat 1975] [Low 1974] [Kant 1977] [Barstow 1977] were designed to improve programs' efficiency, and many of their ideas have found their way into the techniques we now describe. Those systems had program construction, transformation, and optimization as their primary task; we are proposing general methods to provide self-improving abilities to other kinds of AI programs (understanders, theory formers, expert simulators, planners, etc.).

One valuable source of prediction comes from the program's experiences as it runs. If a program can "learn" from its experiences, it might try applying each piece of newly acquired knowledge to the task of specializing itself, modifying itself to run more efficiently in the current runtime environment. We believe that a program's "intelligence" can be increased in this way; that is, by increasing its ability to acquire appropriate knowledge, to organize that knowledge, and to refine the conditions under which that knowledge is recognized to be applicable. For any fixed quantity of manpower resources we expend, there is a limit to the size/complexity of programs which can be successfully implemented. This barrier might be overcome by programs which are self-extending, which actively do things to enlarge their input domain, their capabilities.

Notice that representing a corpus of knowledge as a minimal (canonical) generalization/specialization hierarchy, with interpretatively-computed inheritance, may or may not be cognitively economical: this technique favors expressibility, compaction of representation, and efficiency in updating and altering the knowledge base, but at the expense of performance. It is true that a dog is a mammal, and a mammal is an animal, and from those two we could compute that a dog is an animal (see Fig. 9), but if the same arc is being repeatedly recomputed then it is more cognitively economical to store one redundant arc. Studies by ourselves and others [Hayes-Roth and Hayes-Roth 1974] [Rips et al. 1975] indicate just such redundancies being created and strengthened by repetitions.

Obviously, the economy of specific representations and related inference methods depends heavily on underlying machine architectures and costs. We assume that intelligent systems aim chiefly to produce useful results (e.g., novel hypotheses) as rapidly as possible. In short, they should search cleverly and fast. Thus, *a priori* notions of aesthetics or design economy have little bearing on cognitive economy. Massive redundancies in storage, processors, and inference processes seem more directly relevant. Furthermore, we suppose that, like people, intelligent software should eliminate slow, interpretive searches for previously solved problems (especially if the same problems are repeated often). To this end, cognitive economy may be achieved through dynamic compilation of action sequences. Such compiling would reduce the time-cost to effect the corresponding behavior; on the other hand, this efficiency would appear to be gained only at the cost of interpretability, interruptibility, and modifiability. Those are desirable attributes iff a program potentially needs continual changing. In typical situations, both efficiency of compiled forms and accessibility to declarative forms are intermittently needed. These different needs point again to the economic benefits of maintaining simultaneously alternative and redundant representations.

When It Pays to Adapt

We can abstractly characterize the situations in which an AI program would find it cognitively economical to monitor and adapt itself: the resources available for specific tasks vary significantly; or the specific tasks have (at least locally) some similarities, some structure which enables them to be solved quickly in the same particular way. Here are some examples:

The load average of a time-sharing system fluctuates with time of day, but users demand that their speech understanding system always provides the best guess it can in real time.

A hospital simulation program can run at excruciating levels of detail, and it lives on a dedicated machine. But there's been a bad accident nearby and within one minute the police must have an estimate of how many emergency cases the hospital can accept.

A program is given an enormous amount of resources to prepare for each task, but little resources between the time a task is presented and an answer is called for. It may be cost effective for the program to spend some of its preparation (inter-task) time trying to discover new algorithms, representations, etc. We are struck by this in nature (evolution), though as yet just glimpse it in AI programs (such as Eurisko).

A system is sold to several customers, each with a distinct but complex task environment in which it will live. It then becomes cost effective to spend some resources streamlining the program (either manually or automatically) to fit that user. In terms of equation 1, the environment changes only once, so no continuous monitoring and modification is performed; the onetime cost of the modification can be amortized throughout the lifetime of the program (i.e., the penalty for non-specialization is integrated over time.) The well-known SYSGEN generation of tailored operating systems is a simple example of this.

While maintaining a file system, the program notices the patterns in each user's accesses to his files, and of the community as a whole, and occasionally selects a new data structure for some part of the system to make it run more efficiently. See [Low 1976].

Appendix 2. The Assumptions

Summary: The claims we make elsewhere in this paper presuppose that "intelligence" can be adequately modelled as heuristic search, guided by pattern-directed rules, implementable on a uniprocessor.

Every scientific theory is constructed in a rich context of surrounding theories, methods, and standards determining which experiments are reasonable ones to perform and which point of view to take when interpreting the results -- in short, a *paradigm*. We feel it useful to articulate the "core" of one's paradigm: the assumptions one's theory rests upon. For that reason, this appendix is devoted to a presentation of our assumptions, including: a model for intelligence (Appendix 2.1), a model for how intelligent computer programs may be organized (Appendix 2.2), and a model of the characteristics of present computing engines (Appendix 2.3). Briefly:

(i) We continually face searches in more or less immense spaces; intelligence is the ability to bring *appropriate* knowledge to bear, to speed up such searching. Increasing intelligence then comprises increasing knowledge, improving its organization, and refining the conditions for its applicability. How are these new discoveries made? Empirical induction (generalizing from experimental and other observations), analogy, and the criticism of existing theory all lead to new conjectures, new possibilities.

(ii) Intelligent systems can make the applicability of their knowledge explicit by representing that knowledge as condition-action rules *IF situation, THEN appropriate reaction*. Such pattern-directed inference systems (PDIS) can benefit from a schema representation (frame, unit, being, script, etc.), because this adds structure which the rule descriptions can exploit.

(iii) Current computing technology presents us with limited cycles, cheap storage, and expensive knowledge acquisition.

Appendix 2.1.: A Model of Intelligence

Summary: Since "intelligence" depends upon bringing relevant knowledge to bear, it can be increased not only by adding new knowledge, but also by better organizing the existing knowledge.

Many human cognitive activities, including most of those commonly referred to as "requiring intelligence," can be cast as searches, as explorations through a search space, meandering toward some goal. We call a problem-solver "more intelligent" if it can work efficiently toward a solution even in the face of an immense search space and an ill-defined goal. Somehow, it is imposing more structure on the problem, and using that to search more effectively. How might it do this? According to our model of human intelligence, it accomplishes the task by bringing knowledge to bear, knowledge not supplied explicitly in the problem statement. This knowledge can be about problem-solving in general (e.g., how to recognize and break cultural blocks [Adams 1974]) or about the problem domain specifically (e.g., which groups of chemical atoms can usually be treated as aggregate superatoms [Feigenbaum 1977]).

This implies that a problem solver can become more effective by (i) increasing its knowledge, (ii) better organizing its knowledge, and (iii) refining its criteria for deciding when various pieces of knowledge are applicable. In terms of production (If-Then) rules, these correspond to adding new rules, modifying the data structure in which rules are held, and tuning the conditions (If parts) of existing rules.

One very basic assumption we are making is that of *continuity*: the problem-solving environment does not change its character too radically, too rapidly. That is, our environments are *fluid* but not gaseous. Let's see why this assumption is required. If the program abstracts from its experiences a plausible heuristic *H*, it may be able to analyze past scenarios to verify that possessing and obeying *H* would definitely have improved its performance (e.g., led to the current state of knowledge quicker); but its only justification for believing that *H* will help it in the future is the empirical evidence for the continuity of the environment. Learning can be translated effectively into action only when the structure of the environment has not altered too greatly.

A related assumption is that of *irreversible knowledge accretion*. Our body of facts and observations continually grows and only occasionally shrinks. Certainly the abandonment of a whole system of thought is a rare event indeed (see [Kuhn 1972]), whereas the adoption of new systems of thought is part of the standard educational process.

New knowledge is discovered by specializing some broadly applicable but weak principles, or by generalizing particular experiences. The latter includes abstraction (condense some experiences into a heuristic which would have greatly aided us in the past if only we'd had it) and analogy (draw parallels to other existing facts and heuristics, and also to the *paths* which led to their discovery.) One sometimes "spirals in" [Lakatos 1976] toward precision and competence (see Fig. 10).

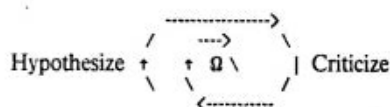


Figure 10. The Lakatosian loop of criticism and improvement of hypotheses, spiralling ever closer to truth and understanding.

Appendix 2.2: Model of Intelligent Program Organization

Summary: Since we want our AI programs to access, reason about, and expand their own knowledge bases, it is useful to represent such knowledge in a clean, modular form (e.g., employing any one of the current schematized representations.)

The preceding remarks about intelligent problem solving apply equally to "hardware" and "wetware" alike. To be intelligent, computer programs must ultimately grapple with the tasks of knowledge acquisition, representation, and refinement. We cannot provide an absolute answer to how they should be organized, but we can posit some design constraints which have proven useful so far.

A very general heuristic in AI programming is the following:

If your program is going to modify its own *X*,
Then make *X* as separable, clean, and explicit as possible.

In our case, *X* can be instantiated as "knowledge," or as "applicability conditions for each piece of knowledge." In this case, the heuristic advises us to represent our knowledge in a separate, clean, explicit form, say as knowledge modules having some fixed internal structure, and also advises us to keep the applicability conditions for each knowledge module separate from the rest of the knowledge it contains.

This naturally leads us to a pattern-directed inference system, in which knowledge is broken into separate modules, each with an explicit set of relevancy tests [Waterman and Hayes-Roth 1978]. Such systems arising in Pittsburgh and London may champion syntactic purity, while those arising in California may tend toward the baroque, but such variations should not obscure their common source of power. The PDIS architect breaks his program's knowledge into a set of condition-action production rules.

Having a clean representation for rules means having a clean, precise, elegant language in which to express them. By structuring the conceptual knowledge of the system, by partitioning each module's knowledge into several categories, a rule can condense an entire cumbersome description into a simple reference (often a single pointer). The popular schematized representations of knowledge (scripts for episodes, frames for static situations, beings for specialists, units for everything) enable a rule like "If there are no known methods specific to finding new examples of prime numbers, then..." to have its condition coded as a simple null-test on the "To-get" subplot of the "Examples" slot of the schema called "Prime Numbers." By a judicious choice of slot types and subplot types, the system builder can reduce most triggering conditions to such quick checks on the state of various (subplots of) slots of schemata.

Additional knowledge can be brought to bear to locate efficiently all the rules which *might* have their conditions satisfied in a given situation, and also to decide which rules to execute (obey) among those whose IF- parts have triggered (been satisfied).

Appendix 2.3: A Model of (Present) Computers

Summary: Since we are considering the problem of building computer models of intelligent behavior, many of our assumptions deal with the characteristics of present-day computers. They are the symbol manipulators we use, but were not designed for that general purpose.

The foremost quality of computing machines, as we know them today, is their limited "inferential bandwidth." Although they provide virtually (*sic*) limitless storage capacity, these machines can process only a small fraction of this information in a reasonable amount of time. Two primary reasons for this disparity between storage and generation capacities include: (1) knowledge acquisition proceeds slowly and expensively; and (2) inference processes depend on operations that ordinarily require cycles of uniprocessor systems.

The first problem places a cultural limitation on the development of intelligent machines. This development is restrained by our inability to express what we would like our machines to do or to convert these expressions into effective machine interpretable codes.

The second problem reflects a technological constriction of our potential. The so-called von Neumann bottleneck, caused by machine architectures that sequentially interpret program steps through a single logical processor, strongly motivates concern for programs that can minimize unnecessary computations.

In some future environment, perhaps neither of these problems will persist. In the meantime, however, we assume that two of the best ways to increase AI system productivity, derived from these current limitations, will be to expedite knowledge acquisition and to avoid unnecessary computing (for instance, by opting to "store & recall" rather than "forget & recompute").

Appendix 3: Monitoring the Task Environment

Summary: The body of this paper has amply illustrated how a program might profit from knowledge it gathered dynamically. This suggests that rather than working on the given problem exclusively, a program may be better off to expend some time learning: learning about the specific problem, its broader domain, or problem-solving in general. Note this suggestion would encompass traditional education (being told), empirical research (observing), and theoretical exploration (predicting). While such "very high-level" problem-solving strategies typify human problem-solvers (especially those of us who have rationalized spending twenty or more years "in school"), very few programs to date have employed any of these forms of learning to improve their operation.

We discussed in Section 1 the "environmental fluidity" conditions under which it could be cost effective for an intelligent system to be able to improve its performance dynamically. To accomplish this, a program must first be able to monitor its processing, i.e., to sense, record, and analyze its representations, data structures, algorithms, inference procedures, and the way they are currently being relied upon. It can then use this gathered knowledge to modify or redesign itself to perform more efficiently.

Some programs (and wish-lists for program capabilities) have included learning more about the problem being worked on. For instance, McCarthy's advice-taker and Balzer's dialogue system (to get airline reservation programs written) would be told, Gelemter's theorem-prover (employing diagrams as analogic models) could observe, Lenat's AM could observe and predict. In this paper, however, we are stressing learning about the ways the program is being currently employed.

Learning by Being Told

There are many situations in which a user could provide advice to a program:

(a) The user might want to convey to the program some task-specific item: "Integration by parts won't work on this problem"; "This is a very hard problem." This can easily be input as part of the user's statement of the problem for the system to work on; e.g., the user might designate some methods as being preferred or constrained, he might estimate a resource allotment, etc.

(b) Another type of advice will apply permanently to the nature of the program's functioning (e.g., "Answers, even if uncertain, must be produced in real time;" "Hard problems will be starred.") Such knowledge can be reflected permanently in the design of the program, in its code.

(c) But there is an intermediate level of advice, some information the user knows will concern the next several tasks to be submitted --- not just the present one and not all future ones: "The next four problems will be supplied in order of expected difficulty;" "Several of these problems share common assumptions and raise related issues;" "The user for the next entire session will be a physician;" "Until I find out why F34 returned NIL, I'll be in debugging mode, not production mode." Such advice is rarely communicable even to AI programs.

We believe this type of intermediate-level advice would improve programs if they had the opportunity to accept and use it. In general, we would like to advise the program which capabilities to consider and which to ignore during a prolonged period of solving closely related problems. A fixed (static) approach, for example, would be to define a few modes (e.g., Debug, Demo, Production, Input,...) and permanently build into the code all changes required to specialize the system for each mode. A flexible (dynamic) approach would provide a larger language in which the user could describe his current mode of usage of the program (e.g., by typing "USAGE of KNOWLEDGE-BASE will be INCREASED VERY-MUCH"). The program would translate such statements into changes in its data structures, algorithms, explanations, core management strategies, etc. In the current example, heavy usage of the knowledge base might cause the program to swap out its compiler, optimizers, and other code, enabling it to hold many more knowledge chunks at a time in core. The ability to hold more knowledge in memory would directly affect the rate and probability of productive inferences. Similar effects have been shown for humans: [B. Hayes-Roth 1979, *J. Cog Sci*]

Learning by Observing

Spending a little time/space reconfiguring to reflect the current state of the world (the nature of the recent and succeeding inputs) seems worthwhile; the problem is how to know what that abnormal nature of the environment is. In the last subsection, we discussed the case where the user explicitly tells the program. But this is often impossible (as when the user is unaware of an impending irregularity, or when the "user" is Nature), and almost always a nuisance (learning the language in which to express such advice, remembering to do it, taking the time to do it).

The next step seems to be to allow the program to *infer* such facts about its run-time environment from observations. What kinds of self-monitoring can be usefully employed?

The method which comes to mind first, perhaps, is to save a complete record of everything that ever happened to this program: all the inputs it received (when, from whom), function calls and accesses it made internally, paths it chose/rejected (and why), and outputs it generated. The program could in principle induce, from such an exhaustive history-list, patterns such as "very light usage of compiler;" "SQRT is always called on the result of SUM-OF-SQUARES."

The space costs of such an archive, and the temporal cost of the procedures to infer useful patterns, are surely prohibitive. The more economical approach is to analyze in advance (and change slowly throughout the program's lifetime [Samuel 63]) certain features we would wish to look for in a complete history record, and then to have the program record just enough dynamically to gather and maintain those features. Some of these will be domain-independent and commonly useful: Frequency of calls on various functions, requests for subtasks, etc.; Patterns in the arguments to calls on certain functions and inputs to the program; Patterns in the sequence of calls on functions; Patterns in functions' results (e.g., F34 always seems to return a User-Name). Some of the features will be domain-dependent specializations of the preceding features ("Average degree of input equation" is a special "pattern in the input.")

Learning by Predicting

The theoretician and the engineer share a powerful technique: manipulating a model of a system to generate predictions and identify those possible outcomes that would be desirable. This is a third route to learning. How might programs produce it? What sorts of theories, or models, can be employed to generate new information?

Models of the program's users can be valuable. This includes maintaining and extending profiles of individual users and whole groups. When a user opens a dialogue with Eurisko with the word "Let", it is a safe bet that he's a mathematician. Later, when he

uses the word "function," his membership in that group makes it easy to determine which meaning he has in mind (instead of, say, what a physiologist or a computer scientist or sex therapist might mean by "function.") When the program wishes to mention the square root of -1, it uses its MATHEMATICIAN model to choose an appropriate notation ("i" for mathematicians instead of, say, "j" for engineers). Models can be arranged in a hierarchy, with inheritance of features from more general groups to their subsets, a lattice ultimately terminating in models of individuals. User inputs would be recognized by various models, which would then hypothesize membership in that group (or explicate some future conditions which would confirm or deny such inclusion). Once triggered in this way, the user's apparent groups (and its generalizations) would actively filter his inputs and outputs, could modulate the program's actions (e.g., by changing the worth of the "Proof" concept, which in turn would change the priority rating of tasks on an AM-like agenda and thus alter the program's choice of what to work on next.) New user models can be created either top-down (e.g., when a model has many diverse instances or when it's proven historically to be very valuable, then try to split it into several specializations, each having a share of the general model's instances) or bottom-up (i.e., notice that a group of individuals share a set of common characteristics, and create a new user group out of them).

The preceding kind of prediction is based on a theory of user types, embodied in a set of "typical representative" models. The same source of power can be used in many ways: stereotyping of events (as in scripts), stereotyping of static complexes (as in frames), stereotyping of communities (as in actors, beings, and contract nets). In all these cases, the source of power lies in quickly recognizing probable membership in (applicability of) some schema, after which all the rest of the schema (plus knowledge about all its generalizations) is presumed to be relevant. Programs might build up and utilize scenarios of how problems are solved typically in their domain. The building-up could be top-down via specialization or bottom-up via generalization, or even "sideways" via analogy. The models would be used for prediction (recognize relevant models and activate them, have them hypothesize probable future situations). Together, the models would form a theory of how problems are solved in that domain.

Appendix 4: Cognitive Economy in Any Environment

Summary: A coherent theory of cognitive economy must characterize the sources of power of its methods, and describe ways in which they specialize for environments having various properties. While the body of this paper concentrated upon cognitive economy in fluid environments, we hope to see such a comprehensive theory develop in the future. For that reason, we shall present (briefly, here in an appendix) a few techniques for enhancing cognitive economy in arbitrary environments. We have already described in great detail (Section 2.2) the caching of intermediate results (intelligent redundancy) and will now treat two additional methods: expectation-simplified processing (intelligent focus of attention) and multiple levels of abstraction (intelligent knowledge structuring.)

Appendix 4.1: Expectation-Simplified Processing

Summary: For efficiency's sake, an intelligent system should be willing and able to add new facts, but should be eager to add surprising new facts. Surprises can only be noticed by contrast with expectations, so an intelligent system should maintain a context of expectations and filter incoming observations against that. Furthermore, expectations and surprises can aid an intelligent system in comparing its model and processing of the domain to the real world. Through such monitoring, discrepancies may be found and diagnosed, leading to changes in the model making it more consistent with observed behavior. Our discussion here centers on the importance of using such expectations to focus and filter intelligent processing.

The world bombards our senses with data, much more than we can process in detail in real time; yet we can't live in a hundred times real time. We survive by ignoring most of that data, or, more precisely, by knowing (almost immediately deciding) what can be ignored. We maintain a set of expectations, against which the incoming torrent is matched. Almost all of it will match those expectations, and we then need merely process the unexpected inputs, the surprising observations. We reserve our computing for those opportunities which promise us genuine new facts, rather than reconfirmation of known ones.

Such concentration on the abnormal is a natural side-effect of being guided by expectations --- albeit usually stereotyped ones --- about situations, events, and people. One is able to zip through most of what is encountered, match it immediately to the active schema, and move on. The only processing which is slow and tricky is that dealing with the remaining anomalies, exceptions, puzzles.

The more general position is that each slot indicates not merely a default, but also an indication of how important it is to confirm that value. Some slots will have meta-comments like "I'm worth computing or verifying", or just the opposite. For example, in our "Room" frame, we have a default Ceiling, but in everyday life we know to not squander time checking that each room we're in has a ceiling. Similarly for emergency exit signs: they usually exist, and we rarely feel it's cost-effective to confirm that. We have in both situations a very fast algorithm we can apply if ever we are in a situation in which we need that information. For ceilings, it's $\lambda()$ (look upwards); for exit signs, it's "look for red on white lettering." While this may seem to be a much more complex program to execute, recall that if we truly need to run it --- say in a sudden power failure --- we expect those lit signs to be one of the few things we could still see.

The policy for whether, when, and how to check a default slot value can be arbitrarily complex. The program may be designed to reason at great length about how much time to expend reconfirming the expected. It may reason about the task of reasoning about how much time to spend, and so on.

Several increases in efficiency are realizable through expectation-filtering:

(a). Perceptual set: Expectations allow you to see what you expect with less effort. For emergency exit signs, when we need to find them, we know to look for horizontal and vertical red lines; this is 99% effective. When we look at our red LED watch and have a perceptual set to read the time, we of course would not mistake the image for that of the EXIT.

(b). Surprise: If we observe Y when we expected X, our attention is captured. In one way, sensitivity to data inconsistent with expectations has been heightened. We saw in (a), above, that reliance on frames will dull our chance of noticing a surprising datum (since we will tend to interpret any data as consistent with the current frame). This point (b) says that if we do notice it, then it will have a significant impact. Carefully building up an expectation and then abruptly shattering it with a jarring surprise is the archetypal template for a good joke. As Arthur Koestler observes, the jarring of conflicting matrices of thought is what powers art (*ah!*), science (*aha!*) and humor (*haha!*).

(c). Defer: By relying upon expectations, upon defaults, we can usually defer processing to find the true value in the current case forever. At least, when we do finally have to make a particular perception, it may at that time be easier to make. When the ceiling is falling, it's easier to see; when all the lights go out except for the EXIT signs, it's easier to see them. Generalizing, we may state that it's worth a little computing to reason out whether (and maybe even how long) a slot-filling can be ignored.

(d). Predict and Prepare: This includes the whole continuum we discussed in the Caching section: bind a variable, store the triple $(F.x.F(x))$, store enough information so that $F(x)$ could be recomputed very quickly, store information which would make

$F(x')$ more easily computable (for x' similar to x , and F' similar to F), store away a partial model of the world with good and bad features tagged. Notice that moving along this continuum, programs grow more flexible, more descriptive (less scalar), more dynamic. In all cases, the motivation for doing this storage is to (i) speed up the recognition of (the need to do) similar computations in the future, and (ii) speed up or even eliminate those computations when you decide they are relevant and you want the values they would return. These are the two uses of scripts, but also of cached constants (e.g., caching the value returned by OWNED-BY, a function from CARS to PEOPLE). A better example is *user models*: they range from a single bit (E.g., a NOVICE/PRO toggle), to a few numbers (e.g., in PARRY), to a dynamic concept-modelling scheme (e.g., in Eurisko).

In the interests of cognitive economy, we conclude that programs should build up expectations, monitor them, and change their models as needed. A natural mechanism for doing this is the use of pattern directed knowledge modules (PDMs), which can react to changing situations, to sudden disconfirmations of predictions. These can be used as triggers or demons; e.g., if you hear someone shout "Fire!" in a theater, that takes precedence over Jane Fonda. The second use of PDMs arises when an expectation is not met: the program then also has some chance of modifying the culprit rule(s) that made that false prediction (and/or adding new rule(s) which would make the right one.)

Appendix 4.2: Levels of Abstraction

Summary: An intelligent system's domain model is typically an approximation of reality. It is often the case that knowledge exists at several levels of detail and abstraction. The amount of detail required in intelligent processing depends on the problem-solving context. By appropriately structuring knowledge, a problem-solver can take advantage of abstraction by reasoning at the level appropriate to the given situation. We argue here that abstraction is crucial for intelligent systems which must process over large knowledge bases or within widely-varying time constraints.

In many real-life problem solving situations, the "goal" is not a single, precise answer. Rather, there is a cost-benefit tradeoff between the accuracy of the solution and the amount of resources consumed in producing it. Computer programs which are designed only to return exact answers lie at one extreme end of this curve, and rarely near the optimal point (of diminishing returns). As McCarthy and others have observed, when a coffee cup spills, it's not optimal to carefully integrate the equations of hydrodynamics to decide whether or not to jump away from the table.

This notion of a cost-benefit curve presupposes a kind of "continuity" in the environment, as we discussed in Appendix 2.1. Namely, truncated search is assumed to be better than random selection of a legal move. Almost paradoxically, if the environment is mercurial, then the optimal point on the tradeoff will cause our program to be very expressive.

Cognitive systems which must cope with the world have an omnipresent need to approximate reality --- a need which is well met by maintaining multiple representations of the world at various levels of abstraction; i.e., an entry in the knowledge base has all its generalizations also stored explicitly, redundantly.

Multiple levels of abstraction ("MLA") are useful for quickly approximating reality. In Section 1, we described a situation in which a hospital simulation program might be required upon occasion to provide as good an estimate as possible to a query within a given (short) time period. Another example is the spilling of a coffee cup: the decision of whether to jump back, of what items will get soaked, is made very rapidly, heuristically, not by integrating the hydrodynamic differential equations.

MLA also facilitate *analogizing*: they can make that process easier, faster, and occasionally even more valid. Analogies can be recognized when two distinct concepts coincide at some abstract

level. Conversely, if (corresponding slots of) two apparently related concepts (such as two homonyms) do not share related structure at higher levels of abstraction, that suggests that an analogy between them would be superficial and lacking in power.

Multiple levels of abstraction cost a certain amount to implement, both initially (man-hours of programming) and as the program runs (cpu time spent maintaining the redundancy, choosing which level to work on next, etc.) For problems of all sizes, this feature aids in the expression of new problems to the program, since they may be input at whatever level(s) seem appropriate to the user. Thus multiple levels of abstraction will always add to expressiveness, and will either detract or add to efficiency depending upon how small or large the task is. MLA is very useful when the program's resources are curtailed or, even better, varied dramatically in magnitude from run to run (a program which was *always* limited to a 15-second execution time would be designed specifically for its task, but a program which might get anywhere from 2 seconds to 5 hours for the same simulation question could better benefit from multiple levels of abstraction.) MLA contributes more to efficiency as the quotient (task size)/(resource allotment) increases in size, and as its variance increases.

There is a more general result here: the methods which humans find useful for easy expression of problems may be useful to programs attacking very large problems. For instance, consider the decision about whether to do inference by a system of pattern-invoked experts, or, instead, to preprocess the triggering conditions into a large discrimination network. The former approach is more expressive, the latter more efficient --- at least for small problems. For very large tasks, the need to remain flexible looms very large. It is then more economical to retain pattern-directed control; the compilation of the patterns is so complex the whole procedure would have to be redone frequently during the course of execution, if we opted for that alternative instead. A case in point is the HARPY speech understanding program [Lowerre & Reddy 1979]: whenever the grammar changes --- at any level --- the entire speech recognition network must be recompiled. HEARSAY II [Lesser and Erman 1977], while slower in performance, has sufficient flexibility (from several independent pattern-directed knowledge sources) to obviate such monolithic integrative recompilation delays.

Acknowledgements

We wish to thank Dave Barstow, John Seely Brown, Bruce Buchanan, Ed Feigenbaum, Cordell Green, Russ Greiner, Greg Harris, Barbara Hayes-Roth, Allen Newell, Herbert Simon, Mark Stefik, and Don Waterman for several discussions which have left their mark in this paper.

References

- Adams, J. L., *Conceptual Blockbusting*, W.H. Freeman: San Francisco, 1974.
- Anderson, R. H. and J. J. Gillogly, "Rand intelligent terminal agent (RITA): design philosophy." R-1809-ARPA, The Rand Corporation, Santa Monica, Calif., 1976.
- Balzer, R., N. Goldman, and D. Wile, "Informality in program specifications," *Proc. of the 5th Int. Joint Conf. on Artificial Intelligence*, Cambridge, Mass., 1977, 389-397.
- Barstow, D., "A knowledge-based system for automatic program construction," *Proc. of the 5th Int. Joint Conf. on Artificial Intelligence*, Cambridge, Mass., 1977, 382-388.
- Berliner, H. J., "Chess as problem solving: the developments of a tactics analyzer," Ph.D. Dissertation, Carnegie-Mellon

- University, Pittsburgh, 1974.
- Bobrow, D. G. and B. Raphael, "New programming languages for artificial intelligence research," *Computing Surveys*, 6, 1974, 153-174.
- Bobrow, D. G. and T. Winograd, "An overview of KRL, a knowledge representation language," *Cognitive Science*, 1, 1977, 3-46.
- Brachman, R., "Theoretical studies in natural language understanding", BBN Report 3888, September, 1978.
- Burstall, R. M. and J. Darlington, "A transformation system for developing recursive programs," *J. ACM*, 24, 1977, 44-67.
- Cohen, H., "What is an Image?", *Proc. Sixth International Joint Conference on Artificial Intelligence*, Tokyo, August, 1979, 1028-51.
- Collins, A. M. and M. R. Quillian, "Retrieval time from semantic memory," *J. Verbal Learning and Verbal Beh.*, 8, 1969, 240-247.
- Collins, A. M. and M. R. Quillian, "How to make a language user," in (E. Tulving and D. Donaldson, eds.) *Organization of Memory*, Academic Press: New York, 1972.
- Conrad, C., "Cognitive economy in semantic memory," *J. Experimental Psychology*, 92, 1972, 149-154.
- Darlington, J. and R. M. Burstall, "A system which automatically improves programs," *Proc. of the 3th Int. Joint Conf. on Artificial Intelligence*, Stanford, Calif., 1973, 479-485.
- Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall: Englewood Cliffs, N.J., 1976.
- Feigenbaum, E. A., "The art of artificial intelligence: themes and case studies of knowledge engineering," *Proc. of the 5th Int. Joint Conf. on Artificial Intelligence*, Cambridge, Mass., 1977, 1014-1029.
- Fikes, R. E., P. E. Hart, and N. J. Nilsson, "Learning and executing generalized robot plans," *J. Artificial Intelligence*, 3, 1972, 251-288.
- Fiksel, J. R. and G. H. Bower, "Question answering by a semantic network of parallel automata," *J. Math. Psych.*, 13, 1976, 1-45.
- Gelernter, H., "Realization of a geometry-theorem proving machine," in (E. A. Feigenbaum and Julian Feldman, eds.), *Computers and Thought*, McGraw-Hill: New York, 1963, 134-152.
- Green, C., "A summary of the PSI program synthesis system," *Proc. of the 5th IJCAI*, Cambridge, Mass., 1977, 380-381.
- Green, C. and D. Barstow, "On program synthesis knowledge," *J. Artificial Intelligence*, 10, 1978, 241-279.
- Hayes-Roth, B. and F. Hayes-Roth, "Plasticity in memorial networks," *J. Verbal Learning and Verbal Behavior*, 14, 1975, 506-522.
- Hayes-Roth, B. and F. Hayes-Roth, "The prominence of lexical information in memory representations of meaning," *J. Verbal Learning and Verbal Beh.*, 16, 1977, 119-136.
- Hayes-Roth, B. and C. Walker, "Configural effects in human memory," *J. Cognitive Science*, 1979.
- Hayes-Roth, F., P. Klahr, J. Burge, and D. J. Mostow, "Machine methods for acquiring, learning, and applying knowledge," P. 6241, *The Rand Corporation*, Santa Monica, 1978.
- Hayes-Roth, F. and V. R. Lesser, "Focus of attention in the Hearsay-II speech understanding system," *Proc. of the 5th IJCAI*, Cambridge, Mass., 1977, 27-35.
- Kahn, K., "Making Aesthetic Choices", *Proc. of the 6th Int. Joint Conf. on Artificial Intelligence*, Tokyo, 1979, 448-50.
- Kant, E., "The selection of efficient implementations for a high level language," *Proc. ACM SIGART-SIGPLAN Symp. on Artificial Intelligence and Programming Languages*, SIGART Newsletter #64, 1977, 140-146.
- Klahr, P., "Planning techniques for rule selection in deductive question-answering," in (D. A. Waterman and F. Hayes-Roth, eds.), *Pattern-Directed Inference Systems*, Academic Press: New York, 1978.
- Knuth, D., *Fundamental Algorithms (The Art of Computer Programming, Volume 1)*, Addison Wesley, 1968.
- Kuhn, T., *The Structure of Scientific Revolutions*, 1972.
- Lakatos, I., *Proofs and Refutations*, Cambridge University Press: Cambridge, 1976.
- Lenat, D. B., "Beings: knowledge as interacting experts," *Proc. of the 4th Int. Joint Conf. on Artificial Intelligence*, Tbilisi, USSR, 1975, 126-133.
- Lenat, D. B., "AM: an artificial intelligence approach to discovery in mathematics as heuristic search," Memo AIM-286, Stanford AI Lab, 1976.
- Lenat, D. B., "The Nature of Heuristics II and III", *J. of Artificial Intelligence*, 20, 1, March, 1983.
- Lesser, V. R. and L. D. Erman, "A retrospective view of the Hearsay-II architecture," *Proc. of the 5th Int. Joint Conf. on Artificial Intelligence*, Cambridge, Mass., 1977, 790-800.
- Low, J., "Automatic coding: choice of data structures," Memo AIM-242, Stanford University, 1974.
- Lowerre, B. and D. R. Reddy, "The Harpy system," in (W. Lea, ed.), *Recent Trends in Speech Recognition*, Prentice-Hall: Englewood Cliffs, N. J., 1980.
- McCarthy, J., "The advice taker," in (M. Minsky, ed.), *Semantic Information Processing*, MIT Press: Cambridge, 1968.
- Mostow, J. and F. Hayes-Roth, "Machine-aided heuristic programming: a paradigm for knowledge engineering," N-1007-NSF, The Rand Corporation, Santa Monica, Calif., 1979.
- Newell, A., J. C. Shaw, and H. A. Simon, "Chess-playing programs and the problem of complexity," in (Feigenbaum and Feldman, eds.) *Computers and Thought*, McGraw-Hill: New York, 1963, 39-70.
- Newell, A. and H. A. Simon, *Human Problem Solving*, Prentice-Hall: Englewood Cliffs, N.J., 1972.
- Nilsson, N., *Problem Solving Methods in Artificial Intelligence*, 1972.
- Rich, E., "Building and Exploiting User Models", *Proc. of the 6th Intl. Joint Conf. on Artificial Intelligence*, Tokyo, 1979, 720.
- Rips, L. J., E. J. Shoben, and E. E. Smith, "Semantic distance and the verification of semantic relations," *J. Verbal Learning and Verbal Behavior*, 12, 1973, 1-20.
- Samuel, A., "Some Studies on Machine Learning of Checkers," in (E. A. Feigenbaum and Julian Feldman, eds.), *Computers and Thought*, McGraw-Hill: New York, 1963.
- Sandewall, E., private communication, August 22, 1979.
- Stefik, M., "An examination of a frame-structured representation system," Memo HPP-78-13, Stanford University, 1978.
- Waterman, D. A., R. H. Anderson, F. Hayes-Roth, P. Klahr, G. R. Martins, and S. Rosenschein, "Design of a rule-oriented system for implementing expertise," The Rand Corporation, Santa Monica, 1979.
- Waterman, D. A. and F. Hayes-Roth, *Pattern-Directed Inference Systems*, Academic Press: New York, 1978.

THE ROLE OF EXPERIMENTATION IN THEORY FORMATION

Thomas G. Dietterich

Bruce G. Buchanan

Department of Computer Science
Stanford University
Stanford, CA 94305

Abstract

This paper presents an analysis of the role of experimentation in theory formation. Experimentation serves three purposes: (a) hypothesis testing, (b) gathering of new data to constrain the theory generator, and (c) manipulation of the external system to reveal its structure. A theory formation system, EG, is described that employs experimentation and observation techniques to develop a theory of the UNIX file system and executive-level file commands. This theory formation task is more complex than previous efforts, and the goal of the project is to determine which existing theory formation methods are applicable and what new methods need to be developed. Previous techniques are reviewed, and none of them are found to be applicable. A new technique, based on controlled experimentation, is described, and a hypothetical trace of EG's execution is presented.

Key terms: Theory formation, generate-and-test, controlled experimentation, exploratory experiments, observation experiments, hypothesis-test experiments, credit-assignment, new terms.

1 Introduction

Despite major advances in recent years, Artificial Intelligence models of learning and scientific theory formation are still primitive. This paper describes the EG project—whose goal is to investigate theory formation in a domain that bears a closer resemblance to ordinary scientific domains than previous AI theory formation tasks. We have chosen the task of constructing a theory of the UNIX¹ operating system, because it provides ample opportunities for experimentation under program control. In this paper, we describe this theory formation task and then review previous approaches to theory formation including generate-and-test, version spaces, Meta-DENDRAL, and AM/Eurisko. From this review, we conclude that heuristic search techniques similar to those used in Eurisko are applicable to this domain with modifications to increase the role of experimentation. Three basic functions of experimentation are (a) to test hypotheses, (b) to gather more data for guiding the theory formation process, and (c) to explore the various states of the system being modelled. The paper concludes with a description of the EG program and a hypothetical example of its operation.

1.1 Description of the theory formation task

The main goal of the EG project is to investigate a large-scale theory formation problem to determine which existing AI techniques can be scaled-up and what new techniques need to be developed. The

theory formation task that EG must solve is to develop a theory of the UNIX) operating system by designing, performing, and analyzing experiments with UNIX executive commands. The theory to be developed has four components (see Figure 1): input/output syntax, executive-level commands, system calls, and file-system data structures.

-
- Input and output syntax. (E.g., `Incommand ::= 1n source dest`)
 - Abstract programs for each executive command. (E.g., `1n command`, `cp command`)
 - System calls. (E.g., `Create-file`, `Delete-file`, `Change-working-directory`)
 - File system data structures. (E.g., file directories, protection codes, current working directory)
-

Figure 1: Four components of EG's theory of UNIX

Input-output syntax includes the syntax of strings typed by the user (commands, file names, arguments, switches, and so on) as well as the syntax of strings printed by the computer (directory listings, prompt characters, error messages, and so on).

The command component of the theory involves describing the behavior of each executive command as an abstract program. There are two basic classes of commands: observation commands, which simply display information about the current state of the system, and manipulation commands, which perform some action on the file system. Manipulation commands typically involve such steps as checking the prerequisites of the action to be performed, printing error messages, requesting confirmation for dangerous actions, and, of course, invoking the appropriate system calls to perform the actions.

The final two components of the theory are closely coupled. The file system is described as a set of mappings and data structures that implement those mappings. The system calls are then defined as operations on these data structures. For example, there is a many-to-one mapping from file names to files, and this mapping is implemented as a table in a directory file. The `Create-file` system call adds a new entry to this table (as well as writing a file on the disk).

This theory is substantially different from any theories developed by previous learning programs in the inter-connectedness of the concepts in the theory. "Theories" formed by Lenat's AM [1976] and Eurisko [1982] programs, for example, are simpler in that they investigate systems of nearly-independent concepts. Each concept can

¹UNIX is a trademark of Bell Laboratories.

be defined and evaluated without modifying the definition of any other concepts. In EG, however, a change in the definition of a data structure, for example, can have far-reaching effects on the definitions of system calls, commands, and even I/O syntax.

In defining this particular task, we were guided by an analogy with the way people learn about operating systems. People are rarely taught the details of an operating system. Instead, they read manuals and then conduct experiments. An examination of any operating systems manual will reveal many incompletenesses. For the UNIX manuals we have inspected [Kernighan and McIlroy, 1976; Lomuto and Lomuto, 1983], the syntax and main behavior of each command are well-specified. However, the underlying structure of the file system and the exact behavior of executive commands in unusual situations are very incompletely described. By analogy, EG starts with a fairly complete model of the I/O syntax of UNIX. It is given partial models of the more common executive commands and very little information about the underlying data structures and system calls. It also has a body of general knowledge about computers (e.g., data structures for implementing a many-to-one mapping) and knowledge about the kinds of actions that one would normally want to perform on files and similar data objects.

1.2 Review of previous work in theory formation

Given this knowledge about operating systems in general and UNIX in particular, there is still an immense space of possible theories. We believe that clever experimentation is required in order to search this space efficiently. Before describing the approach that EG employs, let us review existing techniques for searching large hypothesis spaces.

The most naive approach to searching any space is to generate all candidate solutions and test them to see if they are indeed solutions. In this domain, this approach corresponds to generating all theories of UNIX consistent with the starting knowledge base and testing them through experiments. This approach is grossly inefficient, because it does not use the results of hypothesis testing to improve the search as it progresses.

Mitchell's [1978] candidate-elimination algorithm is a technique that completely incorporates information obtained from each test into the generator of possible hypotheses. We believe that the version space approach would not work in this domain, however, because the theory of UNIX is too large to be usefully viewed as a single "concept." The UNIX theory has many nearly-independent subcomponents (e.g., individual commands) that give the space of possible theories a very branchy partial order. Consequently, the candidate-elimination algorithm would not perform well in this space.

The approach employed in Meta-DENDRAL appears initially to be applicable. Meta-DENDRAL is given a set of molecules (with known structure) and their corresponding mass spectra, and its goal is to develop a theory that predicts how molecules will break apart inside the mass spectrometer. It employs what has been termed the "single-representation trick" [Dietterich, et al., 1982] to search the space of possible cleavage rules. First, each data point in each mass spectrum is translated into a set of specific cleavage rules that could have caused that data point to appear. Then syntactic pattern matching methods are applied to generalize these cleavage rules to obtain a small set of general cleavage rules that explain most of the data. An analogous approach in EG's domain would start by gathering training instances of commands and their behaviors. Then the set of maximally-specific UNIX theories that could explain these behaviors would be computed. Finally, syntactic methods would be applied to generalize these theories to obtain a single theory that explained all of the behavior. Unfortunately, this approach is not feasible because the set of

maximally-specific theories that explain a given command is too large to compute. Furthermore, it is not obvious which data should be gathered initially to guide the theory formation process. A more dynamic approach appears to be called for.

The AM and Eurisko programs employ a dynamic, opportunistic approach to discovery tasks. It is worthwhile investigating how analogous methods might be applied to EG's task. AM and Eurisko are discovery systems that search a space of possible constructs looking for interesting ones. A direct application of Eurisko to the operating system domain would involve giving Eurisko general knowledge about operating systems along with heuristics for the design of good operating systems and then letting it run to see what interesting systems it discovers. To convert Eurisko into a theory formation system we need to replace the "interestingness" heuristics with techniques that force Eurisko to generate theories that explain the observed behavior of UNIX. This is roughly the approach we are taking. In place of "interestingness" heuristics, EG plans and carries out observation experiments to reveal important structure in UNIX. This structure is then incorporated into the growing theory of the operating system.

1.3 The role of experiments in learning and theory formation

Experimentation (or active instance selection) has been employed in several learning and theory formation systems (see [Buchanan, Mitchell, Smith, and Johnson, 1977]). It provides two basic functions: testing hypotheses and gathering further information to constrain the hypothesis generator.

The simplest learning methods rely only upon the ability of an experiment to test the truth of some proposition and return a YES/NO answer. In pure hypothesis testing, this answer merely indicates that the proposition (or supporting theory) is correct or incorrect. However, it is possible to use this information to help guide the theory generator. For example, as we mentioned above, Mitchell's [1978] version space approach guarantees that the generator will generate only theories that are consistent with the data. Hence, every YES/NO answer returned from an experiment is completely incorporated into the generator of theories. In this role, experiments can be viewed as providers of new data from which the theory is developed. Even this role has not been developed much in existing learning systems. Substantial speedups in learning can be obtained by clever use of experiments to "split the hypothesis space in half" (see [Mitchell, 1978]).

It is possible to exploit YES/NO answers further through credit assignment. If credit and blame can be assigned to individual components of the candidate theory that was tested, then the theory generator can be instructed to modify only the parts of the theory that are still incorrect. There are many techniques for assigning credit and blame. One approach described by Dietterich and Buchanan [1981] is to perform controlled experiments. The generator can generate a set of hypotheses by varying only one component of the hypothesis at a time. Each hypothesis is then tested, and a YES/NO answer is obtained. Because only one component has been varied since the last test, all credit and blame can be assigned to that component for any changes in the overall evaluation of the hypotheses. This strategy only makes sense if it is possible to factor each complete hypothesis into independent subcomponents. If this can be accomplished, then each subcomponent can be tested independently prior to combining the subcomponents into one complete hypothesis. If such a factoring does not exist, then this strategy reduces to exhaustive generate-and-test.

In some domains—including the proposed operating system domain—more than a simple YES/NO answer is available from experiments. In Arthur Samuel's [1967] book-move version of his

checkers learning system, for example, each experiment not only tested whether a proposed move was the correct one—it also returned the correct move as a result. This information was then employed to adjust the current hypothesis.

The more information an experiment provides, the more the learning or theory formation system can learn from that experiment. For example, in the LEX learning system, each experiment returns a complete trace of the efforts of the problem solver to solve the given integration problem. A powerful analytic technique has recently been developed for LEX that involves analyzing this trace and attempting to "invert" each integration operator that was part of the optimal solution path. If this inversion can be accomplished, then the evaluation criteria for what it means to solve a problem can be "pushed through" these inverted operators to compute directly the heuristics for each of the operators in the solution sequence. Utgoff [1982] has even employed this technique to compute the definitions of new terms to be included in the generalization language of the system.

We hope to show with EG that similar techniques can be employed in a more complex theory formation task. A very important difference between EG and all previous theory formation systems, however, is that the system being experimented upon is external to the theory formation system. Consequently, it is not possible to observe directly its internal state. This makes evident a third function of experimentation—it permits the theory formation program to manipulate the external system in order to reveal its internal structure. In the UNIX domain, for instance, experiments can force UNIX into a new state that has never been previously observed. For example, a command to change the current working directory makes a variety of new behaviors observable including changes in the relative names of all files in the system. Once a new state is entered, observation commands can be issued to learn more about that state. From this perspective, it can be seen that EG's task is to explore two spaces: the space of possible operating system theories and the space of possible states of the external system.

1.4 Summary

We are now in a position to state the two basic hypotheses of the EG project:

1. A heuristic search technique similar to that employed in Eurisko is sufficient to discover theories in this domain, and
2. Planned experimentation is essential in large domains in order to avoid combinatorial explosion in the search for theories.

Both of these hypotheses rest on a more fundamental claim: Theory formation can only be accomplished if the system under study can be factored into weakly-interacting components. The essence of heuristic search is to subdivide the overall theory formation task into small subtasks to which heuristics can be applied. The essence of experimentation is to isolate some subcomponent of the system under study in order to observe its behavior (and thus provide guidance to the heuristic search for theories).

This project can be viewed as an attempt to test these hypotheses by developing a new experimentation-based theory formation method.

2 Forming theories about operating system commands

As described in section 1, EG starts with a partial theory of the UNIX file system and incrementally improves this theory by conducting online experiments with a UNIX system over an Ethernet connection. The starting theory of UNIX includes the basic commands for listing file directories, copying files, and deleting files. Also included is an initial theory of the file system and the system calls that manipulate it. This initial theory is incomplete and oversimplified, but it includes enough information to enable EG to manipulate and observe the state of the UNIX system. EG's task is to fill in the missing parts of the theory and deepen the oversimplified parts in order to obtain a complete and accurate theory (at the given level of description).

This section starts by describing the architecture of the EG program. Then a hypothetical example of EG's behavior is presented.

2.1 Program architecture

EG is designed as a set of knowledge sources that interact through a global database and a global agenda. The global database (also referred to as the blackboard) contains a representation of the current theory and any alternatives under consideration. The global agenda is an ordered list of outstanding tasks to perform. An executive routine repeatedly analyzes the agenda, chooses a task, and invokes a knowledge source to perform the task. As the knowledge source runs, it may post new tasks on the agenda.

Tasks fall into two major categories. One kind of task is placed on the agenda when some new information has been posted on the blackboard. This kind of task asks that some knowledge sources be invoked to compute the consequences of this new information and propagate them to other components of the theory. The second kind of task identifies some hole in the theory. It asks that some knowledge sources be invoked to design and execute an experiment to help fill this hole.

Tasks are ranked on the agenda according to their importance. Importance is determined by a collection of heuristic rules such as "If some other task is depending on the results of this task, then this task is important" or "If this task might indicate a serious error in our theory, then this task is important."

From the most abstract viewpoint, the seven knowledge sources in EG can be grouped into two categories: those that generate theories and those that test theories. There are four generator sources, one for each component of the theory:

- TIO—Input/Output theorist
- TC—Command program theorist
- TSC—System call theorist
- TDS—Data structure theorist

Each of these has the task of generating possible modifications to the corresponding component of the UNIX theory. These modifications may involve expanding the theory to include a finer level of detail or revising the theory to correct errors.

There are three knowledge sources that test theories and perform experiments. These are

- The experiment proposer—which examines a problem and attempts to find an experimental method that will help solve it.
- The experiment planner—which develops a specific experiment to perform, and
- The experiment monitor—which performs the experiments and gathers the raw responses from UNIX.

The general behavior of the system is to repeatedly identify a hole in the theory, design and execute an experiment to gather some data, and then invoke appropriate theory-generator knowledge sources to analyze the data and modify the theory.

2.2 An extended example: the ln command

This section presents an extended example of EG as it attempts to understand the **ln** command. This command permits the UNIX user to create an alias name for an existing file. After the command **ln source dest** is executed, both names **source** and **dest** point to the same file, so that changing one file changes the other.

We will start by describing the state of EG's UNIX theory at the point where it begins to work on the **ln** command. Then, we will present a hypothetical sequence of knowledge source invocations.

Suppose that EG has been working for a while on the structure of the file system. It has discovered that the file system is structured as a tree of directories and now it is investigating commands (whose names and syntax it already knows) to see what effects they may have on the file system. One of these commands is the **ln** command. Here is the state of knowledge at this point in EG's researches.

Recall that the theory of UNIX is built of four components: file system data structures, system calls, command programs, and I/O syntax. EG's current theory of data structures states that the UNIX file system is structured as a tree, where each interior node in the tree is a directory and each leaf node is a file. There is a global state variable that points to a particular directory in this tree called the current working directory. A file is designated by giving a path through this tree from the current working directory to the desired file or directory. A downward link in the path is specified by giving the name of the file or directory. An upward link is specified by giving the special name **..**. Each directory implements a mapping between file names and actual files. This mapping is many-to-one and converts file names into unique global identifiers of files. The mapping is implemented in the directory as a table in which file names and file identifiers are juxtaposed. A unique file identifier is an address that points to the actual string of characters that makes up the file. Each file can be thought of as having a canonical path name that starts at the root of the file structure.

The second component of EG's current theory—the system call component—includes the following primitive actions:

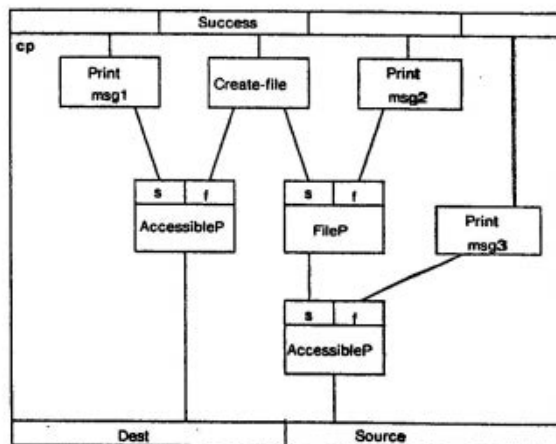
- **create-file(name, contents)**
Create a new file with the indicated name and contents. This is defined as creating a new unique file identifier, storing the **contents** at that address, and adding a new row **<name, unique identifier>** to the appropriate directory.
- **delete-file(name)**
Delete the file with the indicated name. This is defined as removing the directory entry for **name**.

- **rename-file(oldname, newname)**
Change the name of a file. This just changes the name field in the appropriate directory entry.
- **addlink(source, dest)**
Create a new name (**dest**) for an existing file (**source**). This is defined as adding a new entry to the appropriate directory associating **dest** with the same unique identifier as **source**.
- **filep(name)**
Check to see if **name** is the name of a file and not a directory.
- **accessiblep(name)**
Check to see if **name** is defined as either a file or directory.

For each of these primitive actions, there are prerequisites that can be inferred from the current theory of the file-system data structures. For instance, **Create-file** only works if the name does not already exist. **Addlink** requires that the **source** exist and that the **dest** not exist. **Filep** has **Accessiblep** as a prerequisite.

The third component—the command procedure component—includes procedures for the following commands:

- **ls**
Print a list of the file names in the current working directory.
- **ls -ld name**
Print file information about the file or directory indicated by **name**.
- **cp source dest**
Create a new file **dest** whose contents are copied from **source**.
- **rm source**
Delete the file **source**.



- msg1: Dest: File exists
 msg2: Source is a directory
 msg3: Source: No such file or directory

Figure 2: Level 2 theory of the **cp** command.

For each of these commands, EG has an abstract program that defines the command in terms of the system calls given above (see Figure 2). To represent these programs, I am using a slight modification of the programmer's apprentice representation for "deep plans" (see [Rich and Shrobe, 1976]). In this notation, a program is represented as a partially-ordered graph of boxes. Each box denotes a primitive action to be performed. The boxes are linked by control paths that indicate that one box must be performed before another. Conditional boxes have multiple outputs, one for each possible outcome of the action. Although not shown in this simple example, this notation also provides methods for representing iteration and recursion.

Finally, the syntax component of EG's UNIX theory includes a complete grammar for file and directory syntax and for command syntax. The following BNF rules show a portion of this grammar.

```
command ::= lncommand | cpcommand
          | othercommand

lncommand ::= ln sp filename sp filename
cpcommand ::= cp sp filename sp filename

filename ::= legalchar
           | filename legalchar
             (repeat up to 13 times)

sp ::= " "
     | sp " "
```

Notice that the syntax for the **ln** command is provided. The syntax for several commands is given to EG at the start. However, the BNF rule for **command** also includes the nonterminal **othercommands**, which has no rules defining it. Hence, EG knows that its list of commands may not be complete.

Given this body of knowledge, let us suppose that EG has performed an **ln** command and inductively inferred that the purpose of the **ln** command is to perform an **Addlink** operation on its two arguments. Here is one sequence of knowledge-source invocations that might occur.

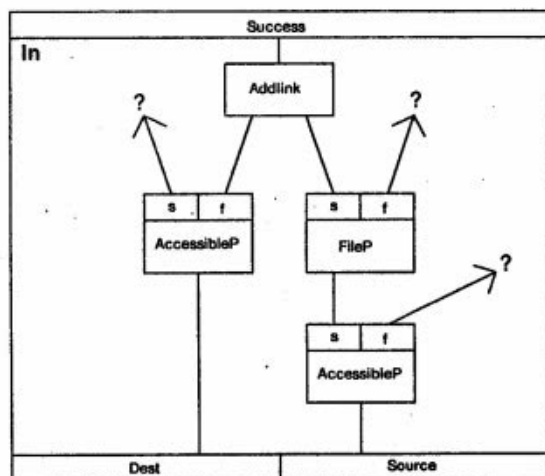


Figure 3: Initial theory of the **ln** command.

Step 1. The TC knowledge source is invoked to build a theory of the **ln** command. In other words, it is called upon to develop all possible abstract programs that might implement the **ln** command using the available system calls. TC produces only one theory for **ln**, which is shown in Figure 3. (In my first implementation of this system, the TC builds this theory by backward chaining from the goal of the command.) This theory says that the main step of the command is to perform an **Addlink** operation on two arguments, the source and destination file names. In order to do this, all of the prerequisites of **Addlink** must be checked. Hence, test boxes have been included that check (a) that the source file is accessible, (b) that the source file is a file, and (c) that the destination file is not accessible. Unfortunately, this is not a complete program for the **ln** command. There are three dangling links: cases where TC cannot guess what the **ln** command will do. These correspond to the three cases in which an error is present in the command's arguments. Hence, TC posts three tasks on the agenda—one for each "hole" in the theory.

Step 2. The experiment proposer is invoked to design experiments that resolve the first of these ambiguities—determining what happens when the source file is not a file. The basic plan for such an experiment is to issue an **ln** command for which **filep(source)** is false and observe what happens. Such an experiment will only be valuable if the observation can be made and the output successfully interpreted. Hence, the experiment proposer must find a set of constraints on the arguments to the **ln** command that force control out the desired branch, while at the same time preventing any other, possibly confusing, events from happening.

I have developed a simple marker-propagation algorithm that can perform this computation. It starts by applying labels to the desirable and undesirable paths in the program. Then, these labels are propagated through the various conditional boxes to determine how each conditional box should be "set." This problem is closely related to the problem of line sensitization in digital circuit test generation [Roth, 1966].

In this particular case, the solution is to choose a source name that is accessible, but that is not a file name (i.e., it is a directory name), and to choose a destination name that is not accessible. An **ln** command with those arguments will not perform the **Addlink** operation nor will it send control out any of the other dangling paths. Once the experiment proposer has developed this plan, it adds a task to the agenda suggesting that the experiment planner be invoked to complete the design of this experiment.

Step 3. The experiment planner is invoked to choose specific names for this experiment. If EG does not have an up-to-date model of the current state of UNIX, then some observation commands must first be issued to find out what files exist. Then, the choice of file names is simple. In general, the problem of choosing specific arguments for an experiment is somewhat tricky, since the choices may not be independent of one another. In such cases, techniques similar to those employed by Stefik [1980] will need to be applied to make some choices and then use those to constrain the other choices. In addition to planning the main step, the experiment planner must also plan the data-gathering commands that are to be issued. In this case, we have some heuristics that tell us that the likely effects of this command are to modify the objects named by its arguments and to display some output. Hence, the following plan is developed:

```
ls -ld d
Should display
"drwxr-xr-x 2 x1      32 Apr 20 21:04 d"
ls -ld b
Should give the error message "b not found"
ln d b
May print an error message
```

```
ls -ld d
Should display
"drwxr-xr-x 2 x1      32 Apr 20 21:04 d"
ls -ld b
Should give the error message "b not found" or else
directory information about b
```

where **d** is a directory that already exists, and **b** is a non-existing name. Observation actions (the **ls** commands) have been placed before and after the key step (the **ln** command). Notice that the experiment planner computes expectations so that the experiment monitor can determine whether things are "on-course." If **ln** were judged to be a dangerous command, then commands would have been included to create a scratch directory instead of using an existing directory.

Step 4. Now the experiment monitor is invoked to perform the experiment and collect the results. The execution of this experiment produces the following responses from UNIX:

```
% ls -ld d
drwxr-xr-x 2 x1      32 Apr 20 21:04 d
% ls -ld b
b not found
% ln d b
d is a directory
% ls -ld d
drwxr-xr-x 2 x1      32 Apr 20 21:04 d
% ls -ld b
b not found
%
```

(Here are some notes to aid the reader in understanding this trace. The **%** is the UNIX prompt character. The output of the **ls** command—for example "**drwxr-xr-x 2 x1 32 Apr 20 21:04 d**"—is a single line giving information about the file or directory named by its argument. This information includes the protection bits, the owner, the length in bytes, the modification date, and the name.)

Once the experiment monitor has obtained these responses from UNIX, it posts them on the blackboard and adds a task to the agenda suggesting that some knowledge source should interpret this data.

Step 5. The TIO knowledge source interprets these data to mean that the **ln d b** command had no effect except to print the message "**d is a directory**". This is interpreted as an error message, but, since EG has no ability to understand English, the only thing it can notice about the message is that it mentions the **d** argument (this is typical for UNIX error messages). It is important to note that the problem of interpreting the data is inherently underconstrained. There are two reasons for this. First, the meaning of a string depends on the context in which it appears. A given string appearing in the output, for example, could be the contents of a file, since files can contain arbitrary strings. Or, the string might contain several subfields that denote various objects in the computer, such as protection codes, user names, times and dates, and so on. Second, EG's I/O grammar is never complete, so it is always conceivable that a given string denotes some new class of objects that has never before been encountered by EG. Given this lack of constraint in the parsing process, it is critical that EG be given some knowledge about what to expect. In this context, since we are violating a prerequisite of a primitive action, we are expecting an error message or else some change in one of the command arguments.

Once TIO has interpreted the output, it places this interpretation on the blackboard and adds a new task to the agenda to propagate the consequences of this interpretation to other components of the theory.

Step 6. Hence, the TC knowledge source is invoked to update the command component of the theory in light of the fact that **ln d b**

produced only an error message and had no other effects. The **ln** command theory is updated by attaching a **Print** action of the form **Print(concat(source, " is a directory"))** to the dangling link coming from the **FileP** test.

The process of filling in the dangling links of the **ln** command program continues in this way until the plan shown in Figure 4 is obtained. This theory might serve quite well for a long time, but at some point, EG might attempt to create a link that crosses device boundaries. The starting theory of the file system sketched above is incorrect, because it assumes that a directory maps names to unique file identifiers. It turns out that directories map names to file identifiers that are unique for a given device. The combination of the device and the file identifier uniquely designates the file. As a consequence, it is impossible to create links across devices, since the linking operation involves changing only the within-device file identifier.

Suppose that EG attempted the following cross-device **ln** command:

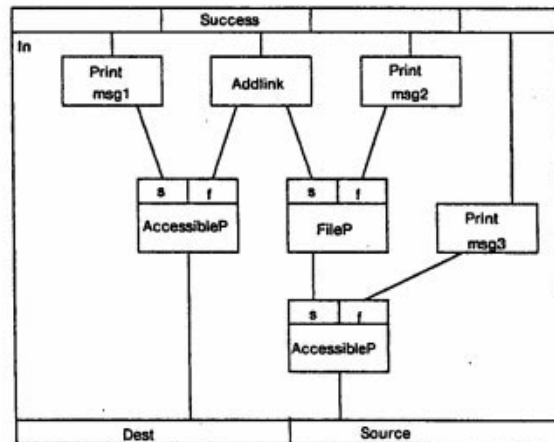
```
% ln filea /tmp/filea
filea: Cross-device link
%
```

The line **filea: Cross-device link** is an unexpected error message. Here is the sequence of knowledge source invocations that results:

Step 45. The experiment monitor notes immediately that this message violates the expectations for the experiment in progress (i.e., this **ln** command wasn't supposed to produce any output at all). Hence, the execution of the experiment is halted, and TIO is queued to interpret this message.

Step 46. TIO interprets this as an error message (since it is a single line and mentions one of the arguments to the **ln** command).

Step 47. TC updates the theory to include an new conditional box. This box is testing some unknown condition at some unknown point in the **ln** program and printing the error message if the test succeeds. The TC can give this condition box a synthetic name, **Cross-device**, but it can't be sure what arguments are being

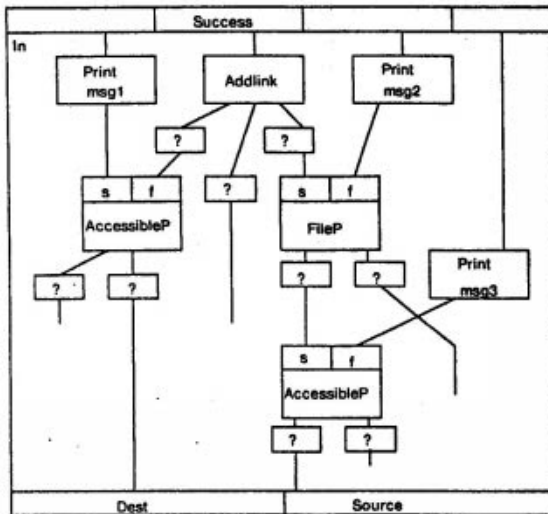


msg1: Dest: File exists

msg2: Source is a directory

msg3: Source: No such file or directory

Figure 4: Theory of **ln** after all dangling paths have been observed



Where ? stands for

msg1: Dest: File exists
 msg2: Source is a directory
 msg3: Source: No such file or directory
 msg4: Source: Cross-device link

Figure 5: Possible locations of the Cross-device test

tested. Any data structure in the operating system might conceivably be checked, and EG doesn't even have a complete list of such data structures. Figure 5 shows the possible locations of the Cross-device check.

TC identifies two problems that must now be solved. First, the exact positions of Cross-device in the `ln` program need to be determined. Most error messages indicate violated prerequisites, hence, the position of each Cross-device check will indicate that some primitive action has this condition as a prerequisite. Second, once the positions have been identified, the prerequisite information must be propagated to the higher levels of the theory. These two problems are added to the agenda.

Step 48. The experiment proposer is invoked to work on the first problem. It has two methods for pinning down the position of a test. One is to hold the arguments to `ln` constant while varying the objects that they denote, and the other is to compare the `ln` command with commands that contain some of the same primitive boxes. By applying the first method, the experiment proposer proposes that `ln filea /tmp/filea` be attempted, but with `filea` now being a non-existent file.

Step 49. The experiment planner must figure out a way to accomplish this. For example, EG could simply delete `filea`. However, if the contents of `filea` are valuable, then EG should instead rename `filea`. In this case, EG chooses to delete `filea` using the `rm` command:

```
rm filea
Should produce no output
ln filea /tmp/filea
Should either print
filea: No such file or directory
or else filea: Cross-device link
ls -ld filea
Should print filea not found
ls -ld /tmp/filea
Should print /tmp/filea not found
```

Step 50. Plan execution produces the following output:

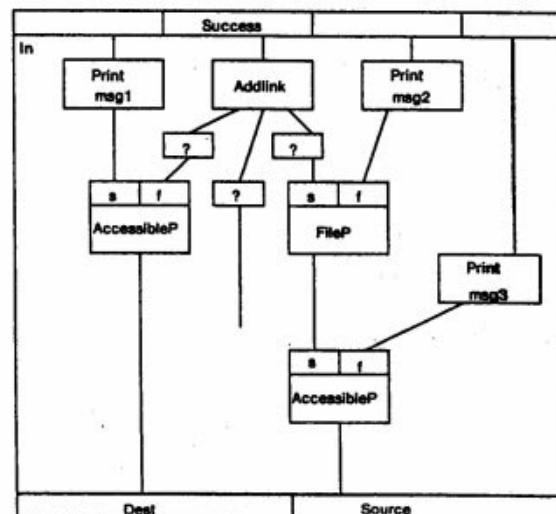
```
% rm filea
% ln filea /tmp/filea
filea: No such file or directory
% ls -ld filea
filea not found
% ls -ld /tmp/filea
/tmp/filea not found
%
```

Step 51. TIO interprets this as being the expected output of the `ln` command when the source file does not exist.

Step 52. TC can therefore conclude that, since Cross-device must be checking something besides the existence of `filea` and since nothing else changed from one experiment to the next, Cross-device should also have been true in this case. Hence, Cross-device is not a prerequisite of AccessibleP.

Similar experiments can be conducted to show that Cross-device is not a prerequisite of the other AccessibleP test or the FileP test. Hence, Cross-device must be a prerequisite of the Addlink action. The resulting theory is shown in Figure 6. Note that the exact position of the Cross-device test is still not known.

An alternative research strategy would be to compare the `ln` command with the `cp` command. Initially, it appears that these



msg1: Dest: File exists
 msg2: Source is a directory
 msg3: Source: No such file or directory

Figure 6: Locations of Cross-device after experimentation

commands have identical prerequisite structures (compare Figures 2 and 3). The only difference is that the final step in the **cp** command is a **Create-file** instead of an **Addlink**.

Step 49 (alternative approach). The experiment proposer suggests that the command **cp filea /tmp/filea** be executed to see if the **Cross-device link** message will be printed. In this experiment (as in the previous step 49), **filea** names an existing file and **/tmp/filea** names a non-existing file.

Step 50 (alternative approach). The experiment planner develops the following experiment:

```
cp filea /tmp/filea
  Should produce no output or else
  the message filea: Cross-device link
ls -ld filea
  Should print
  -rw-r--r-- 1 x1      27 Apr 27 10:44 filea
ls -ld /tmp/filea
  Should print directory line for /tmp/filea
  or else the message /tmp/filea not found
```

Step 51 (alternative approach). The experiment monitor performs this experiment and obtains the following output:

```
% cp filea /tmp/filea
% ls -ld filea
-rw-r--r-- 1 x1      27 Apr 27 10:44 filea
% ls -ld /tmp/filea
-rw-r--r-- 1 x1      27 Apr 27 10:45 /tmp/filea
%
```

Step 52 (alternative approach). TIO interprets the two output lines as directory information.

Step 53 (alternative approach). TC interprets this as meaning that **cp** performed normally, and hence, that **Cross-device** was not checked. It follows that **Cross-device** is not a prerequisite of **AccessibleP**, **FileP**, or **Create-file**. Hence, it must be a prerequisite of **Addlink**.

Even though the position of the **Cross-device** operation has been determined, this does not resolve the question of what it is checking. EG now takes up this problem.

Step 65. The experiment proposer suggests performing several different experiments in an attempt to discover a pattern in the situations in which **Cross-device** is true. These experiments vary conditions that might possibly be relevant to **Addlink**, such as the name and location of the source and destination files in the directory structure, the size of the file, the owner of the file, and so on. These experiments are not shown here.

Step 102. Eventually, TSC (the system-call knowledge source) analyzes the data and discovers that **Cross-device(s,d)** is true if and only if the first names in the canonical file names of **s** and **d** are different. This turns out to correspond to device boundaries on the UNIX system where these experiments are run.

Step 103. TIDS (the data structure knowledge source) attempts to find a change in the data structure theory that can explain why this would be a prerequisite of **Addlink**. One such change involves altering the definition of the mapping between unique file identifiers and files. In the starting theory, this was a one-to-one mapping represented by juxtaposition. If this mapping is extended to map from the ordered pair
 <first-name-of-the-canonical-file-name,
 unique-identifier> to files, then **Addlink** can be defined as modifying only the unique-identifier field, so that two files must share the same first-name-of-the-canonical-file-name in order for the **Addlink** operation to make any sense.

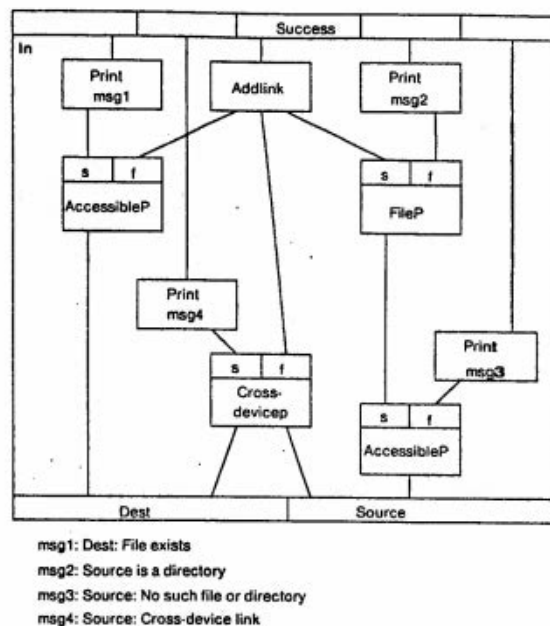


Figure 7: Final program for the **ln** command

Step 104. Given this revision in the definition of **Addlink**, TC revises the program for the **ln** command to include **Cross-device** as a third independent test to check the prerequisites of **Addlink**. Figure 7 shows the final program for the **ln** command.

3 Relationship to scientific theory formation

The hypothetical example shown above exhibits some interesting behavior, and one can easily draw parallels with the way in which people learn operating systems. It is worthwhile to ask, however, if EG can tell us anything about scientific theory formation.

Despite the artificial aspects of the operating system domain, we believe that there are several important similarities between this theory formation task and ordinary science.

- The theory being formed is not a single sentence or equation, but rather a description of the operating system at four distinct levels of analysis.
- Interaction with the operating system takes place at the level of strings of characters, which must be interpreted in terms of the current theory. (Observations are "theory-laden" [Hanson, 1958].)
- Data are gathered indirectly through the design, execution, and interpretation of experiments.
- Experiments go beyond simple hypothesis-test experiments—in which a single YES/NO answer is obtained—to include exploratory experiments, in which some new stimulus causes the operating system to enter an unknown state and then observation commands are issued to learn about that state.

Since EG's task resembles a scientific task, we can now ask what EG's methods tell us about scientific methods. In addition to hypothesis testing, EG demonstrates two other important roles for experimentation in science: data gathering and exploration of the system being studied. EG may also shed some light on the origin of theoretical terms. The methods by which EG revises its theory of data structures and system calls can be viewed as a technique for developing new theoretical terms. EG isolates and uses these terms before giving them a precise definition. This technique is analogous to the method employed by Utgoff [1982]. New terms at one level of description are eventually defined as composites of existing terms at a more primitive level.

4 Summary

The goal of the EG research project is to attack a complex theory formation task in order to see what existing methods can be scaled-up and what new methods need to be developed. The task facing EG is to develop a theory of the UNIX operating system that explains the observable behavior of the system at the executive level. Our hypotheses are that (a) no existing methods will work in this domain and (b) manipulation and observation experiments will be critical to controlling the search for theories in this domain.

The EG program is an agenda-driven system in which only a few current hypotheses are maintained. The agenda is a list of problems that require attention including (a) intermediate results whose consequences need to be propagated to other parts of the theory and (b) holes in the theory that need to be filled. Theory-generation knowledge sources perform type-(a) tasks, and experimentation knowledge sources design experiments to solve type-(b) problems. The theory revision process can require that new data structures and system calls be defined in order to make sense of operating system behavior. Implementation of EG has only been under way for a short time.

5 Acknowledgments

This paper owes much to extensive conversations with Jim Bennett on the generate-and-test paradigm for intelligent systems. Specific comments on an early draft of this document have been provided by Jim. The design and development of EG has benefitted from discussions with Mike Genesereth, Saul Amarel, and Lindley Darden. Jock Mackinlay originally suggested the idea of experimenting with operating systems, and Chuck Paulson helped relate the present work to work on digital circuit testing and diagnosis. This research was supported by an IBM graduate fellowship (to TGD), by ARPA grant no. N00039-83-C-0136, by NIH grant no. RR 00785, and by NLM grant number 5-PO1-1-M03395.

6 References

- Buchanan, B. G., Mitchell, T. M., Smith, R. G., and Johnson, C. R., Jr., "Models of Learning Systems," in J. Belzer, A. G. Holzman, and A. Kent (eds), *Encyclopedia of Computer Science and Technology*, Vol. 11, Marcel Dekker, New York, pp. 24-51, 1977.
- Buchanan, B. G., and Mitchell, T. M., "Model-directed learning of production rules," *Pattern-Directed Inference Systems*, Waterman, D. A. and Hayes-Roth, F. (eds.), Academic Press, New York, 1978.
- Dietterich, T. G. and Buchanan, B. G., "The role of the critic in learning systems," Rep. No. STAN-CS-81-891, Department of Computer Science, Stanford University. December, 1981.
- Dietterich, T. G., London, R., Clarkson, K., and 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. Also available as Rep. No. STAN-CS-82-913, Department of Computer Science, Stanford University, May, 1982.
- Hanson, N. R., *Patterns of Discovery*, Cambridge University Press, Cambridge, 1958.
- Kernighan, B. W., and McIlroy, M. D., "The UNIX Programmer's Manual," Bell Laboratories, 1976.
- Langley, P. W., "Descriptive discovery processes: Experiments in Baconian science," Rep. No. CS-80-121, Computer Science Department, Carnegie-Mellon University, 1980.
- Lenat, D. B., "AM: An artificial intelligence approach to discovery in mathematics as heuristic search," Rep. No. STAN-CS-76-570, Computer Science Department, Stanford University, 1976.
- Lenat, D. B., "Eurisko: A program that learns new heuristics and domain concepts," Rep. No. HPP-82-26, Heuristic Programming Project, Department of Computer Science, Stanford University, 1982.
- Lindsay, R. K., Buchanan, B. G., Feigenbaum, E. A., and Lederberg, J., *Applications of artificial intelligence for organic chemistry: The DENDRAL project*, McGraw-Hill, New York, 1980.
- Lomuto, A. N., and Lomuto, N., *A UNIX* Primer*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1983.
- Mitchell, T. M., "Version spaces: an approach to concept learning," Rep. No. STAN-CS-78-711, Department of Computer Science, Stanford University, 1978.
- Mitchell, T. M., Utgoff, P. E., Banerji, R., "Learning by experimentation: acquiring and modifying problem solving heuristics," Rep. No. LCSR-TR-31, Computer Science Department, Rutgers University, 1982.
- Rich, C., and Shrobe, H. E., "Initial report on a LISP programmer's apprentice," Rep. No. TR-354, AI Laboratory, Massachusetts Institute of Technology, 1976.
- Roth, J. P., "Diagnosis of automata failures: A calculus and a method," *IBM Journal of Research and Development*, pp. 278-281, October, 1966.
- Samuel, A. L., "Some studies in machine learning using the game of checkers. II—Recent progress," *IBM Journal of Research and development*, 11:601-617, 1967.
- Stefik, M. J., "Planning with constraints," Rep. No. 80-784, Computer Science Department, Stanford University, 1980.
- Utgoff, P. E. and Mitchell, T. M., "Acquisition of Appropriate Bias for Inductive Concept Learning," *Proceedings of the Second National Conference on Artificial Intelligence*, Pittsburgh, August, 1982.