Int. J. Man-Machine Studies (1983) **18**, 513–541

Learning search strategies through discrimination

PAT LANGLEY

The Robotics Institute, Carnegie-Mellon University, Pittsburgh, Pennsylvania 15213, U.S.A.

(Received 20 June 1982)

SAGE is an adaptive production system model of strategy learning. The system begins a task with weak, overly general operators and uses these to find a solution to some problem by trial and error. The program then attempts to resolve the problem, using its knowledge of the solution path to determine blame when an error occurs. Once the faulty operator has been found, the system employs a process of discrimination to generate more conservative versions of the rule containing additional conditions. Such variants are strengthened each time they are relearned, until they come to override their precursors. The program continues to learn until it can solve the problem without errors. SAGE has learned useful heuristics in the domains of the slide–jump puzzle, solving simple algebra equations and seriating blocks of different lengths.

1. Introduction

Upon first coming to a problem, a novice can use only weak, general methods, such as heuristic search and means–ends analysis. After considerable experience, he draws on much more powerful, domain-specific techniques for directing his search. Chase & Simon (1974) have reported evidence for such a distinction between expert and novice chess players, while Larkin, McDermott, Simon & Simon (1980) have modeled similar differences between experienced and inexperienced physics problem-solvers. In recent years, researchers in Artificial Intelligence and Cognitive Science have paid considerable attention to expert behavior but have largely ignored the process by which novices become experts as the result of experience. In the following, I present a theory that explains the transition from weak, general methods to domain-specific, powerful strategies.

One goal of science is to develop *general* theories of behavior, and from this viewpoint a major drawback of expert systems is their reliance on domain-specific knowledge. In contrast, a learning theory should be able to explain the shift from novice to expert in a general, domain-independent fashion. Langley & Simon (1981) have proposed a set of basic principles that such a learning theory should incorporate.

Generation of alternatives. The learning system must be able to produce new behaviors; the best way to learn is to make mistakes.

Knowledge of results. The learning system must be able to distinguish between good and bad (or better and worse) performance.

Causal attribution. The learning system must be able to attribute correctly good or bad performance to specific components of the performance system.

Hindsight. The system must be able to modify its behavior based on its knowledge of results and causal attributions.

0020-7373/83/060513+29$03.00/0

Given components that are instantiations of these principles, an AI system should be able to improve its performance as a function of experience; in other words, it should be able to learn. Other forms of learning are possible; for example, one can learn new procedures by being told, or by deducing them from known relations. However, the above principles would appear necessary if one is to learn from *experience*, although the particular incarnations of those principles might differ considerably from system to system.

In this article I describe SAGE, a system that incorporates these principles to improve its behavior as a result of experience. SAGE stands for Strategy Acquisition Governed by Experimentation, and though the system is not intended as a detailed model of human learning, it is intended as a general theory of mechanisms that are *sufficient* for strategy acquisition. The program is presented with a set of *legal* operators for solving some task, and its goal is to discover a revised set of *heuristically useful* operators that will let it solve the problem (and others like it) with little or no search. SAGE has learned useful heuristics in three rather different domains—the slide–jump puzzle, algebra problems in one variable and a length seriation task. However, before moving on to the details of the system and its learning techniques, we should first review some related work on strategy learning.

2. Previous research on strategy learning

Although learning would seem to have considerably intrinsic interest, few researchers have attempted to construct strategy learning systems until quite recently. The delay seems to have been due partly to the lack of a "learnable" representation for strategies. As we shall see, most of the recent systems represent strategies as sets of relatively independent rules, and rule-based formalisms such as production systems are relatively young as AI representations go. Below, I consider five systems that improve their problem solving strategies as a function of experience, and that employ some or all of the basic principles described above. Although excellent work has been done on constructing programs from sample traces, this paradigm seems different enough from the current approach that I will not focus on it here.

2.1. LEARNING ALGEBRA FROM EXAMPLES

Neves (1978) has developed ALEX, a program that learns strategies for solving simple algebra problems. This system is initially presented with examples of legal operators for moving between states, such as adding the same number to both sides of an equation. From these examples, ALEX creates rules for recognizing future examples of those operators. Once it has become familiar with algebraic operators, the system is presented with sample solutions to selected algebra problems. Based on these sample solutions, ALEX constructs rules for solving similar problems in different contexts. The first step in this process is to identify which operator is responsible for each transition, and the operator recognizing rules prove indispensable in this task. In some cases, no single operator is sufficient to account for a transition, and ALEX is forced to employ means–ends analysis to find a sequence of known operators to account for the change. In the extreme case, this method can be used to find a solution to the entire algebra problem; the resulting trace can be used in the same manner as one provided by a tutor.

ALEX is stated as an *adaptive production system*. That is, its initial performance component is composed of a set of condition–action rules called *productions*, and learning consists of the addition of new rules that result in new behaviors. However, before ALEX can construct rules for applying the operators found in the sample sequence, it first has to discover the *conditions* under which those operators should be applied. To this end, Neves employs two simple heuristics:

> the symbols affected by an operator probably play a role in the condition;
> symbols involved in more severe changes should be preferred as conditions; for example, removal of a symbol is more severe than its transformation.

This learning method is sufficient to find the correct conditions on a number of algebra operators. In addition, Neves constructed a similar program (Neves & Anderson, 1981) that learned list processing algorithms (such as reversing a list) through an analogous method.

Thus, Neves' system incorporates two of the four learning principles discussed above. ALEX has the ability to generate behaviors of its own, as well as to accept sample sequences from a tutor. The program is certainly able to modify its behavior on the basis of past experience, so we can conclude that it possesses hindsight. However, since the system assumes that all rules it creates are correct at the outset, it possesses no mechanisms for achieving knowledge of results or assigning credit. In other words, ALEX provides no means for recovering from an error if an incorrect rule is learned, and this must be considered a serious shortcoming of the program.

2.2. IMPROVING STRATEGIES ON THE TOWER OF HANOI

Anzai (1978*a*, *b*) has reported on a program that improves its strategies for solving the Tower of Hanoi puzzle. The system begins by solving the problem with a depth-first search strategy, retaining information about previous states it has encountered and previous moves it has made. As with ALEX, Anzai's program is stated as an adaptive production system; various learning heuristics look for patterns in the trace information, and create new condition–action rules when these patterns occur. Once these rules have been constructed, they can be used to help direct the search process in a more intelligent manner. One of the most interesting features of Anzai's system is that certain of its learning rules cannot be used at the outset; simpler forms of learning must occur before these more sophisticated techniques may come into play, so that the system evolves through a number of distinct stages.

The first of these simple heuristics notes when an action leads to a state that has occurred before. A closely related rule notes cases in which two separate operator sequences (one longer than the other) lead to the same state. In both of these situations, the system constructs a rule to *avoid* applying the responsible operators in the future. In the first case, the undesirable operator is the one leading back to the previous state; in the second case, it is the last operator applied in the longer of the two paths. The conditions placed on these rules include information about the states and operators involved in the noted paths; also, the system knows enough about the Tower of Hanoi task to generalize across the disks involved in the patterns.

The rules learned in this fashion can only suggest moves that should be avoided, but they are enough to direct the search toward an initial solution. After a solution path has been found, more powerful learning heuristics can be applied. For example,

Anzai's system begins with only partial descriptions of the final goal state, but once the exact solution is known, the system constructs a more complete description that can aid the search process in future attempts. Another heuristic leads to the creation of rules for asserting subgoals. This applies when the system has been unable to satisfy a goal at some state ($s1$) because the necessary action ($a1$) violates some constraint of the problem, and later, another action ($a2$) leads to a new state ($s2$) in which that constraint is no longer violated. The result is a new rule that suggests $s2$ as an appropriate subgoal whenever $s1$ is desired. Such a rule is more powerful than the avoidance rules, since it leads to positive suggestions in place of negative ones.

In summary, Anzai's system begins by generating a search tree for the Tower of Hanoi task. Along the way, it learns rules for pruning the tree by avoiding undesirable moves. Once it has reached the solution to the puzzle, it learns rules that establish subgoals, enabling it to generate *only* those portions of the search tree that lie on the solution path. Thus, one can see that the program incorporates all four of the components discussed earlier, since it generates alternatives, determines whether behavior is desirable, assigns credit to particular operators and constructs rules to avoid or prefer these operators in the future.

However, Anzai's model of strategy acquisition contains some undesirable features as well. First, the system employs considerable knowledge of its task domain to aid the learning process. Anzai & Simon (1979) have discussed the general principles behind the model, but this is no substitute for a generalized program. Anzai (1978*c*) has shown that very similar techniques can be used to learn seriation strategies, but again this was a separate program from the Tower of Hanoi learner. Second, Anzai (like Neves) assumes that all learned rules are correct, and that no need for error recovery will ever arise. This assumption could lead to serious problems if an incorrect avoidance rule were constructed, since it might tell the system to avoid a move necessary to solving the problem.

2.3. BRAZDIL'S ELM

Brazdil (1978) has described ELM, a PROLOG program that learns heuristics for solving problems in the domain of arithmetic and simple algebra. The system is provided with a set of legal operators for solving such problems, but does not initially know in which order to apply them. When ELM is presented with a sample solution, it attempts to replicate the solution path by trying all applicable operators. Since only one operator agrees with the solution trace, that rule is given priority over its competitors. In the future, it will be preferred to other operators that may also be applicable. The result is a partial ordering among operators that ELM can use to direct its search in profitable directions.

Occasionally, the system encounters a problem that indicates that operator A should be preferred to operator B, while another problem suggests exactly the opposite. In such cases, ELM searches for some predicate that was true when A should have been applied, but that was false when B was the correct move. Upon discovering such a predicate, the program creates a variant rule A′ that includes the predicate as an additional condition. The same process is applied to B, resulting in a variant on this operator as well, say B′. Both A′ and B′ are placed above A and B in the partial ordering, so that if either of the more specific rules is true, it will be preferred to the

original versions of the operator. This approach has much in common with the approach to strategy learning taken in the current paper.

From this description, one sees that Brazdil's system can determine when its performance has improved, that it assigns credit and blame to responsible operators, and that it modifies its behavior based on this knowledge. However, ELM must rely on a benevolent tutor to provide sample solutions, since it does not generate new behaviors on its own. The method for generating a more specific variant on an operator does provide some means for error recovery, so that if an incorrect ordering is established, this can effectively be corrected at a later date. Still, there is no provision to handle cases in which spurious conditions are added to an operator, except to construct ever more specific versions and to give them priority.

2.4. IMPROVING HEURISTICS FOR SYMBOLIC INTEGRATION

Mitchell, Utgoff, Nudel & Banerji (1981) and Mitchell, Utgoff & Banerji (1982) have described LEX, a computer program that discovers heuristically useful conditions on operators for solving problems in symbolic integration. Like Brazdil's (1978) system, LEX begins with legal operators for searching a space, but does not know how to apply these rules in an intelligent fashion. The system begins by solving integration problems with a form of best-first search and, once it has found a solution path, it uses this path to identify good and bad instances of the operators that were applicable along the way. For example, if the application of an operator led to a state on the solution path, this would be considered a good instance. However, if some other operator could have been applied at the same point, this portion of the tree is expanded in an attempt to find another route to the answer; if no alternative path can be found with reasonable effort, the initial move in this branch would be treated as a bad instance of the alternative operator.

Once it has identified good and bad instances of an operator, LEX employs Mitchell's (1977) version space technique to identify the heuristically useful conditions on its application. New good instances lead the system to determine that certain potentially relevant conditions are actually unnecessary, while bad instances tell it that other conditions must be true for the operator to be selected. However, even before the exact form of the heuristic is determined, LEX can use its partial description to decide on the relevance of this operator compared to others; this allows the program to improve its performance steadily as it is mastering the correct form of the various heuristics.

In LEX, we see a strategy learning system that incorporates all four of the principles that we discussed at the outset. The system generates its own behavior in the form of solution paths to sample problems, and used this information to distinguish good behavior from bad behavior, and to assign credit and blame to responsible operators. From good and bad instances of these operators, the program carefully determines the conditions under which the operators can be profitably applied. The system of Mitchell *et al.* appears to be a robust strategy learner with considerable potential and, although the researchers have tested the system only in the domain of integration, it seems to be implemented in a general fashion. One of LEX's drawbacks is its reliance on an "is-a" hierarchy among function types (e.g. sine is a trigonometric function) that must be given to the system by its authors; however, this appears to be a limitation of the current implementation and does not, in principle, seem necessary. Another

limitation is that the version space technique, like the generalization-based methods to which it is related, has difficulty in learning rules with disjunctive conditions; although such heuristics may not appear in the domain of symbolic integration, they do in other tasks (such as the slide–jump puzzle).

2.5. FORMULATING MORE EFFICIENT STRATEGIES

Neches (1981) has described HPM, a program that models the transformation of well-specified but inefficient strategies into more efficient versions. Like the other strategy learning systems described above, HPM learns from trace information about the operators that have been applied and the states that have resulted. However, it addresses a somewhat different issue from the other programs we have discussed. Rather than starting with little knowledge of a task and learning to direct search through the problem space, Neches' program starts with a well-understood algorithm for performing a task and revises this algorithm as it detects short-cuts that will allow the same result to be achieved more efficiently.

As with Neves' and Anzai's programs, HPM is stated as an adaptive production system, and Neches' learning heuristics are stated as condition–action rules. These rules match off patterns in the trace information that is left when performance rules carry out the desired algorithm. Neches has proposed a number of basic transformation types that can lead to more efficient procedures.

Reduction to results: converting a computational process to a memory retrieval process.

Reduction to a rule: replacing a procedure with an induced rule for generating its results.

Replacement with another method: substituting an equivalent procedure for a more expensive one.

Unit building: grouping operations into a set that can be accessed as a single unit.

Deletion of unnecessary parts: eliminating redundant or extraneous operations.

Saving partial results: retaining intermediate results which would otherwise have to be recomputed later in a procedure.

Re-ordering: changing the sequence in which operations are performed.

These transformation types correspond to the actions of various adaptive productions for altering the make-up of the performance system, but one can easily imagine the conditions under which these rules would apply. For example, if two different tests are applied to decide whether some action should be taken, and one test always gives the same result as the other, then remove one of the tests (probably the more expensive one) from the procedure, since it is an unnecessary component.

Neches has used HPM to model human learning on a sequence generation task, as well as a task involving the drawing of pictures with a computer graphics program. The system has also successfully modeled the evolution of children's addition strategies. In some schools, children are initially taught an addition algorithm called the "min strategy", not because it is efficient but because it is easy to learn. However, after using this strategy for some months, many children show evidence of spontaneously transforming it into a more efficient algorithm. HPM not only accounts for the overall transformation between these two strategies, but for intermediate strategies that have been detected along the way.

It is difficult to evaluate Neches' system according to the principles mentioned above. Certainly, HPM generates its own behavior, but this is well-defined behavior that was programmed into the system at the outset. In a sense, the program achieves knowledge of results and assigns credit, but its measure of goodness is different from our other examples, focusing on computation time rather than which branch of a tree to search. And certainly the system possesses hindsight, since it modifies its behavior based on the above information. Thus, HPM incorporates all four of the learning principles, though it does so in rather unusual ways. However, this appears to be more a limitation of the framework I have proposed than of Neches' system. It results from the fact that Neches has focused on learning efficient strategies from well-specified tasks, while most other work in the area has focused on learning well-specified strategies from ill-defined ones that initially involve search.

2.6. COMMENTS

In conclusion, we see that in recent years, considerable interest has arisen with respect to strategy learning, and that a number of systems have been developed that attempt to model this process. Each of these systems employs some of the principles proposed in the Introduction, and those which incorporate all of the principles appear to be the most robust and versatile. In addition, all five of the systems represent operators as separate rules that can be reasoned about independently. This appears to be an important feature, for it means that the learning process can be factored into manageable components. Also, each of the systems employs some form of operator traces, both for assigning credit and for learning the heuristically useful conditions on these operators. Both the independence of operators and the presence of trace information appear to be generally useful features that any strategy learning system should find profitable.†

Although the systems just described have told us a number of important facts about the acquisition of problem-solving strategies, more work needs to be carried out. In particular, none of the existing systems has yet shown that it can learn in more than a single domain. Since one of the major goals of science is generality, an acceptable theory of strategy learning must prove its domain-independence by learning on a number of different tasks. In the following sections, I describe SAGE, a system which meets this more stringent test. The program borrows some important ideas from the earlier systems, including the notion of waiting until a complete solution path has been found before attempting to learn. However, the system also introduces some new methods that expand the class of heuristics that can be learned, as well as making the learning process more gradual and more robust.

3. An example: the slide–jump puzzle

SAGE's behavior on the *slide–jump* puzzle provides a useful introduction to its performance and learning characteristics. In this puzzle, one starts with equal numbers of two types of coins (say quarters and nickels) set in a row. All quarters are on the left, all nickels are on the right, and the two sets are separated by a blank space. The

† This is perhaps too strong a statement. For a radically different approach to strategy learning that is organized around the discovery of useful evaluation functions, and which does not rely on these two features, see Rendell's (1982*a*, *b*) descriptions of PLS1.

goal is to interchange the positions of the quarters and nickels. However, quarters can move only to the right, while nickels can move only to the left. Two basic moves are allowed: a coin can *slide* from its current position into an adjacent blank space, or it can *jump* over a coin of the opposite type into a blank space. A coin may not jump over another coin of the same type. Table 1 presents one of the two (symmetric) solution paths for the four-coin problem. Although there are only 31 states in the problem space associated with this particular puzzle, humans have considerable difficulty in reaching a solution.

TABLE 1
Solution path for the four-coin slide–jump puzzle

QQ—NN	Initial state
Q—QNN	Slide a quarter from 2 to 3
QNQ—N	Jump a nickel from 4 to 2
QNQN—	Slide a nickel from 5 to 4
QN—NQ	Jump a quarter from 3 to 5
—NQNQ	Jump a quarter from 1 to 3
N—QNQ	Slide a nickel from 2 to 1
NNQ—Q	Jump a nickel from 4 to 2
NN—QQ	Slide a quarter from 3 to 4

SAGE starts this task with one condition–action rule for sliding and one for jumping, as well as some additional rules which support the search for a solution. The initial operators are correct in that they propose only *legal* moves, but they lack conditions for distinguishing *good* moves from *bad* moves. As a result, the program makes many poor moves on its first pass, and is forced to back up whenever it reaches a dead end. The goal of learning is to discover heuristically useful conditions on the operators that will let the system solve future problems without any search.

Once SAGE has solved the problem with its depth-first search strategy, it attempts to find a second solution, but this time it has the initial solution to guide its search. When one of the operators is incorrectly applied, the system knows its mistake immediately. If such an error occurs, SAGE compares the last correct application of the offending operator to the current incorrect application. Upon finding a difference between the two situations, the system adds a new rule to its repertoire; this rule is identical to the original operator, except that it includes the difference as an additional condition that will keep it from applying in the undesired situation. The program continues to learn in this fashion, constructing more conservative rules when errors are made and strengthening rules when they are relearned, until it traverses the entire solution path with no mistakes.

The reader may wonder why the program should bother to improve its operators once it has the solution to the puzzle in mind. The reason is simple: there is a chance that these revised operators will also be useful in related problems for which the solution path is *not* available. To the extent that the learned heuristics transfer across different but related problems, they can be very useful in reducing search. As we shall see in more detail later, the operators SAGE learns on the four-coin puzzle let it solve the six-coin puzzle immediately, without backtracking.

4. The performance system

A learning system can only exist in the context of some performance framework, and in this section I discuss the performance system possessed by SAGE at the outset of its explorations. The performance components for each task differ because they require different operators; however, all of the initial programs have some important features in common, and I will focus on these similarities. First I discuss the production system language in which SAGE is implemented. Following this, I consider the representation of problem states and the operators which alter those problem states. Finally, I examine SAGE's mechanisms for backing up and for recognizing the solution of a problem.

4.1. THE PRISM LANGUAGE

SAGE is implemented in PRISM, a programming language designed for exploring the space of production system architectures, especially those involving learning phenomena. This language is discussed in greater detail by Langley & Neches (1981). Like other production system formalisms, PRISM represents procedural knowledge as condition–action rules called *productions*. A production system operates in *cycles*; each cycle, every rule is matched against the current state of *working memory*. From among those rules which match the current memory, one or more are selected and their actions are applied. These actions alter the state of working memory (or, in the case of learning, the state of production memory), making a new set of productions true. The system cycles in this fashion until no rules are true or until a stop command is encountered.

PRISM differs from most production system languages in the mechanisms it provides for learning. These include techniques for designating new rules, discriminating faulty ones, strengthening productions on the basis of use or relearning, and generalizing from rules with identical forms. In this article I will focus on the discrimination and strengthening processes, which are discussed in more detail in the following section. PRISM also has the ability to store declarative knowledge in a long-term propositional network, and provides over 30 user-modifiable parameters for altering the system's behavior. The language has many features in common with OPS4 (Forgy, 1979) and ACTF (Anderson, Kline & Beasley, 1980) formalisms, which have strongly influenced the development of PRISM, but it has considerably more flexibility than either of these earlier systems.

4.2. REPRESENTING PROBLEM STATES AND OPERATORS

A problem solver must have some *representation* to work upon. He starts with an *initial* state and gradually transforms this into the *goal* state. Although SAGE necessarily uses different representations for the different tasks it attempts, they all have one major feature in common: each problem state is represented as a number of distinct elements in working memory. Since operators affect only some of these elements, the entire problem state need not be respecified each time a move is made. Instead, those elements which have become true are added to memory, while those which are no longer true are removed. Also, this composite representation allows partial information about the problem state to be included as conditions on an operator to the exclusion of less-relevant information.

SAGE operators are stated as PRISM productions. When the conditions of one of these productions is met, then it can be legally applied; in such cases, the rule inserts a goal to remove some elements and add others. For example, the basic *slide* production† can be stated:

If a *coin* is in *position1*,
  and *position2* is to the *direction* of *position1*,
  and *position2* is blank,
  and that *coin* can move to the *direction*,
then consider sliding that *coin* from *position1* to *position2*.

This operator will be true whenever a coin is next to the blank space and that coin is allowed to move in that direction; it will work for either coin and for any pair of adjacent positions. Once a goal has been set, another production is responsible for updating the problem state by actually adding and deleting elements from memory. This division of labor simplifies matters in later runs when the solution path is known, for an incorrect goal may be caught before it is implemented. Note that although the above rule proposes *legal* moves, there is no guarantee that these will be *good* moves. As stated, the slide operator will generate many bad moves, such as sliding two nickels in a row, which leads to a dead-end state. SAGE's learning mechanisms must generate more conservative and powerful operators before useful actions are consistently generated.

4.3. CONTROLLING THE SEARCH

Before SAGE can learn from its mistakes, it must be able to identify those mistakes. This means it must know the correct solution path to a given problem. And to find a solution using the very weak operators it has at the outset, the system must be able to search. In addition to operators for moving through the problem space, effective search requires the ability eventually to recognize a fruitless path, and the ability to backtrack once such a path is recognized. SAGE carries out a form of *depth-first* search in each of the tasks it attempts, preferring those operators with the highest *strengths*. As we have seen, learning consists of constructing more powerful operators that direct that search down fruitful paths; however, before these variants can actually improve the system's behavior, they must become stronger than the more general rules they are attempting to replace.

When SAGE decides it has reached a bad state or dead-end, it backtracks to the previous state. This is possible only because the system keeps a trace of all previous moves made along the current path. This trace serves a second purpose in that it provides a record of the solution path once the goal has been reached. During the initial search, SAGE also remembers bad moves it has made along the current path, so that it can avoid making these moves a second time. However, this negative information is removed once the solution is found, since the system must be allowed to retake these false steps in later runs if it is to learn from its mistakes.

In addition to searching, a problem solving system must be able to determine when it has reached a solution. SAGE employs a single production for recognizing such

† For the sake of clarity, I will present only English paraphrases of the actual productions; in these paraphrases, an italicized term stands for a variable which will match against any symbol. In this case, the variable *coin* will match against the symbol *quarter* or *nickel*, rather than against a particular coin.

states. The rule for slide–jump may be paraphrased:

If a *coin1* moves to the *direction1*,
  and a *coin2* moves to the *direction2*,
  and *position* is blank,
  and no *coin1* is to the *direction2* of *position*,
  and no *coin2* is to the *direction1* of *position*,
then you are finished.

The goal-recognizing rules for algebra and seriation are necessarily different, but both are stated as single productions in much the same spirit as the above rule. If the system has managed to solve the problem without needing to backtrack (i.e. with no search), then SAGE infers that no learning is required and the program halts. In the following section, I consider the system's response to cases in which errors have been made, and the manner in which it uses the correct path to improve its performance.

5. Learning improved operators

Once SAGE has solved a task by trial and error, it moves into learning mode. At the outset of this stage, the system uses its initial operators to generate alternatives just as it did the first time around. Since the solution path is known, incorrect goals are spotted as soon as they are proposed; this means that SAGE never strays far from the correct path, and backtracking is unnecessary. These errors lead to calls on the PRISM discrimination mechanism, which creates variants of the operators responsible for the mistakes. As these variants are produced, they compete for attention with the original versions. The program continues to create new rules and to strengthen existing ones until useful variants mask the original operators and the solution is reached flawlessly. Below, I discuss how SAGE achieves knowledge of results and assigns credit, and how the system alters its behavior based on this information.

5.1. ASSIGNING CREDIT AND BLAME

During the learning phase, SAGE stores the instantiations of productions that are responsible for adding elements to memory. After the program proposes a move in learning mode, it compares this move to the analogous one in the known solution path. If the two are identical, then the proposal is carried out and the program continues onward to the next problem state. In addition, the following rule is applied:

If you have a goal to make *move1* at *depth1*,
  and *move1* lies on the solution path at *depth1*,
then carry out the goal to make the move,
  and retrieve the instantiation that proposed this goal,
  marking it as a positive instance of the responsible production.

In such cases, no learning is required since the correct operator has applied in the correct manner. However, this rule stores the fact that the responsible instantiation is a good instance of the production, since this information is essential for recovering from errors when they do occur.

As long as the proposed moves agree with those along the known solution path, SAGE continues to make these moves and is satisfied with its progress. If instead the

proposed and correct moves differ, a slightly different production is applied:

If you have a goal to make *move1* at *depth1*,
  and *move1* does not lie on the solution path at *depth1*,
then abandon the goal to make the move,
  and retrieve the instantiation that proposed this goal;
  mark this instantiation as a negative instance,
  and weaken and discriminate the responsible production.

This rule forces the proposed move to be abandoned, since it will lead the system away from the known route to the solution.† It retrieves the instantiation that proposed the move, and weakens the responsible production. In addition, it calls on the discrimination mechanism to generate more conservative variants of this production, which I discuss in more detail below.

The process by which SAGE assigns credit and blame to responsible rules is somewhat subtle, and deserves some discussion. In some learning systems, the assignment of credit is difficult, because many steps may be taken before an error is detected, and this means that any of these steps may have been the cause of the error. However, because SAGE has the entire solution path available for inspection, the system can check on the correctness of each step independently. As a result, faulty moves are detected immediately, and because all previous moves are guaranteed to be correct (since they were checked as well), the most recently applied operator must be responsible for the error. Thus the system never strays more than a single move from the solution path, and its learning process is greatly simplified. Sleeman, Langley & Mitchell (1982) present a fuller discussion of this approach to credit assignment based on solution paths.

5.2. THE DISCRIMINATION PROCESS

Once SAGE has identified the production responsible for an error, it calls on the *discrimination* process. The goal of discrimination is to formulate more conservative versions of the production with additional conditions that will prevent their application in the undesired case. Brazdil (1978) and Anderson *et al.* (1980) have explored very similar learning strategies. Note that the discrimination learning approach contrasts sharply with more traditional AI learning methods, such as those of Winston (1970), Hayes-Roth & McDermott (1976) and Vere (1975). These researchers have concentrated on the inverse problem of *generalization*, in which one starts with very specific rules and removes restrictions as more examples are gathered. Although generalization-based approaches are quite useful when a benevolent tutor is available to present carefully designed instances, I personally suspect that discrimination-based approaches will fare much better when the system must learn on its own initiative. I have presented the arguments for the superiority of discrimination over generalization elsewhere (Langley, 1983). Bundy & Silver (1982) have also examined the similarities and differences between these two approaches to learning.

The discrimination mechanism employed by SAGE retrieves the most recent good instantiation of the faulty rule, and compares it to the current bad instantiation with

† This assumption holds only if a *single* optimal solution path exists; this results in some complications on the slide–jump task (which has two isomorphic solutions) that I shall discuss later.

the goal of finding some *differences* between them. SAGE's discrimination mechanism first retrieves all elements that were in memory during the good application, as well as those that were in memory during the errorful application. Together with the good instantiation, the first set of elements make up what Bundy & Silver have called the *selection context*; the bad instantiation, combined with the elements in memory at that time, is called the *rejection context.* Given these two sets, SAGE searches for elements that were present in the selection context, but had no *analogous* elements in the rejection context. Upon finding such a difference, the system creates a variant of the production that led to the error. The new rule is identical to the old, except that it includes the good element (with some constants replaced by variables) as an additional condition. This condition ensures that the new production would still be true when the original rule correctly applied, but would not be true in the incorrect situation.

The discrimination process also searches for elements that were present in the rejection context, but which had no analogs in the selection context. When such an element is found, it is included as a *negated* condition in a new variant rule. As before, the new condition ensures that the production would have been true during the correct application but not during the faulty one. In addition to searching for single elements that were present in one context but not the other, SAGE also searches for *conjunctions* of elements that differed between the two situations. In fact, the system cannot discover arbitrary conjunctions, but only *chains* of elements (where the chaining occurs through symbols shared by a pair of elements). This gives the discrimination process two additional abilities. If a chain of elements occurred during the selection context, then a variant is constructed with a number of new conditions (one condition for each element in the chain). If the chain occurred during the rejection context, a variant is created with the elements contained in a *negated conjunction*; such a rule may apply if any one of its negated conditions is present in memory, but not if all are present simultaneously. SAGE considers chains up to four elements in length, and constructs a production for every difference that it discovers. This ability to discover conjunctions of differences is one of the main distinctions between SAGE's discrimination process and those employed by Brazdil (1978) and by Anderson *et al.* (1980).

Some additional details of the discrimination process are worth mentioning. First, the system is given information about which symbols should be chained through in searching for conjunctions. For example, since the symbol *goal* occurs in a large number of elements (connecting them to each other), it would make little sense for SAGE to chain through this symbol. Similarly, the system is told which symbols it should treat as significant when searching for differences. Thus, in the slide–jump task, SAGE is told that the exact type of coin it has moved should be ignored (though the relations between these types on different moves is not ignored). When the variant rule is constructed, significant symbols are retained in the new conditions, while irrelevant symbols are replaced by variables. Finally, to reduce the number of elements that the discrimination process must consider when searching for differences, only some of the potentially relevant elements are examined. For example, in the slide–jump puzzle, SAGE examined the moves that have been made previously, but not the configuration of coins on the current or previous moves. In summary, although the discrimination heuristic appears to have considerable generality, it requires certain domain-specific knowledge to make it effective. I will consider some more concrete examples of this learning method shortly.

5.3. THE STRENGTHENING PROCESS

Each production in the system has an associated *strength*, which plays a major role in deciding whether it will be applied. On each cycle, SAGE selects from the set of true productions the one with the greatest strength. (If multiple instantiations of the same production are true, or if two or more rules have identical strengths, one is selected randomly.) Each of the original operators is given a high initial strength, so that when a variant rule is first created, it will be weak compared with the production from which it was spawned. However, productions are automatically strengthened whenever they are relearned. Thus, if the same discrimination occurs a number of times, the resulting rule will eventually come to mask its predecessor.

The strengthening process plays an important role in directing SAGE's search through the space of possible rules. Unlike some earlier learning systems, SAGE does not rely on a benevolent tutor to present positive and negative instances in a carefully prepared order. As a result, many differences may exist between a given good and bad instance of an operator, and a new variant must be constructed for each of these differences. Since some of these productions may be incorrect, it is important that they do not have a permanent effect on the system's behavior.† By strengthening rules when they are recreated, SAGE's behavior is influenced only by variants that have proven useful in a number of situations. Thus, strength is best viewed as a measure of the *success rate* of a variant production, with preference being given to the most successful versions. In addition to directing search through the space of rules, the strengthening process has also been useful in learning in the presence of noise and incomplete representations (Langley, 1983).

6. SAGE at work

Although I have considered SAGE's learning mechanisms in the abstract, I have not yet described their effects on particular tasks. Below I trace the system's evolution in the domains of slide–jump, algebra and seriation. Definite similarities exist among the three domains, but significant differences occur as well. In each case, I consider the program's initial operators, the variants of these operators it learns along the way, and some statistical measures of the system's improvement with experience.

6.1. LEARNING SLIDE–JUMP OPERATORS

In previous sections, I have considered SAGE's representation for states of the slide–jump puzzle, along with its initial operators for sliding and jumping.‡ In addition, I have discussed rules for implementing the goals proposed by these operators, for backtracking and for recognizing the solution state. SAGE also includes six productions which are domain-independent; these initialize the problem state when the system

† One might suggest that rules leading to errors be removed entirely from production memory, and this would do away with the need for a strengthening strategy. However, some tasks require that the original rules be retained at a low preference level. For example, in the second half of the slide–jump task, the original slide operator must be used to suggest moves, since none of the more powerful variants are applicable.

‡ In the run described here, the initial slide operator was given greater strength than the jump operator. If these had started with equal strengths, the system would have quickly come to prefer the jump rule (productions are weakened when they lead to errors), and since this takes one a long way toward solving the problem, much less interesting learning would have occurred. Because of this feature of the task, SAGE concerns itself only with generating more conservative versions of the slide operator.

starts over on a problem, recognize when a proposed move differs from the correct one (and evokes the discrimination process) and recognize when the correct move has been suggested. The last two productions were described at some length in the previous section. These six rules form the central core that is shared by the three incarnations of the system described in this paper.

SAGE was initially presented with the four-coin slide–jump task. After finding the solution path by depth-first search, the system attempted to solve the problem a second time.† When the correct first move (sliding a quarter from 2 to 3) was followed by an incorrect proposal (sliding a quarter from 1 to 2), the discrimination routine was called. This created two variants of the slide operator:

If a *coin* is in *position1*,
  and *position2* is to the *direction* of *position1*,
  and *position2* is blank,
  and that *coin* can move to the *direction*,
  and you have not made any previous moves,
then consider sliding that *coin* from *position1* to *position2*.

and

If a *coin* is in *position1*,
  and *position2* is to the *direction* of *position1*,
  and *position2* is blank,
  and that *coin* can move to the *direction*,
  and you did not just slide a *coin* from *position2* to *position3*,
then consider sliding that *coin* from *position1* to *position2*.

The first of these rules will only apply on the first move of every task, and includes nothing in its conditions to direct the search. The second variant is more selective; it would not have proposed moving a coin from position 1 to 2, provided that coin had just been moved from 2 to 3. In general, it would not slide a coin *into* a position which had just been vacated by another slide move.

After backing up, SAGE suggested jumping a nickel from 4 to 2; this agreed with its previous experience, so the suggestion was carried out. However, another mistake was made on the third move, when the system considered sliding a quarter from 3 to 4. This time discrimination reproduced the two variants shown above, causing them to be strengthened; in addition, a new production was constructed:

If a *coin* is in *position1*,
  and *position2* is to the *direction* of *position1*,
  and *position2* is blank,
  and that *coin* can move to the *direction*,
  and you did not just jump the other coin from *position2* to *position3*,
then consider sliding that *coin* from *position1* to *position2*.

† The slide–jump task has two optimal solution paths which are "mirror images" of each other; the particular path is determined entirely by the initial move. To ensure that SAGE makes the same first move on every run, a special production was included that increased the activation of part of the problem description. This was sufficient to focus the system's attention on the relevant coin and to avoid the effort of trying to distinguish between two equally good moves, but this approach lacks elegance, and I am actively searching for a better way of handling the difficulty.

This rule states that one should not slide a coin *into* a position from which one has just jumped the other brand of coin. Note that this production may still propose those moves avoided by second variant, while the second variant may still propose this type of sliding move. In their current forms, both rules are overly general.

At this point SAGE correctly slid a nickel from 5 to 4, giving it a new selection context for the slide production to consider in subsequent discriminations. Next the system jumped a quarter from 3 to 5, followed by a proposal to slide a nickel from 4 to 3. This error led to the creation of a fourth production:

If a *coin* is in *position1*,
  and *position2* is to the *direction* of *position1*,
  and *position2* is blank,
  and that *coin* can move to the *direction*,
  and you have just jumped a *coin* from *position2* to *position3*,
then consider sliding that *coin* from position*1* to *position2*.

In this case, a positive condition was included in the variant, suggesting that slides should occur after a jumping spree has been completed, and moving in the same direction. In addition, discrimination produced four other less-useful variants which made reference to moves earlier in the problem. By now SAGE had reached the halfway point for the problem. Since only one move was possible† at each step from this point onward, the program finished with no more mistakes. However, earlier errors had been made, so the system attempted to solve the four-coin problem a third time. In this run, identical mistakes occurred, and each of the variants was strengthened.

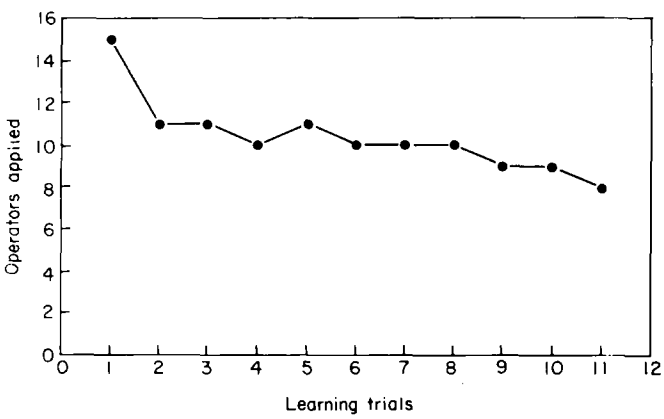

FIG. 1. Learning curve for the slide–jump task.

The program continued along these lines, until during the fifth run the second variant came to surpass the original rule in strength. After this, the more conservative production was applied whenever possible. When this rule led to the same error as

† Note that the third and fourth variants are *not* true on the slides required for the last half of the problem. For this reason, it is essential that the original slide rule remain available, and that the variants simply come to be preferred when competition occurs.

before on the third move, discrimination resulted in yet another version of the operator:

If a *coin* is in *position1*,
  and *position2* is to the *direction* of *position1*,
  and *position2* is blank,
  and that *coin* can move to the *direction*,
  and you did not just slide a *coin* from position*2* to *position3*,
  and you did not just jump the other coin from *position2* to *position3*,
then consider sliding that *coin* from *position1 to position2*.

This rule includes the negated conditions of both the second and third variants, stating that one should not propose a slide if *either* condition is met. Other variants† were created as well, but never gained sufficient strength to have any effect on the system's behavior. After five more runs, the above rule acquired more strength than its precursor, and on its eleventh run, SAGE reached the goal state without error. In addition, the system successfully applied its new knowledge to the six-coin task, finding a solution on its first attempt with no backtracking.

Figure 1 presents SAGE's learning curve on the four-coin slide–jump task. The graph plots the number of operators that were applied before solution against the order in which each run occurred. A comparison of the second to eleventh runs shows the improvement when information about the solution path was available, while a comparison of the first and eleventh attempts shows the speed-up when this information was absent. As the figure indicates, the addition of more powerful operators nearly cut the system's search in half.

6.2. LEARNING ALGEBRA OPERATORS

For its second learning task, SAGE was presented with algebra problems involving a single variable. The program was given a tree structure representing the initial problem state, and the goal was to arrive at a tree with only two branches—one pointing to the variable and the other to a number. The system's initial operator was like the slide operator, in that it set up goals to modify only part of a problem state. As in slide–jump, the basic operator was quite general in its initial form. It may be stated:

If you see a *number* as the argument of *function1*,
  and *function2* is a function,
then apply *function2* to both sides with *number* as its argument.

SAGE was told about the four standard arithmetic functions—addition, subtraction, multiplication and division. Given the initial state $3x - 1 = 5$, this production might try adding 3, 1, or 5 to both sides, subtracting 3, 1, or 5 from both sides, and so on. However, only one of these (adding 1) will lead to a reduction in the complexity of the expression, and the program's goal was to discover the appropriate conditions for distinguishing between this move and others.

In addition to its basic operator, the algebraic SAGE was given productions for carrying out goals once they have been set. These included rules for adding new

† Altogether, SAGE generated some 18 variants on the initial operator, but only four of these can be considered useful; fortunately, the strategy of giving variants low initial strengths and strengthening upon recreation was sufficient to focus attention on the more promising rules.

branches to the tree, for combining terms when possible, and for dropping identity elements. (The system did not learn with respect to these rules.) SAGE decided to backtrack if a goal proposed by one of the operators had failed† to reduce the expression's complexity. In such cases, the inverse operator was applied and the previous state was regained.

TABLE 2
Solution path for SAGE's first algebra problem

$3x - 1 = 5$	Initial state
$3x = 6$	Add 1 to both sides
$x = 2$	Divide both sides by 3

Table 2 presents the solution path for $3x - 1 = 5$, the first algebra problem given to SAGE. After completing the task by trial and error, the program attempted to solve the problem a second time in learning mode. After a number of false starts, the system eventually tried adding 1 to both sides, giving it a correct instantiation of the operator on which to base its learning. When it incorrectly tried to subtract 3 on the next move, the discrimination mechanism produced two variant rules. One of these required that the function to be applied to both sides be present at the level of the tree being examined; because of the problem representation, this was true half of the time (when addition and multiplication were the relevant operators), but was not very useful for directing search. However, the second variant was more promising:

If you see a *number* as the argument of *function1*,
  and *function2* is a function,
  and *function1* is the inverse of *function2*,
then apply *function2* to both sides with *number* as its argument.

This rule states that one should apply the *inverse* of the function to both sides of the equation; for example, if division by 2 occurred on one side, this production would propose multiplying through by 2. Although incomplete, this variant is certainly better than the original.

For its next move, the system considered dividing by 6. This error led to the reconstruction and strengthening of the variant shown above, but it also generated another useful version:

If you see a *number* as the argument of *function1*,
  and *function2* is a function,
  and *function1* occurs at the top level,
then apply *function2* to both sides with *number* as its argument.

This production restricts its attention to non-embedded terms. For example, given the state $3x - 1 = 5$, it might try adding, subtracting, multiplying or dividing by 1, but not by 3 or 5. During the rest of this run, SAGE strengthened each of the useful

† This means that SAGE never strays more than one move from the correct path. This in turn implies that the system could in principle learn without completely solving the problem first. However, since it follows the same strategy for algebra as for other domains, SAGE waits until a complete solution has been found before it attempts to learn.

rules a number of times and created one other spurious variant. The problem was completed when the system suggested dividing by 3, still largely by trial and error.

In the third run, SAGE began to take advantage of its more conservative operators to direct the search. Two initial errors occurred when the program considered multiplying and dividing by 5. In each case, the two useful variants were strengthened, and the first of them (the inverse rule) finally surpassed the original version. Although this rule was still overly general, its direction combined with luck to solve the problem without further errors, and the fourth run was solved perfectly in two steps (again partially by chance).

At this point, SAGE was presented with a more challenging problem that required four operations, $(5x-2)/3+4=5$. In this context, the inverse rule led to errors, and the correct version of the operator was finally produced:

If you see a *number* as the argument of *function1*,
  and *function2* is a function,
  and *function1* is the inverse of *function2*,
  and *function1* occurs at the top level,
then apply *function2* to both sides with *number* as its argument.

Some 11 additional variants were produced as well, but none were created as often as the above version, and so never provided serious competition. Five more runs were required before the final production exceeded its predecessor in strength and the problem was solved without error. Figure 2 presents SAGE's learning curve on the

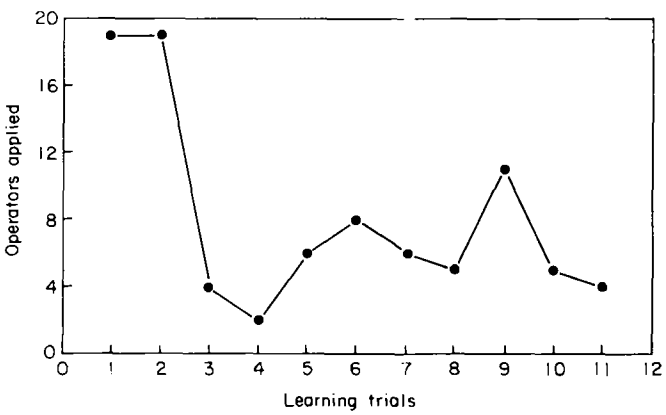

FIG. 2. Learning curve for the algebra task.

two algebra tasks. The increase during the fifth run corresponds to the introduction of the second (more complex) problem, while the increase during the sixth run and the peak during the ninth run were due to the chance application of incorrect operators. Although the system eventually mastered the correct strategy, in this case its progression was far from smooth.

6.3. LEARNING SERIATION STRATEGIES

SAGE's final problem was to learn a strategy for seriating blocks. In this task, the system was presented with a pile of blocks and asked to line them up in order of

decreasing height. Young (1976) and Baylor, Gascon, Lemoyne & Pother (1973) have constructed production system models of children at various stages of seriation ability. Although SAGE is not intended as a detailed psychological model, it does suggest one way in which novices could gain expertise in this domain.

The most efficient method for seriating a set of blocks is to select the largest one and place it in the line, then select the next largest, and so on until all have been included in the ordering. However, SAGE's initial operator is too weak to propose such directed moves; this original rule may be paraphrased as:

> If you have a *block* in the pile,
> then consider moving *block* to the end of the line.

In addition, the system contains rules for implementing its goals, for noticing an illegal state (when a taller block is placed to the right of a shorter one), and for backing up (after reaching an illegal state or when all possibilities have been tried). In this domain, learning consists of *transferring* knowledge to the operator which is already present in the test for illegal states. As before, this transfer is carried out by that set of productions common to all versions of the system.

Relying on its single operator to propose moves, SAGE carried out a depth-first search through the space of possible orderings. When a proposal was implemented, the composite representation of the problem state was modified as it was in previous tasks. For example, if block C were placed in the line next to block A, the program would delete the fact (*in-pile C*), and add the facts (*in-line C*) and (*C is-left-of A*), assuming the position of A were represented by the element (*in-line A*). The relative sizes of objects were represented by additional elements such as (*A is-taller-than C*), which remained unchanged throughout the problem.

SAGE was initially presented with a five-block seriation problem, in which block A was the tallest and block E the shortest. After finding the correct five-move path by depth-first search, the system moved into learning mode. Following some poor selections at the outset, the system correctly selected blocks A and B for the first two moves. At this point, SAGE passed over block C and selected D as candidate for the next move. This error led to two variants on the original operator:

> If you have a *block1* in the pile,
> and you have another *block2* in the line,
> and *block2* is taller than *block1*,
> then consider moving *block1* to the end of the line.

and

> If you have a *block1* in the pile,
> and there is no *block2* in the pile such that *block2* is taller than *block1*,
> then consider moving *block1* to the end of the line.

Here we find two cases in which the discrimination process was forced to chain through elements in order to find differences between the selection and rejection contexts. The first of these productions says that one should remove a block from the pile if another directly larger than it is already in the line. This is a useful variant, but since it cannot apply when nothing is in the line, it will never suggest an initial move. The second rule (in which the new component is actually a negated conjunction) states

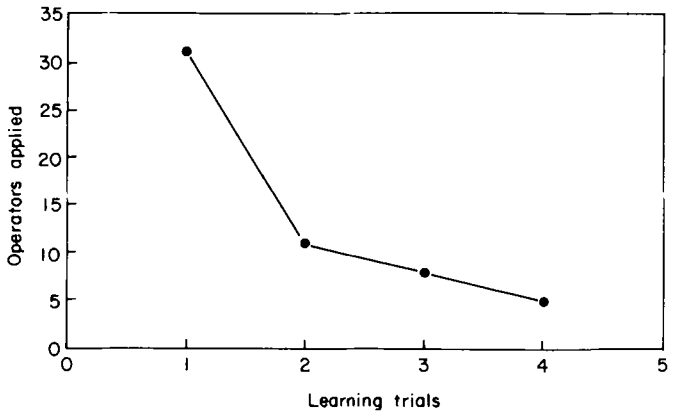

FIG. 3. Learning curve for the seriation task.

that one should always move the largest block in the pile; it is more useful, since it even suggests the correct initial move.

For its next move, SAGE correctly selected C, the largest block remaining in the pile. However, it suggested moving E on the next round instead of D, and the resulting call on the discrimination process produced a third variant that could be used to direct the search process:

If you have a *block1* in the pile,
  and you have another *block2* in the pile,
  and *block1* is taller than *block2*,
then consider moving *block1* to the end of the line.

This rule may propose that any block in the pile except the smallest be moved; it is an improvement over the original, but far from perfect. During the same time period, the first two variants were strengthened as well. Following these events, the program correctly moved D and then E to complete the task.

At the outset of the third run, SAGE attempted to move blocks B and C, and these errors led to the strengthening of the first two variants discussed above. A correct suggestion for moving A was followed by a proposal for moving C, and in this case only the second (and most useful) production was incremented. This was sufficient to push its strength beyond that of the original operator, and since this rule represents the optimal strategy, SAGE solved the remainder of the third run and all of the fourth run with no mistakes. Only the three variants described above were generated for this problem. Figure 3 presents the learning curve for the five-block seriation task.

7. Generality of the system

Earlier in the article, I promised to describe a *general* theory of strategy acquisition. In order to guarantee that generality, SAGE was tested on three rather different domains. However, one might hope that more could be said than the simple claim that the system could learn on three different tasks. Below I consider the generality of SAGE's representation of problem states, along with that of its learning mechanisms.

I also discuss some limits on the theory's generality, before moving on to some suggestions as to how the system might be improved along this dimension.

7.1. GENERALITY OF THE REPRESENTATION

One place to search for generality is in the representation used by a system on different tasks. As we have seen, SAGE employs very similar representations for the slide–jump, algebra and seriation domains. The predominant feature of these representations is the dispersal of information into small chunks. In the case of slide–jump, each chunk included knowledge about the type of coin occupying a particular position, or the adjacency of two positions. In the domain of algebra, SAGE represented the current state of the problem as a tree, with each branch of the tree stored as a separate chunk.† And in the seriation task, the current location of a block (in the pile or the line), and its relative position when in the line, were stored as separate pieces of information.

There were three reasons for dividing the knowledge state into such small components. First, this representation allowed the statement of operators which matched against relevant parts of the problem state, but which did not need to mention irrelevant portions. Second, these operators could alter parts of the state without bothering with those parts that were unaffected by the action; the entire state need not be rewritten every time an action was taken. Finally, the division of information into small chunks was required by the discrimination mechanism, which searched for differences in terms of working memory elements that were present in one situation but not in another. When discrimination led to variants with additional conditions, these could be added independently of features that were not found to differ. If SAGE is applied to new domains, we can expect that such componential representations will have the same benefits.

7.2. GENERALITY OF THE LEARNING MECHANISMS

SAGE's learning mechanisms can be divided into two components that can be evaluated individually along the generality dimension. The first of these consists of the productions responsible for relating proposed moves to those occurring along the solution path. These rules are responsible for determining good and bad moves, and for assigning credit or blame to the productions proposing the moves. Bundy & Silver (1982) would classify these rules as the *critics* of the system. Since the same critics are used in all three versions of SAGE, one is led to conclude that they are quite general heuristics that can aid learning in any domain for which one can initially find a solution path by trial and error, or in which a benevolent tutor provides a sample solution. Some difficulties arise when multiple solution paths exist, and this is one issue that future work should address.

The second component consists of the discrimination and strengthening mechanisms that are evoked by the critics when an undesirable move is detected. The notion of strengthening a rule upon recreation provides little opportunity for criticism; since strength represents a measure of success, this heuristic would seem applicable to any learning task. Whether such a strengthening process is *required* is another question

† Neves (1981) used a very similar representation in modeling the acquisition of algebra strategies, and my first insights into the usefulness of such a representation resulted from discussions with him.

entirely; Langley (1983) considers some of the advantages of strengthening in dealing with noise and incomplete representations. The discrimination process appears to provide a viable alternative to generalization-based methods for finding conditions on a rule, and since it is evoked when errors occur, it seems well-suited to the task of strategy acquisition. In addition, this learning method has been successfully applied to the domains of concept learning (Langley, 1983) and language acquisition (Langley, 1982). Langley (1983) discusses the generality of the discrimination process in more detail, but it is apparent that this approach to procedure learning is a general one that deserves to be explored further in the future.

7.3. LIMITS ON SAGE'S GENERALITY

Of course, SAGE is not as general as one might desire. We have seen that the system must be provided with operators for moving through the relevant problem spaces, along with tests to determine when the goal state has been reached. In addition, the discrimination mechanism must be told which symbols it should consider relevant in its search for differences, and which terms it should generalize across when building rules. For example, in the slide–jump task, the type of coin was irrelevant, while the type of move made at earlier times was quite important. Also, certain information was not considered by the discrimination process, such as the relative positions of coins in the slide–jump problem, although it is not clear whether this bias was necessary to insure learning.

However, SAGE's generality is limited in a much more subtle manner as well. Although the system's learning mechanisms appear to be quite general in themselves, the representation they operate upon was carefully selected. We have seen that the representations for the different tasks have much in common, but one can imagine other representations of these problem spaces that share the same features. Thus, the question is whether these alternative representations could also lead to successful learning. If not, this fact is not necessarily fatal for the theory, since some representations are clearly more useful than others even in problem-solving (the mutilated checkerboard is a popular example). However, this would mean that our theory of strategy acquisition would be incomplete until some explanation was provided for the origin of the particular representations that SAGE requires for learning. In the following section, I discuss one way in which such an explanation might be provided, along with other directions for future research.

8. Directions for future research

Science is an incremental process, and though the theory discussed in previous sections has many noteworthy features, it is far from a complete model of the strategy learning process. It is appropriate, therefore, to consider the directions in which the theory might be extended. Below I discuss a number of dimensions along which such extensions might be attempted.

8.1. EXTENSIONS TO OTHER DOMAINS

Although I have made significant claims for the generality of SAGE, one might well question whether the three domains discussed so far constitute a suitable test of that generality. One natural goal for future research would be to see if the system can

learn strategies in still other domains. Obvious tasks would include puzzles such as those addressed by Newell, Shaw & Simon's (1960) General Problem Solver, like the Tower of Hanoi and Missionaries and Cannibals. Because these tasks involve fairly small problem spaces with well-understood structures, they are ideal for initial tests on developing systems like SAGE. And though such puzzles have much in common with the slide–jump task, they provide considerable variety in their details. For example, within this class one can find heuristics that involve disjunctive conditions, and others that involve both previous move and state information. These are precisely the types of variations necessary to determine the class of heuristics that SAGE is capable of discovering.

Although puzzles have a definite role to play in testing the generality of our theory of strategy acquisition, they do have limitations. In general, these tasks involve rather small search spaces and only a few operators. If SAGE is to prove its generality, it must be presented with more challenging tasks as well, and for this purpose the field of mathematics seems ideal. We have seen that SAGE can learn to solve simple algebraic problems in one variable, but much more difficult tasks can be formulated. In particular, the task of symbolic integration seems an interesting one, both because of its complexity and because it would allow one to compare SAGE's learning trace with that of the LEX system of Mitchell *et al.* more directly. Geometry theorem-proving is another likely domain, which has been studied by Anderson, Greeno, Kline & Neves (1981). One seldom speaks of expert puzzle-solvers, but these mathematical domains are difficult enough so that if SAGE actually learned a significant number of useful heuristics, it could legitimately be claimed that it had made the transition from a novice to an expert system.

8.2. THE UNDERSTANDING PROCESS

Before one can solve a problem, much less *learn* to solve it, the solver must have some representation upon which to operate. Hayes & Simon (1974) have attempted to model the *understanding* process, in which the problem-solver constructs such a representation from a written description of the problem. I observed earlier that SAGE's representations were somewhat carefully crafted to ensure that learning would be possible, and that this was a drawback of the current theory. However, if the system were augmented with an understanding component that could *construct* such representations from written instructions, it would be much less dependent on a benevolent programmer. Of course, the translation from text to internal representation is a complex problem in its own right, but for limited areas of discourse it appears to be relatively straightforward. The class of tasks we are considering tend to involve abstract relations between simple objects, suggesting that the construction of a fairly general understanding module would not be too difficult.

Of course, just because the problem-solver has devised a representation for some problem, this does not mean it is the *best* representation, and certain representations can make a problem very difficult to solve. Simon & Hayes (1976) have shown that slightly different written statements of a task can lead one to quite different representations of the same problem, one of which is considerably easier to handle. In such cases, one might attempt to *transform* the current representation into another that is more amenable to treatment. Some very interesting work has been done along these lines by Amarel (1966) and Korf (1980), and SAGE could certainly benefit by the

addition of a component for intelligently transforming representations. However, this research is still in its early stages, and until further progress has been made, SAGE will have to live with carefully crafted representations or, if an understanding routine is added, with carefully worded problem statements.

8.3. LEARNING FROM SAMPLE SOLUTIONS

We have seen that some of the earlier research on strategy learning, such as the work of Brazdil and Neves, involved the presentation of sample solutions to problems. Given that SAGE determines its own sample traces through search, it would seem simple to add a component that would accept solutions from a benevolent tutor as well. The only complication involves the representation of these solution paths. In its current form, SAGE stores a solution path as a sequence of operators (and their arguments), while the sample solution would presumably be presented as a sequence of states.

There are two ways to deal with this discrepancy. First, the system could be revised to represent solution paths it determined on its own as state sequences instead of operator sequences. However, operator traces are usually much simpler than state traces, so that such a change would introduce additional complexity into both the system's working memory and its productions. A better solution would be to incorporate Neves' strategy of including rules for *recognizing* operators based on the differences between successive states. Such rules would be domain-specific, and might even be constructed along with the operators by the understanding component of the system. Upon firing, they would add an operator trace to memory, which could then be used by SAGE's existing critics to determine correct and incorrect moves.

8.4. INTRODUCING ADDITIONAL LEARNING MECHANISMS

Although SAGE is an elegant and general theory of strategy acquisition, there are aspects of the learning process that it does not address. Future versions of the system should incorporate additional learning heuristics that are concerned with these issues. For example, Anzai has shown that certain types of learning can occur before a complete solution path has been found, and SAGE might well be updated to include rules such as his loop-detecting heuristic. Similarly, Neches has considered heuristics for improving the efficiency of well-specified strategies, and once SAGE has mastered the conditions under which an operator should be applied, there is no reason why further learning could not combine these rules into macro-operators or detect redundancies in its solution. Thus, the current system is best viewed as part of a more global theory of strategy learning, that incorporates methods from other approaches as well.

The above extensions can be implemented in two rather different ways. Anzai and Neches have chosen to state their learning heuristics as productions that note regularities in state and operator traces. The current version of SAGE includes a few rules like this, but they are quite simple and are responsible only for noting good and bad moves and evoking the discrimination process. The discrimination mechanism itself (along with the heuristic for strengthening rules upon recreation) is implemented in LISP, and the second choice is to implement additional learning methods in this manner. Neves & Anderson (1981) have discussed a number of such heuristics, including generalization, composition and proceduralization. Composition has been used to explain the automatization of existing strategies by the combination of simple

operators into more complex ones, and proceduralization has been used to model the translation of declaratively-stated rules into production format. Such learning methods have a potential role to play in a global theory of strategy acquisition, but the automaticity with which they are evoked is a significant drawback. More likely, future versions of SAGE will incorporate heuristics that are a compromise between these two extremes, that are stated as productions with conditions that restrict their application, and action sides that evoke powerful LISP functions, much as the discrimination process is implemented in the current version.

8.5. LEARNING TO LEARN

In the preceding pages, I have occasionally referred to the discrimination process as searching a space of possible rules, and this analogy suggests a more radical extension to the system. The very generality of SAGE's discrimination method is also its main weakness, since it cannot take advantage of knowledge about the particular domain in which it is working. One can imagine more restricted versions of the discrimination heuristic that would apply under more specific conditions, but if these variants were introduced by the programmer, SAGE would lose much of its elegance. However, if the system could *learn* these more specific heuristics on its own, it would gain additional power without any loss in generality.

The view of discrimination as a search process suggests a scheme by which such domain-specific learning rules might be created. SAGE learns by finding a solution path through some problem space, and then using that path to assign credit to particular instantiations of operators. Now in principle there is no reason why this operator cannot be the *discrimination* operator, with the problem space being the space of rules through which the discrimination process moves in search of correct rules. We have seen that this learning heuristic generates many useless variants, and these would be treated as negative instances for which the discrimination operator should not have been applied. Variants containing only useful conditions would lie along the "solution path", and would be treated as positive instances. Of course, SAGE could only distinguish between useful and spurious variants after it had completed learning about a particular task; once it had solved a number of problems without errors, it could be fairly certain that it had found the correct path through the rule space, and could begin to learn about the discrimination process itself. The result of this higher level learning would be domain-specific versions of the discrimination heuristic which would in the future generate only useful variants and bypass spurious rules entirely.

Although the details of the "learning to learn" scheme remain to be worked out, a number of requirements are clear. First, the discrimination heuristic would have to be restated as a condition–action rule, so that the introduction of additional conditions would both restrict its application and direct its search. Second, this revised learning rule would have to leave some trace of its application in memory, both to provide information about the solution path and for the critics to decide whether a good or bad move had been made. Third, SAGE would have to be extended to deal with multiple solution paths, since correct variants can always be generated in a number of different ways. Finally, the system would have to be given information about the task domain that could be used to direct search intelligently through the space of rules. Unless such information is present, the discrimination process would not be able to detect any differences between good and bad instances of its application. This

appears to be the most difficult of the necessary extensions, since it is not clear just what sorts of information would be relevant to directing the learning process. However, there seems to be no reason in principle why such bootstrapping could not occur, and applying the discrimination heuristic to generate more powerful versions of itself is an appealing notion.

9. Conclusions

In conclusion, SAGE is a general strategy learning system that has been tested in three different domains. Although the program has a number of limitations, these suggest some natural extensions to the system that are within the range of existing techniques. However, can one draw any additional conclusions from the program's behavior? Certainly, the system's success provides evidence that the four learning principles proposed at the outset are general, and that they play a central role in strategy improvement. In addition, SAGE presents an example of how these principles can be implemented. Thus, it is apparent that the generation of alternatives can be cast in a depth-first search or means-ends analysis framework. Similarly, achieving knowledge of results and the attribution of causes can be simplified by retaining a trace of the original solution path. And finally, the mechanisms of discrimination and strengthening are powerful methods for modifying behavior in the light of this knowledge.

One of SAGE's most interesting characteristics is that it learns *gradually*. In this respect, it mimics the incremental nature of much human learning. Some of the system's slowness derives from the strengthening procedure; a production must be strengthened a number of times before it masks the rule on which it was patterned. But the discrimination process itself sometimes works in stages. In the slide–jump puzzle, a variant of the original slide operator first had to override its parent before it could make its own mistakes and generate its own offspring. A similar process occurred across different problems in the algebra domain. Thus, the incremental nature of SAGE's learning mechanisms are a promising feature deserving of further study.

Clearly, much work remains to be done, and I have already suggested some directions in which SAGE might be extended. Although future versions of the system may have many additional abilities, they will probably retain the central core of learning heuristics that have served the current system so well. Applied to more complex and challenging task domains, these programs should further our understanding of the processes by which novices evolve into experts, and extend our knowledge of strategy acquisition through experimentation.

I would like to thank Stephanie Sage, Derek Sleeman, Tom Mitchell, and David Klahr for their suggestions, and for comments on an earlier version of this paper.

References

AMAREL, S. (1968). On the representation of problems of reasoning about actions. In MICHIE, D., Ed., *Machine Intelligence* **3**. New York: American Elsevier.

ANDERSON, J. R., KLINE, P. J. & BEASLEY, C. M. (1980). Complex learning processes. In SNOW, R. E., FEDERICO, P. A. & MONTAGUE, W. E., Eds, *Aptitude, Learning, and*

Instruction: Cognitive Process Analyses. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

ANDERSON, J. R., GREENO, J. G., KLINE, P. J. & NEVES, D. M. (1981). Acquisition of problem-solving skill. In ANDERSON, J. R., Ed., *Cognitive Skills and Their Acquisition.* Hillsdale, New Jersey: Lawrence Erlbaum Associates.

ANZAI, Y. (1978*a*). Learning strategies by computer. *Proceedings of the Second National Conference of the Canadian Society for Computational Studies of Intelligence*, pp. 181–190.

ANZAI, Y. (1978*b*). How one learns strategies: processes and representation of strategy acquisition. *Proceedings of the Third AISB/GI Conference*, pp. 1–14.

ANZAI, Y. (1978*c*). Self-organizing production system and its application to simulation of self-organizing aspects of human cognitive behavior. *Proceedings of the International Conference on Cybernetics and Society*, pp. 1503–1507.

ANZAI, Y. & SIMON, H. A. (1979). The theory of learning by doing. *Psychological Review*, **86**, 124–140.

BAYLOR, G. W., GASCON, J., LEMOYNE, G. & POTHER, N. (1973). An information processing model of some seriation tasks. *Canadian Psychologist*, **14**, 167–196.

BRAZDIL, P. (1978). Experimental learning model. *Proceedings of the Third AISB/GI Conference*, pp. 46–50.

BUNDY, A. & SILVER, B. (1982). A critical survey of rule learning programs. *Proceedings of the European Conference on Artificial Intelligence*, pp. 151–157.

CHASE, W. G. & SIMON, H. A. (1974). Perception in chess. *Cognitive Psychology*, **4**, 55–81.

FORGY, C. L. (1979). The OPS4 Reference Manual. *Technical Report*, Department of Computer Science, Carnegie-Mellon University.

HAYES, J. R. & SIMON, H. A. (1974). Understanding written problem instructions. In GREGG, L., Ed., *Knowledge and Cognition.* Potomac, Maryland: Lawrence Erlbaum Associates.

HAYES-ROTH, F. & MCDERMOTT, J. (1976). Learning structured patterns from examples. *Proceedings of Third International Joint Conference on Pattern Recognition*, pp. 419–423.

KORF, R. E. (1980). Toward a model of representation change. *Artificial Intelligence*, **14**, 41–78.

LANGLEY, P. (1982). Language acquisition through error recovery. *Cognition and Brain Theory*, **5**, 211–255.

LANGLEY, P. (1983). A general theory of discrimination learning. In KLAHR, D., LANGLEY, P. & NECHES, R., Eds, *Production System Models of Learning and Development.* Cambridge, Massachusetts: M.I.T. Press.

LANGLEY, P. & NECHES, R. T. (1981). PRISM User's Manual. *Technical Report*, Department of Computer Science, Carnegie-Mellon University.

LANGLEY, P. & SIMON, H. A. (1981). The central role of learning in cognition. In ANDERSON, J. R., Ed., *Cognitive Skills and Their Acquisition.* Hillsdale, New Jersey: Lawrence Erlbaum Associates.

LARKIN, J. H., MCDERMOTT, J., SIMON, D. P. & SIMON, H. A. (1980). Expert and novice performance in solving physics problems. *Science*, **208**, 1335–1342.

MITCHELL, T. M. (1977). Version spaces: A candidate elimination approach to rule learning. *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, pp. 305–310.

MITCHELL, T. M., UTGOFF, P., NUDEL, B. & BANERJI, R. B. (1981). Learning problem solving heuristics through practice. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pp. 127–134.

MITCHELL, T. M., UTGOFF, P. & BANERJI, R. B. (1982). Learning problem solving heuristics by experimentation. In MICHALSKI, R., CARBONELL, J. & MITCHELL, T. M., Eds, *Machine Learning: An Artificial Intelligence View.* Palo Alto, California: Tioga Press.

NECHES, R. T. (1981). A computational formalism for heuristic procedure modification. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pp. 283–288.

NEVES, D. M. (1978). A computer program that learns algebraic procedures by examining examples and working problems in a textbook. *Proceedings of the Second National Conference of the Canadian Society for Computational Studies of Intelligence*, pp. 191–195.

NEVES, D. M. & ANDERSON, J. R. (1981). Knowledge compilation: Mechanisms for the automatization of cognitive skills. In ANDERSON, J. R., Ed., *Cognitive Skills and Their Acquisition.* Hillsdale, New Jersey. Lawrence Erlbaum Associates.

NEWELL, A., SHAW, J. C. & SIMON, H. A. (1960). Report on a general problem-solving program for a computer. *Information Processing: Proceedings of the International Conference on Information Processing*, pp. 256–264.

RENDELL, L. A. (1982*a*). A new basis for state-space learning systems and a successful implementation. *University of Guelph Technical Report CIS82-1.*

RENDELL, L. A. (1982*b*). State-space learning systems using regionalized penetrance. *Proceedings of the Fourth National Conference of the Canadian Society for Computational Studies of Intelligence*, pp. 150–157.

SIMON, H. A. & HAYES, J. R. (1978). The understanding process: problem isomorphs. *Cognitive Psychology*, **8**, 165–190.

SLEEMAN, D., LANGLEY, P. & MITCHELL, T. (1982). Learning from solution paths: an approach to the credit assignment problem. *AI Magazine* (Spring), 48–42.

VERE, S. A. (1975). Induction of concepts in the predicate calculus. *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pp. 281–287.

WINSTON, P. H. (1970). Learning structural descriptions from examples. *MIT AI-TR-231.*

YOUNG, R. M. (1976). *Seriation by Children: An Artificial Intelligence Analysis of a Piagetian Task.* Basel: Birkhauser.