COGNITIVE SCIENCE 9, 217-260 (1985)

Learning to Search: From Weak Methods to Domain-Specific Heuristics*

PAT LANGLEY

The Robotics Institute
Carnegie-Mellon University

Learning from experience involves three distinct components—generating behavior, assigning credit, and modifying behavior. We discuss these components in the context of learning search heuristics, along with the types of learning that can occur. We then focus on SAGE, a system that improves its search strategies with practice. The program is implemented as a production system, and learns by creating and strengthening rules for proposing moves. SAGE incorporates five different heuristics for assigning credit and blame, and employs a discrimination process to direct its search through the space of rules. The system has shown its generality by learning heuristics for directing search in six different task domains. In addition to improving its search behavior on practice problems, SAGE is able to transfer its expertise to scaled-up versions of a task, and in one case, transfers its acquired search strategy to problems with different initial and goal states.

INTRODUCTION

The ability to search is central to intelligence, and the ability to *direct* search down profitable paths is what distinguishes the expert from the novice. However, since all experts begin as novices, the transition from one to the other should hold great interest for Artificial Intelligence (AI). In this paper, we examine the process by which general but weak methods are transformed into powerful, domain-specific search heuristics. Readers should be able to detect two main themes. In the early sections of the paper, we have attempted to classify the types of heuristics learning that can occur,

* We would like thank Stephanie Sage, who helped in programming and debugging the SAGE system as well as Drew McDermott and Rich Korf, who provided useful comments on an earlier draft.

Correspondence and requests for reprints should be sent to Pat Langley, Department of Information and Computer Science, University of California, Irvine, CA 92717.

as well as the components that contribute to such learning. After these preliminaries have been completed, we explore a particular learning system—SAGE.2—in some detail, both in terms of its structure and in terms of its behavior in different domains. We close with a discussion of some directions in which the system should be extended.

Within any system that improves its search strategies with experience, we can identify three distinct components. First, such a system must be able to *search,* so that it can generate behaviors upon which to base its learning. Second, the system must be able to distinguish desirable from undesirable behaviors, and to determine the components of the system that were responsible for those behaviors; in other words, it must be able to *assign credit and blame.* Finally, the system must be able to use this knowledge to *modify* its search strategies, so that behavior improves over time. Since so much AI research has revolved around the notion of search, it is not surprising that the first of these components is the best understood. Many alternative search strategies have been explored, ranging from very general but weak methods, like depth-first and breadth-first search, to much more powerful methods that incorporate knowledge about specific domains. It is precisely the transition between weak, general methods and specific, powerful methods with which we are concerned. Thus, it is appropriate that a strategy learning system start with some weak search scheme that can be applied to many different domains. However, it is also important that the search control can be easily modified to take advantage of domain-dependent knowledge that is acquired with experience. The areas of credit assignment and modification are less well understood, and we discuss them in some detail in later sections. However, before turning to these matters, let us consider the problem of learning search heuristics in the context of a simple puzzle.

Over the years, the Tower of Hanoi puzzle has been used as a testbed for many different AI systems. We have chosen this task for our example because it is so well-known to the AI community, and because it poses a challenging problem to humans despite its small search space. In this puzzle, one is presented with three pegs on which are placed N disks of decreasing size. Initially, all disks are placed on a single peg, and the goal is to get all of these disks onto one of the other pegs. This task would be trivial except for two constraints on the types of moves that are allowed. First, one can only move the smallest disk from a given peg. Second, one cannot move a disk onto another peg if a smaller disk is already resting on that peg. Taken together, these restrictions considerably constrain the set of legal moves, and make for a challenging problem.

Figure 1 presents the state space for the three-disk Tower of Hanoi problem, originally formulated by Nilsson (1971), while Figure 2 shows two of these states in more detail. Note that although only 27 states exist in the space, the number of connections between these states is very large. One

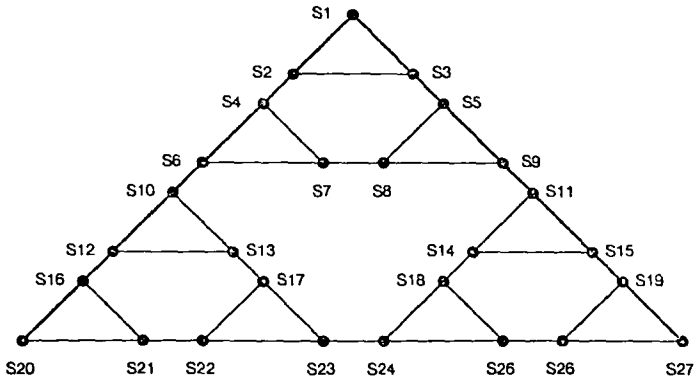

Figure 1. State space for the three-disk Tower of Hanoi puzzle.

result of this high density of connections is that loops are very easy to generate.[1] Another result is that while many paths to a goal are possible, only a few are optimal. In other words, within the state space for the three-disk problem, considerable search may be necessary to find an optimal solution path. Suppose S1 is given as the initial state (in which all disks are on a single peg), and the goal is to reach either state S20 or state S27 (in which the disks are all on another peg).[2] Further assume that we employ a very general but weak search strategy such as depth-first or breadth-first search to solve this problem. Given such weak search control, many nonoptimal moves will be considered before the best set of moves is discovered. For example, a breadth-first search scheme would consider moving from state S2 to S3, as well as the optimal move from S2 to S4. The goal of a strategy-learning system is to discover a set of heuristics that will propose moves lying on the solution path, while avoiding those leading off the path. In the following sections, we consider some of the ways in which such search heuristics can be acquired.

TYPES OF STRATEGY LEARNING

Throughout the history of science, the first step in understanding a set of phenomena has involved the construction of taxonomies or classification schemes. Thus, the early chemists formulated classes such as acids, alkalis, and salts before they began to discover quantitative laws for reactions. Similarly, in biology the acceptance of the Linnaean classification system

[1] Loops are possible because all moves are reversible. For example, one can move from State S2 to S1 as easily as from S1 to S2, though longer loops can also occur.

[2] In most versions of this task, the goal involves moving all disks to a *single* peg; we will discuss the reason for allowing multiple solutions later in the paper.

preceded Darwin's recognition of similarities between classes and his explanation of their evolutionary relations. By analogy, it would seem useful to attempt to categorize the various types of strategy improvement before attempting to explain the processes responsible for them.

Ohlsson (1982) has distinguished between *improvement,* in which search decreases on a single practice problem, and *transfer,* in which practice on one set of problems leads to a reduction in search on a second set of problems. Building upon this distinction, it is possible to subdivide the class of transfer learning still further. One type of transfer involves the *scaling up* of simple problems into more complex ones. We have seen that for puzzles such as the Tower of Hanoi, one can draw a state space diagram representing the possible states and the moves connecting them. The state space for the four-disk puzzle is very similar to that for the simpler problem, and can be generated by replacing each state in Figure 1 by a triangle of states. Given this similarity of structure, one might expect that heuristics learned for solving the three-disk problem would easily transfer to the four-disk problem. However, more steps would be involved in reaching a solution, so this problem is a scaled-up version of the three-disk problem.[3]

Figure 2. Moving disk-1 from peg-A to peg-C on the Tower of Hanoi puzzle.

A second type of transfer occurs when one practices on one problem, and then is presented with another problem that involves the same state space, but has a different initial state or a different goal state. For example, one might learn a set of heuristics for moving from state S1 to S20 or S27 in the three-disk problem, and then be asked to find a path between state S7 and S14. In general, this type of transfer would appear to be more difficult than scaled-up transfer, since one must take goal information into account while constructing one's heuristics.

In domains such as algebra and integration, the state spaces for different problems bear little similarity to one another, since only a few of the many possible operators come into play on a given problem. However, the goals always have very similar forms—to simplify an expression or to solve for some variable. As a result, the above two types of transfer seldom occur in such domains. In these cases, one usually practices on one set of prob-

[3] The difficulty of a problem can sometimes be altered in multiple ways. For example, one can formulate a variation of TOH puzzle that involves three disks and four *pegs.* In fact, this problem can be solved in fewer steps than the standard version, but the point is that difficulty can sometimes be affected in more than one way.

lems, and then is tested on a different set of problems that, while they differ in the structure of their state spaces, have approximately the same *complexity*. This type of transfer constitutes the third member in our classification scheme.

Finally, one may sometimes attempt to use knowledge learned in an area that is only loosely related to the current situation. In such cases, only some of the operators used earlier may be applicable to the space currently being searched, and others that were not applicable before may come into play. Still, one may be able to take advantage of some of the heuristics that were acquired in the first class of problems and apply them to the task at hand; this form of transfer is usually called *learning by analogy*. Taken together, these four classes would seem to cover the ways in which transfer of learning can occur, though one might propose alternate divisions based along other dimensions.

While we do not have the space to review earlier research on strategy learning in detail,[4] it will useful to classify the existing work in terms of our categories. For instance, Anzai (1978) focused on improvement within the three-disk Tower of Hanoi task, but did not address the issue of transfer. In contrast, Brazdil's (1978) concern with arithmetic has led him to explore transfer to scaled-up problems and to problems of equal complexity, and Neves (1978) has also examined the latter in the context of algebra learning. Mitchell, Utgoff, and Banerji's (1983) research on symbolic integration and Anderson's (1981) work on geometry theorem proving have also been concerned with the latter type of transfer. Langley's SAGE.1 (1982a, 1983)—the predecessor of the current system—showed both improvement on a single problem and transfer to scaled-up problems, while Ohlsson's UPL2 (1983) showed both improvement and some ability to transfer to problems with different initial states and goals. Rendell's (1983) PLS1 system was able to transfer its heuristics to both scaled-up problems and to those with different initial and goal states. Like Anzai, Hagert (1982) has focused on improvement on the Tower of Hanoi task, while Korf's (1982) macro-operator learning program was able to transfer its expertise to problems with different initial states. Finally, both Carbonell (1983) and Anderson (1983) have studied learning by analogy, in which knowledge gained in solving one problem is applied to direct search in a quite different problem. We summarize this information in Table 1.

Later in the paper, we will examine the behavior of a particular strategy learning system called SAGE.2. To anticipate our results, we will find that SAGE is capable not only of improvement, but that it is also capable of transfer to scaled-up tasks and to problems of equal complexity. We will

[4] The interested reader is directed to Keller (1982) and Langley (1983) for reviews of some recent work in the area.

TABLE I
Types of Learning Addressed in Earlier Research

	Improvement	*Scaled-up*	*Diff. Goals*	*Equal Comp.*	*Analogy*
ANZAI	X				
BRAZDIL	X	x		X	
NEVES	X			X	
MITCHELL	X			X	
LANGLEY	X	X			
OHLSSON	X		X		
RENDELL	X	X	X		
HAGERT	X				
KORF	X		X		
ANDERSON	X			X	X
CARBONELL	X				X

also find that the current system has difficulty in transferring its expertise to problems with different initial and goal states, but that the potential for this form of transfer does exist. Finally, learning by analogy appears to lie beyond the methods employed by the program. Hopefully, the reader now has a better understanding of the types of transfers that can occur and those types we will focus on in the following pages. Now let us move on to the components of the strategy learning process.

APPROACHES TO CREDIT ASSIGNMENT

As we have seen, the first step in learning is to distinguish desirable from undesirable behaviors, and to determine the parts of the system responsible for those behaviors. This has been called the *credit assignment* problem, and has been explored in a number of domains, ranging from puzzle solving to chess playing. We have arrived at a number of heuristics for assigning credit and blame that appear to be quite general, some of which we have borrowed from other researchers. All of these methods involve the same basic idea—that steps lying along optimal solution paths should be preferred to those leading off those paths. However, the various methods make judgments about preferable moves in quite different ways. Below, we discuss these heuristics in the context of the Tower of Hanoi puzzle and a few other simple tasks.

Complete Solution Paths

One option for distinguishing desirable from undesirable behavior is to wait until a complete solution path has been found for a problem. Moves leading to states on the solution path are desirable, since they led to a solution,

while moves going off the path are undesirable, since they led elsewhere. Mitchell, et al. (1983) have employed this approach to their LEX system, while Langley (1983) has used a very similar approach in his SAGE.1 program. Brazdil (1978) and Rendell (1983) have also employed the complete solution path heuristic. Sleeman, Langley, and Mitchell (1982) have discussed the generality and limitations of this approach to credit assignment.

Let us consider how this technique can be applied to the Tower of Hanoi puzzle. Figure 1 presents the state space for the three-disk puzzle, with the two solution paths connecting the top vertex to the two bottom vertices. Given the legal operators for solving the puzzle, many problem-solving systems can discover the solutions by searching this space. Once the solution paths have been discovered, they can be used to assign credit and blame. For example, since both moves from the initial state S1 lie on the solution path, both would be labeled as good moves. Three moves are possible from each of the resulting states S2 and S3. The moves leading to states S4 and S5 also lie on the solution path, and so would be marked as good moves. However, the moves leading to states S3 and S2 lie off the solution path, as do the two moves leading back to the initial state. Thus, all of these moves would be labeled as undesirable.

This approach is very general, since it can be used to assign blame and credit to any problem that can be solved by search. However, this method is guaranteed to work only if *all* of the shortest solution paths are available. Since some search techniques find only a single solution path, difficulties can arise. For example, a system that solves problems using a form of depth-first search might find one of the solutions shown in Figure 1, but not the other. Given such incomplete knowledge, our credit assignment heuristic would mistakenly label one of the initial moves as undesirable. Mitchell, et al. (1983) have dealt with this problem by carrying out additional search before deciding that a move is bad. Another problem is that while almost any problem can *in principle* be solved purely by search, there are many problems with search spaces so large that some other route must be taken. In these cases, other credit assignment heuristics that do not require complete solution paths must be employed to enable learning to occur *while* the problem is being solved, so that the search process can become directed enough to reach the goal state. We now discuss a number of heuristics that allow credit assignment during the search process, and which open the way to learning while doing.

Noting Loop Moves

When one is attempting to solve a problem in as few steps as possible, returning to a previously visited state (or *looping*) may be safely considered undesirable. Thus, when a move leads to a state through which the problem

solver has already traveled, that move can be labeled as less desirable than another move that does not complete a loop. For example, suppose one is at state S4 in the three-disk Tower of Hanoi problem, and considers moving to states S2, S6, and S7. Since the first of these leads back to the previously visited state S2, it can be labeled as less desirable than the last two moves. Note that this form of credit assignment is relative rather than absolute, as was the case when complete solutions were known. There is no guarantee that the move leading from S4 to S7 will ultimately be deemed desirable (as in fact it will not, since it leads off the solution path). However, one can say that this move is *more* desirable than the one leading back to a previously reached state, and this information may be useful to the modification component of the system. Anzai (1978) has used a loop move detector to good effect in modeling learning on the Tower of Hanoi, but it is clear that this approach can be applied to any domain in which loops can occur during search. Ohlsson (1983) has employed a similar credit assignment technique in his UPL system.

Noting Longer Paths

In general, shorter paths to a goal are more desirable than longer ones. Thus, if a problem solver notes that he has reached some state by two different paths, he can infer that the last move in the longer path should have been avoided. For example, in the three-disk Tower of Hanoi puzzle, suppose one has moved from state S4 to state S7, as well as from S4 to S6. Further suppose that on the next move, one moves from S6 to S7, as well as from S6 to S10. Since the state S7 has been reached by two paths, the last move in the longer path (from S6 to S7) may be judged undesirable. The alternate move from S6 to S10 cannot immediately be deemed good in any absolute sense (though later it would be found to lie on the solution path), but it can be judged as *more* desirable than the move from S6 to S7. Thus, this is another case where the assignment of credit and blame takes on a relative aspect. The shorter-path heuristic is closely related to the loop move method, and appears to be another quite general technique for assigning credit during the search process. Anzai (1978) has applied a very similar technique to learning on the Tower of Hanoi task.

Dead Ends

In solving a problem, a path must be found from the initial to the goal state. However, some paths lead to dead ends from which no steps can be taken except to back up, and it is desirable to avoid these cul-de-sacs if possible. Another generally useful credit assignment heuristic labels as bad the last

move in a path that has led to a dead end. For example, suppose in solving the three-disk Tower of Hanoi problem, one has moved from state S4 to S7. Also suppose that after this, one has tried moving from S7 to S4, from S7 to S6, and from S7 to S8. If the first of these moves is labeled as bad by the loop move heuristic, and the second two are marked as bad by the shorter-path heuristic, then the state S7 may be classified as a dead end. As a result, the move from S4 to S7 may be judged as undesirable, and the move from S4 to S6 may be judged as a better move, since it does not lead to any undesirable state. Again, this heuristic cannot decide that the S4 to S6 move is absolutely desirable (though it does lie on the solution path), but it can determine that this move should be *preferred* to its alternative.

Failure to Progress

We have so far referred to the initial search strategy only in the abstract. However, some search strategies are more powerful than others, and this power can be used in assigning credit and blame before a complete solution has been found. For example, search methods such as means-ends analysis and hill-climbing employ an evaluation function which tells whether one is closer to the goal after a move has been made than he was before. Let us consider a simple example from the domain of algebra. In solving algebra problems in one variable, simplifying the expression will take one closer to the goal (in which the variable is on one side of the equation and a number is on the other). Thus, if a step is taken which does *not* simplify the expression, this may be judged as an undesirable move. Another move made from the same state that *does* lead to a simplification may be judged as more desirable, though (in principle at least) it might not be the best move possible. Neves (1978) employed such a credit assignment principle in his ALEX system, enabling it to learn algebra heuristics before a complete solution had been achieved. The implementation of such a principle might be quite general, as in Ohlsson's (1983) UPL 2 system, which used a form of means-ends analysis, or it might be relatively specific, as in knowing that algebra expressions should always be simplified.

Illegal States

A final heuristic for the determination of credit and blame revolves around the notion of illegal states. In some cases, the problem solver may attempt to make moves which he later recognizes as violating some task constraint. For example, in the Tower of Hanoi puzzle, one might attempt to move the largest disk, even though one or more smaller disks were resting on it. Of course, such a move is undesirable, and any move from the same state that

does not violate a constraint may be judged as better. This is yet another case in which the desirable move is only relatively good, and that move may be judged as undesirable at some later point in the search process. In principle, this heuristic may be applied to any task that involves some form of constraints. However, problem solvers often incorporate such constraints into their operators, and so avoid illegal moves from the outset. Still, this type of mistake occurs among human problem solvers sufficiently often for it to be included in the psychological literature (Simon & Reed, 1976), so we shall keep it on our list of methods for solving the credit assignment problem. Now that we have considered approaches to the first step in the strategy learning process, it is time to move on to the second stage—the modification of behavior.

APPROACHES TO ALTERING SEARCH BEHAVIOR

There exist two rather different approaches to controlling search in an intelligent fashion. In the first scheme, some numerical evaluation function is used to rank states, and those with the highest scores are selected for further expansion. This method is commonly used in game-playing programs. The alternative is to employ heuristics with symbolic conditions to direct search, and this approach has often been applied to puzzle-solving tasks and mathematical domains. As one might expect, both of the methods lead to associated techniques for *altering* search behavior, and both approaches to learning have been explored in the literature. Below we summarize these approaches to strategy acquisition.

Discovering Evaluation Functions

The approach to learning through discovering evaluation functions is an attractive, one and was examined early in the history of AI. Samuel (1959) constructed a checker-playing program that chose its moves on the basis of a linear evaluation function. The system experimentally introduced new terms from a set of predefined features and altered the weights of existing terms, and then noted the result in its playing ability. In this way, Samuel's system eventually progressed to master-level checkers play. Rendell (1983) has explored an alternate approach to finding evaluation functions. His PLS1 program first solves a problem (such as the eights puzzle) using breadth-first search. Once a solution has been found, this information is used to assign a score to each state in the search tree. Using various curve-fitting techniques, Rendell's system generates a function that predicts these scores in terms of a set of predefined features. This function can then be used as an

evaluation function for directing the search process. While such techniques are useful in domains where numeric evaluation functions are appropriate, other methods must be used to acquire heuristics that can only be stated in symbolic terms.

Generalizing Conditions

One technique for learning symbolic conditions begins with very specific rules and *generalizes* as more information is gathered. In this incremental approach, the hypothesized conditions are usually initialized to the first positive instance. When a new positive instance is encountered, it is compared to the current hypothesis and one or more revised hypotheses are generated, based on the features held in common by the two structures. If some of these hypotheses become overly general, they eventually lead to the incorrect classification of negative instances as positive ones and are rejected. Since more than one hypothesis may result from this comparison, some method for controlling search through the rule space is required. Winston (1975) has explored depth-first strategies for searching the rule space, while Hayes-Roth (1976) and Vere (1975) have employed breadth-first search strategies. Since most generalization-based methods search for features held in common by all positive instances, they have difficulty in learning rules with disjunctive conditions. However, Iba (1979) has used an extension of the depth-first scheme to successfully learn disjunctive rules.

Discriminating Conditions

An alternate approach starts with an overly general rule and generates more specific versions through a process of *discrimination*. This occurs when one of the current hypotheses leads to an error, providing evidence that it is too general. The context in which the faulty rule matched the negative instance is compared to the last context in which the same rule matched a positive instance. During this comparison, differences between the positive (desirable) instance and negative (undesirable) instance are found. For each difference, a more specific hypothesis can be constructed that would match against the positive instance but not the negative one. Since multiple hypotheses can result, some search control is required. Brazdil (1978) has used depth-first search to direct the discrimination process, while Anderson and Kline (1979) and Langley (1982b) have employed more complex strategies involving notions of strengthening and weakening. Since the discrimination method compares instances to other instances (rather than to hypotheses), it does not attempt to find features common to all positive instances, and so has no difficulty in learning rules with disjunctive conditions.

The Version Space Approach

Mitchell (1977) has explored the *version space* approach, which incorporates aspects of both the generalization and discrimination methods. This technique begins with a very specific hypothesis and generates more general versions (S) as new positive instances are encountered. As with generalization methods, this is done by finding common features. It also begins with a very general hypothesis and produces more specific versions (G) as experience is gained. However, instead of testing the first set of hypotheses (S) against negative instances to see if they are overly general, it tests them against the second set (G). Similarly, more specific versions of the second set (G) are found by comparing negative instances to members of the first set (S). Mitchell employed a breadth-first strategy to direct search through the space of hypotheses. As more instances are gathered, this bidirectional search converges on the hypothesis best suited to summarize the data. Since Mitchell's method also finds features held in common by all positive instances, it has the same difficulty with disjunctive rules as most generalization-based learning systems.

Implications for Search Behavior

Note that the direction taken in searching for conditions has implications for the performance component of a strategy learning system. For example, if the system moves from specific to general hypotheses through a generalization process, then the associated performance system will be *conservative*. The system will begin by making no bad moves and missing some good moves, but as the system nears the correct hypothesis, its errors of omission will decrease. In contrast, if the system moves from general to specific hypotheses through a discrimination process, then the associated performance system will be a *rash* one, omitting few desirable moves but considering many undesirable ones as well, though the latter will decrease as the correct hypothesis is approached.

While a conservative strategy is useful when a benevolent tutor is available to present positive and negative instances (as in the paradigm of learning concepts from examples), it is less adaptive in learning search heuristics, where a system must generate its own behavior in order to accumulate positive and negative instances of various rules. In this case, the price of commission errors is small, since the only result is added search. However, the price of omissions is great, since learning is impossible in the absence of behavior. Thus, in the context of learning search strategies, the reckless discrimination approach seems superior to the more conservative generaliza-

tion approach.[5] The version space approach is capable of conservative or rash behavior, depending on whether one uses S or G in the match process. However, in this paper we will limit our attention to discrimination-based approaches to strategy learning.

SAGE.2: A SYSTEM THAT LEARNS SEARCH HEURISTICS

Having considered the three components involved in strategy learning, we can now examine a particular strategy learning system in some detail. We will focus on SAGE.2, the second in a line of programs (Langley, 1982a, 1983) that we have constructed to study the process of strategy acquisition. SAGE stands for Strategy Acquisition Governed by Experimentation. Like most other strategy learning programs, SAGE is implemented as an adaptive production system. In other words, it is stated as a set of relatively independent condition-action rules or productions, and learning occurs through the addition of new productions. The program is implemented in PRISM (Langley, 1981), a production system language designed to explore learning phenomena. We now consider the components of SAGE, starting with its representation of states and operators. After this, we discuss the system's initial search strategy, its credit assignment heuristics, and its mechanisms for altering its search strategy in the light of experience.

Representing States and Operators

Any problem-solving system must have some *representation* upon which to work. For a given problem, it must be able to represent the states that constitute the problem space being searched, and to represent the operators that enable the system to move between those states. As we have stated, SAGE.2 is implemented as a production system. Others have argued for the advantages of production system formalisms (Newell, 1972, Anderson, 1976), and we do not have the space to recount those arguments here. However, the choice of production systems leads to a natural style for representing states and operators, and it is appropriate to spend some time discussing that style.

A program that is stated as a production system consists of two main components—a set of condition-action rules or productions and a working

[5] However, Ohlsson (1983) has devised a generalization-based scheme that sidesteps this problem. His UPL2 system begins with a set of overly general rules which lead to search; based on good moves, the program creates specific rules and generalizes them when possible. Although UPL prefers to use such learned rules, it retains the original rules, and so can fall back on them, if the acquired rules fail to propose any move.

memory against which those productions are matched. The working memory tends to be declarative in nature, and changes contents fairly rapidly. In contrast, the production memory tends to express procedural knowledge, and changes only slowly, when learning occurs. During problem solving, new states are generated quite often, while new search procedures are added only occasionally. Therefore, it is quite natural to represent states as elements in working memory, and it is equally natural to represent operators for moving between those states as productions.

Given these design decisions, a question remains as to the precise manner in which states and operators are to be stored. For example, states might be represented as single working-memory elements, as with (in-state S2 (peg-A contains disk-2 disk-3) (peg-B contains disk-1) (peg-C contains)) for the Tower of Hanoi. Alternately, they might be stored as a number of separate elements, such as (disk-1 is-on peg-B in-state S2), (disk-2 is-on peg-A in-state S2), and (disk-3 in-on peg-A in-state S2). Since most production systems languages have limited pattern matching capabilities, the latter of these two schemes is desirable: It lets one express finer distinctions. In fact, this is the representation for states used in SAGE, and it has worked extremely well for our purposes.[6]

Since production system formalisms require a close correspondence between the form of elements in working memory and the form of productions, the choice of representation for states places strong constraints on the representation for operators. For example, the following rule is a natural statement of the conditions under which a disk can be legally moved in the Tower of Hanoi task:

TOH
If you have *disk* on *current-peg* in *current-state*,
  and you have some *other-peg* different from *current-peg*,
  and in *current-state* there is no *other-disk* on *current-peg* that is smaller than *disk*,
  and in *current-state* there is no *third-disk* on *other-peg* that is smaller than *disk*,
then consider moving *disk* from *current-peg* to *other-peg*.

The meaning of this production is self-explanatory, but the correspondence between conditions and working memory may not be so clear. For this rule to be applied, each line must match against some element in working memory. For example, at the outset of the problem, the first line might match against against the elements (disk-1 is-on peg-A in-state S1), (disk-2 is-on peg-A in-state S1), or (disk-3 is-on peg-A in-state S1). Similarly, the

[6] Anzai (1978) employed a representation very much like the first one shown and certainly managed to implement a running system. However, this approach required that he build considerable knowledge into his learning mechanisms about the particular representation he was using. In our opinion, this was one of the reasons why Anzai never managed to get his system to learn in more than a single domain.

second condition would match against the elements (peg-B is-a peg) and (peg-C is-a peg). The remaining negated conditions would match against elements (disk-1 is-on peg-A in-state S1) and (disk-3 is-larger-than disk-1). Italicized terms in the rule stand for variables which can match against any symbol; in addition to matching within individual conditions, variables must bind consistently across conditions for the production as a whole to match. In cases where the negated conditions are successfully matched, they keep the production as a whole from matching. Thus, they can be used to keep this rule from proposing illegal moves, such as moving a disk when a smaller one is resting on it.

Note that the above rule proposes a move but does not actually carry it out; we will call such rules *proposers*. Each proposer contains the legal conditions on an operator, while the operator itself is implemented in a separate rule. This division of labor has two main advantages. First, since we are concerned with improving search strategies, our system need only alter the *conditions* under which actions are proposed. This means that we can ignore the actions involved in an operator and focus on the conditions. Second, as we shall see later, SAGE learns by creating variants of proposers like TOH. In some cases, variants of the same original production fire in parallel, proposing the same action. By introducing an additional step between the move proposal and its implementation, we give the system time to recognize the identity of these proposals and to avoid unnecessary effort.

When a proposal is actually carried out, an *operator trace* is deposited in working memory. These traces refer to the operator that was applied, as well as to the arguments that were passed to it, as in the working memory element (move-1 was move disk-1 from peg-A to peg-B). Information is also stored about the state at which the operator was applied and the state that resulted from its application, as in the element (move-1 led-from S1 to S2). Such trace information is used once a solution has been found, allowing SAGE to chain back up the path, marking traces lying on that path as desirable. The system's other credit assignment heuristics also take advantage of these traces, using them to infer moves leading to undesirable states and to back up to earlier states. SAGE also considers such trace information when it is searching for conditions on its proposers, and can incorporate knowledge of previous moves into the productions it generates. The need for some form of trace data in strategy learning has been emphasized by Neches (1981) and by Langley, Neches, Neves, and Anzai (1980), and our experience with the current system has reinforced our beliefs on this matter.

The Initial Search Strategy

In order to understand SAGE.2's initial search strategy and the manner in which this strategy changes over time, we must consider some more details about the nature of production systems. A given rule may match against the

elements in working memory in more than one way; each such match is called an *instantiation*. Given a set of instantiations, a production system program must have some means of determining which should be applied and which should be saved for later application; this process is called *conflict resolution*. SAGE employs three conflict resolution principles, which are applied in turn. First, instantiations which have been applied before are never selected again; this process of *refraction* keeps the same move from being proposed by the same production, while allowing prior states to be retained in case some other move must be made from them. Second, instantiations matching against more recent states are preferred to those relating to older states; this focuses attention on new states, so that the system continues to explore promising paths. Third, each production has an associated *strength*, and rules with high strengths are preferred to weaker ones; since rules are strengthened each time they are relearned, this number can be viewed as a measure of each rule's *success*, with preference being given to more successful rules.

If two or more rules have equal strength, or if multiple instantiations of a single rule match against elements of the same recency, then more than one move may be proposed at a time. This is the standard situation when SAGE first attempts to solve a problem, since its proposers generally begin with identical strengths, or because it starts with only one such rule. In this case, the system carries out a breadth-first search through the problem space defined by its operators, and the program continues in this exhaustive fashion until credit can be assigned and learning can occur. Once new move proposing rules have been generated and the strengths of the old rules have been altered, search becomes more selective. Although still preferring more recent states, SAGE begins to prefer productions that have been learned many times, and to shun those that have led to errors in the past. However, it retains the ability to consider multiple paths, as long as these paths are generated by rules with the same strengths. For example, it would still be able to find both solutions to the Tower of Hanoi puzzle, since these are perfectly symmetrical. In summary, the system starts by carrying out a blind breadth-first search, and using information it gathers along the way, it ends (perhaps after a number of runs) with the ability to direct its search toward the goal states.

The system must also know when it can stop searching. This is the responsibility of a separate production that recognizes when the goal state has been reached, and adds information to working memory to this effect. For example, the goal-recognizing rule for the Tower of Hanoi puzzle notes when all disks are resting on one of the goal pegs, and adds to memory the names of the states that satisfy this condition. This information is used later in determining the complete solution path. Separate goal-recognizing productions must be provided for each task domain, since the conditions for

the solutions differ. However, the same rule can generally be used for scaled-up versions of a problem; for instance, the goal production for Tower of Hanoi does not refer to the number of disks on the goal peg, and so can be used for the four-disk and five-disk tasks, as well as for the simpler three-disk problem.

SAGE.2's Credit Assignment Heuristics

In an earlier section, we distinguished two basic approaches to altering search behavior. The first of these involved the discovery of evaluation functions, while the second involved the determination of the symbolic conditions under which moves should be proposed. Since SAGE.2 is stated as a production system, the second of these methods seemed most appropriate. As we indicated before, the program employs a discrimination mechanism (as opposed to a generalization or version space method) to determine the heuristic conditions for applying its operators. Since this method inputs a positive and negative instance of some rule, it is appropriate to first consider the manner in which the system assigns credit and blame, and thus distinguishes desirable moves (or positive instances) from those which should be avoided (or negative instances).

SAGE can operate in either of two modes. It can assign credit based only on complete solution paths, or it can attempt to learn during the search process. Since the program's credit assignment heuristics are stated as independent condition-action rules, they can be added or removed without affecting the system's ability to search, though of course this does affect the manner in which learning occurs. Let us begin by focusing on the method relying on complete solution paths. Table 2 shows two productions, ON-THE-PATH and OFF-THE-PATH. The first of these matches against traces of moves that lie along the solution path; upon application, it retrieves the instantiation responsible for proposing the move and stores it as a positive instance of the rule that was matched.[7] The second production matches against traces that originated on the solution path but led off that path when the move was made; upon firing, this rule retrieves the responsible instantiation and marks it as a bad instance of the rule that led to the move. In addition, it weakens the responsible rule so that it will be less likely to apply in the future, and calls on the discrimination learning mechanism. This retrieves the last positive instance of the faulty rule and compares it to the current negative instance in search of differences. Since this heuristic retrieves the most recent positive instance of a rule, SAGE may lose information

[7] The traces matched by these rules are based on move information laid down by the various operators upon application; when a solution is found, SAGE chains back up the solution paths, marking move traces that fall on these paths.

TABLE II
Credit-Assignment Heuristics Based on Complete Solution Paths

ON-THE-PATH
If *move* led from *state* to *good-state*,
  and *state* lies along the solution path,
  and *good-state* lies along the solution path,
then retrieve the rule and instantiation that proposed *move*,
  and store that instantiation as a positive instance of the rule.

OFF-THE-PATH
If *move* led from *state* to *bad-state*,
  and *state* lies along the solution path,
  and *bad-state* does not lie along the solution path,
then retrieve the instantiation and rule that proposed *move*,
  as well as the last good instantiation of the same rule;
  weaken the rule and call on the discrimination process using
    the last good instantiation as the positive instance
    and the current instantiation as the negative instance.

when more than one correct move is made in a row. However, it would be impractical to compare all positive instances to all negative instances, and retrieving the last positive instance seems a plausible compromise.

SAGE's other credit assignment rules avoid this issue by more completely specifying the instances that should be compared. Table 3 presents three of the system's rules for assigning credit during the search process. The first of these, MARKED-BAD, matches when some operator trace has been labeled as undesirable, and some other operator trace originating from the same state has not been so labeled. In this case, SAGE retrieves the rule that fired in each case. If the same rule was applied in both situations, the discrimination mechanism is called with the first move as a negative instance and the second as a positive instance. In addition, the strength of the offending rule is decreased. If the good and bad moves were proposed by different rules, then the discrimination process cannot be applied, but the rule leading to the undesirable state is still weakened.

The remaining productions interact with MARKED-BAD, providing the labeling of states it requires for application. One of these, NOTE-LONGER, matches when the system reaches some state that was visited earlier. It marks the move that led to the revisited state as bad, and backs up, focusing attention on the state from which this move originated. Note that as this rule is stated, it will match against loops as well as against unnecessarily long paths, since a loop can be viewed as the longer of two paths to a state, where the shorter path has length zero. Thus, while these two situations can be separated conceptually, there is no reason to distinguish them as far as the implementation is concerned, as Anzai (1978) has done. The third rule in Table 3, DEAD-END, applies when a state is found from

TABLE III
Credit-Assignment Heuristics Based on Complete Solution Paths

MARKED-BAD
If *bad-state* is the current state,
  and *bad-move* led from *prior-state* to *bad-state*,
  and *bad-move* was undesirable,
  and *good move* led from *prior-state* to *good-state*,
  and *good-move* is not marked as undesirable,
then weaken the rule that proposed *bad-move*,
  and if the same rule proposed *good-move*,
    discriminate using the instantiation for *bad-move* as a negative instance,
    and using the instantiation for *good move* as a positive instance.

NOTE-LONGER
If *current-state* is the current state,
  and *move* led from *prior-state* to *current-state*,
  and *current-state* has been visited earlier,
then make *prior-state* the current state,
  and label *move* as undesirable.

DEAD-END
If *current-state* is the current state,
  and *move* led from *prior-state* to *current-state*,
  and no moves are possible from *current-state*
    that have not already been made,
then make *prior-state* the current state,
  and label *move* as undesirable.

which no moves can be made; it marks the move leading to that state as undesirable and shifts attention back to the previous state. We have not shown rules for noting illegal states or failure to make progress, since these must be implemented for specific domains individually. However, while the conditions of such rules differ from those of NOTE-LONGER and DEAD-END, their actions are identical, and they interact with MARKED-BAD in the same manner—by specifying undesirable moves, and letting this more general rule select better moves starting from the same state and evoking the discrimination process.

Learning Conditions Through Discrimination

As we have seen, once a strategy learning system has distinguished the positive from the negative instances of an operator, it must have some means of altering the conditions under which that operator is applied. In implementing SAGE.2, we chose to employ a discrimination-learning process that begins with overly general rules for proposing moves, and generates variants of these rules with additional conditions as experience is gained. This mech-

anism is presented with a single positive instance of a rule and a single negative instance of the same rule (in terms of their variable bindings), along with the state of working memory in each case. Bundy and Silver (1982) have called the variable bindings and state of memory during the good application, the *selection context,* and the variable bindings and state of memory during the faulty application, the *rejection context.* The discrimination process compares these two contexts, searching for differences which will allow it to distinguish one from the other.

The simplest form of difference involves a working memory element that was present in one context but not in the other. For example, if the trace of a previous move were present in the selection context but not in the rejection context, SAGE would create a variant of the overly general proposer that included this fact (with certain terms replaced by variables) as an additional condition. This variant would never match against the initial problem state, since no such trace would be present at the outset of the problem. Similarly, if an element were found to be present in the rejection context but not the selection context, this fact would be included as a *negated* condition in a variant on the original rule. The resulting rule would only match if this fact (or a similar one) were *not* present in memory.

More complex differences can be stated as *conjunctions* of elements that were present in one context but not in the other. Such differences are generated by a path-finding process that travels through symbols shared by working memory elements. An example will clarify the process. Table 4 presents both a selection context and a rejection context for the TOH rule. The first of these proposes the move from state S2 to state S4 shown in Figure 1, while the second leads to the move from State S3 to State S1. The two contexts are expressed in terms of the *bindings* between variables (in italics) and the symbols against which these variables matched. Thus, in the selection context, the variable *current-state* was bound to state S2, *disk* to disk-2, *current-peg* to peg-A, and *other-peg* to peg-B, leading SAGE to consider moving disk-2 from peg-A to peg-B. This move falls on the solution path, since it removes an obstruction (disk-2) from the largest disk (disk-3). In the rejection context, the variable *current-state* was bound to state S2, *disk* to disk-1, *current-peg* to peg-C, and *other-peg* to peg-A, leading to the action of moving disk-1 from peg-C to peg-A. Since this move takes the system back to the original state, it is undesirable.

Table 4 also presents the elements that were present in memory during each context[8] and from which new conditions are generated. The path-finding process starts from analogous symbols in the two sets of bindings (such

[8] Actually, SAGE considers only those elements which describe the current state or parents to the current state. Since other states considered in parallel can have no effect on the current move, they are ignored. Thus, the state of working memory after SAGE's initial moves can be found by taking the union of the two sets shown in Table 4, together with state-independent elements such as (peg-A is-a peg) and (disk-3 is-larger-than disk-1).

TABLE IV
Selection and Rejection Contexts for the TOH Rule

Selection Context	*Rejection Context*
Variable bindings	
disk—>disk-2	*disk*—>disk-1
current-peg—>peg-A	*current-peg*—>peg-C
other-peg—>peg-B	*other-peg*—>peg-A
current-state—>S2	*current-state*—>S3
Elements in working memory	
(move-1 led-from S1 to S2)	(move-2 led-from S1 to S3)
(move-1 was move disk-1 from peg-A to peg-C)	(move-2 was move disk-1 from peg-A to peg-B)
(disk-1 is -on peg-A in-state S1)	(disk-1 is-on peg-A in-state S1)
(disk-2 is -on peg-A in-state S1)	(disk-2 is-on peg-A in-state S1)
(disk-3 is -on peg-A in-state S1)	(disk-3 is-on peg-A in-state S1)
(disk-1 is -on peg-C in-state S2)	(disk-1 is-on peg-B in-state S3)
(disk-2 is -on peg-A in-state S2)	(disk-2 is-on peg-A in-state S3)
(disk-3 is -on peg-A in-state S2)	(disk-3 is-on peg-A in-state S3)

as disk-2 and disk-1), and attempts to find some path through the "good" elements that has no analogous path through the "bad" elements. Thus, if a path consisting of three elements was present in the selection context but not in the rejection context, a variant of the TOH rule would be based on this difference. This rule would include the three elements (with some constants replaced by variables) as positive conditions, so that it would match in the selection context but not the rejection context.

The path-finding process also searches for paths through the "bad" elements that have no analogous path through the "good" elements. Let us trace the method's discovery of such a difference in the elements in Table 4. Starting from the "bad" symbol S3 and the "good" symbol S2, the path-finding process considers bad elements and good elements that contain these symbols. Since both contexts contain an element indicating that an earlier move led to the current state—(move-2 led-from S1 to S3) and (move-1 led-from S1 to S2)—SAGE must extend these paths by considering additional elements in its search for differences. Thus, the analogous symbols move-2 (for the bad element) and move-1 (for the good element) are marked, and other elements containing these symbols are considered.[9]

For example, the bad path can be extended to include the element (move-2 was move disk-1 from peg-A to peg-B), since this also contains the symbol move-2. At first glance, there appears to be an analogous extension to the good path, using the element (move-1 was move disk-1 from peg-A to peg-C). However, note that the symbol disk-1 is already bound to the vari-

[9] Alternate paths are followed through other analogous symbols, such as peg-B and peg-C, peg-A and peg-A, and disk-1 and disk-1. Note that a symbol may be mapped onto itself, provided it occurs in analogous positions in the two elements.

able *disk* in the rejection context, while this is not true of disk-1 in the selection context. Similarly, peg-A is already bound to *other-peg* in the rejection context, while peg-C is unbound in the selection context. As a result, these two elements cannot be considered analogous, and the path-finding process has found a difference between the two contexts. Based on this difference, SAGE constructs the following variant:

TOH-1
If you have *disk* on *current-peg* in *current-state,*
  and you have some *other-peg* different from *current-peg,*
  and in *current-state* there is no *other-disk* on *current-peg* that is
    smaller than *disk,*
  and in *current-state* there is no *third-disk* on *other-peg* that is
    smaller than *disk,*
  and it is not the case that:
      prior-move led from *prior-state* to *current-state,* and
      prior-move was a move of *disk* from *other-peg* to *current-peg,*
then consider moving *disk* from *current-peg* to *other-peg.*

In addition to the original conditions, this rule (let us call it TOH-1) includes the elements (move-2 led-from S1 to 3) and (move-2 was move disk-1 from peg-A to peg-B), with the specific disk and pegs replaced by variables, embedded within a single negated condition. This rule will match if either of the negated conditions is matched, but not if *both* are matched simultaneously. As a result, it will still match against the selection context in Table 4 but not against the rejection context, which is precisely the goal of the discrimination method. Effectively, the new conditions prevent SAGE from reversing the last move it has made.

In some cases, only a single difference exists between the selection and rejection contexts. Winston (1970) has called these situations *near misses,* and they considerably simplify the learning process, since only one variant need be considered. Unfortunately, near misses seldom occur in the task of learning search heuristics, and a robust system must be able to handle the general case in which many differences exist. (Bundy and Silver [1982] have called these *far misses.*) SAGE deals with far misses by finding *all* paths to length N (in our runs, we have set N to 4) and constructing a variant based on each of these differences, some with new negated conditions like TOH-1, and others with new positive conditions. These conditions may involve descriptions of the current state, previous states, previous moves (as in TOH-1) or any combination of them. This leads to a significant search problem, and we will discuss the system's response to this problem later. However, let us first consider the notion of *difference* in more detail.

In searching for differences, the discrimination process must know which symbols should be used in determining significant differences and which differences should be ignored. For example, it makes sense to dis-

tinguish between working memory elements including the symbol *was* (which describe move traces) and those including *led-from* (which temporally connect these move traces), since they represent different types of information. In contrast, there is no reason to distinguish between internally generated symbols like the states S1 and S2, since there are only the "connecting tissue" used to link together the descriptions of each state and the temporal relations between states. Thus, when it is searching for differences, the discrimination routine never considers two elements as analogous if one contains *was* in the *N*th position and the other contains *led-from* in the same position. However, if one contains S1 and the other contains S2 in the same position, then the two elements will be considered analogous, unless some other (significant) difference exists, or unless one of these symbols has already been associated with some other symbol (such as S3) during the path-finding process. When a variant is constructed, significant terms are retained, while insignificant terms are replaced by variables in a consistent manner.

The case is less clear for the names of operators and their arguments. These symbols are not generated internally, yet if the variants are to retain any generality, some of them must be replaced by variables. Since one seldom wants to generalize across the operators themselves, SAGE treats operator names as significant. However, the arguments of these operators (e.g., objects and their positions) are treated as insignificant and are replaced by variables when a variant is constructed. Note that such decisions are not inherent aspects of the discrimination process; rather, they are *parameters* that are input to the learning method and can be easily modified. Later we will reconsider this decision and its implications for SAGE's learning behavior. For now, though, let us continue with our examination of the current system.

Directing Search Through the Rule Space

Most condition-finding methods, including the standard generalization approach and Mitchell's version space technique, find conditions that are held in common by all positive instances of a concept or operator. As a result, these methods are limited to acquiring *conjunctive* rules. In contrast, SAGE.2's discrimination process compares a *single* positive instance to a *single* negative instance. Because of this, it is capable of discovering *disjunctive* rules as well as conjunctive ones, and this ability can be very important in some task domains. In order to acquire disjunctive rules, the discrimination mechanism must search a larger space of rules than methods based on finding common features, and it must have some means of *directing* this search. For this reason, SAGE compares newly learned rules to those it has constructed earlier. If the new rule is identical to one of the existing vari-

ants, that variant is strengthened. Since the strength of a rule plays a major-role in whether it is selected for application, rules that have been learned more often will tend to be preferred. Thus, strength measures the *success rate* of each variant, and SAGE can be viewed as carrying out a *heuristic* search through the space of rules, selecting those rules that have proven most successful.

In domains involving only a single operator, it would be sufficient to simply strengthen variants whenever they were relearned, since they would eventually come to be preferred to the rules from which they were generated. However, some tasks involve multiple operators, and require that one of these operators be preferred to another. Given the role of strength in selecting rules, the natural response to such situations is to *weaken* rules when they propose an undesirable move. In addition to letting SAGE learn to prefer some operators over others, this strategy also decreases the chance that a faulty variant will be selected for application.

Although the combination of discrimination, strengthening, and weakening will eventually lead to useful search heuristics, many spurious variants will be created along the way. Since the matching process is a major component of programs stated as condition-action rules, we should briefly consider how SAGE handles the potential combinatorial explosion in the matcher. First, the system's condition-action rules are stored in a discrimination network that takes advantage of structure that is shared between rules. Since variants of the same proposer tend to be quite similar to one another, the expense involved in matching many variants of a rule is not much greater than that involved in matching the original rule. However, other components of the system (such as conflict resolution) are also slowed by the presence of many variants, so some further response is required. In addition, SAGE incorporates a thresholding principle. Variants below the threshold are not even incorporated in the discrimination network, and so have no effect on either the match process or conflict resolution (though they are retained for comparison with rules that are learned later). The strengths of new variants are set to a fraction of the rule from which they were spawned, and it is only when a variant comes to exceed its parent in strength that it is considered for application. Since few spurious variants ever become stronger than their parent rules, this method has worked quite well in directing SAGE's search through the space of proposers.

AN EXAMPLE OF SAGE.2 AT WORK

Our overview of SAGE.2 is now complete, but to give the reader a better understanding of how the system learns search strategies, we must examine its workings in specific domains. We now discuss SAGE's learning sequence

on the Tower of Hanoi puzzle, comparing its behavior when using only complete solution paths to its behavior when learning during the search process. We have chosen this task as our main example because it is familiar to many readers, and because most of the credit assignment heuristics discussed earlier come into play. However, since generality is an important criterion for judging learning systems, we will later examine the program's behavior in five other task domains in somewhat less detail.

Learning From Solution Paths

Since we have already discussed the Tower of Hanoi puzzle and its associated problem space, we shall begin by discussing the system's behavior on this problem when using the first credit assignment strategy—learning from complete solution paths. SAGE.2 was presented with a standard three-disk problem: The three disks were placed on a single peg, and the goal was to get all three disks on either of the other two pegs. In other words, the system started at State S1 in Figure 1 and was asked to reach either state S20 or S27 (or both of them). Starting with a breadth-first search strategy, the program first moved to states S2 and S3 and from there considered six moves: from S2 to S4, from S3 to S5, from S2 to S1, from S3 to S1, from S2 to S3, and from S3 to S2. While the system noted that the last four of these moves led to previously visited states, it did not attempt to learn from this knowledge, and simply abandoned these undesirable paths. From the two remaining states S4 and S5, SAGE moves to states S6, S7, S8, S9, S2, and S3. The last two of these moves were identified as loops, so only the first four states were retained for expansion. This search process continued until the program reached the two solution states, S20 and S27.

At this point, the complete solution path heuristic was applied. SAGE chained back up the solution path, marking the traces of moves that lay on the path. Once this was completed, it worked its way back down the marked path, letting the rules ON-THE-PATH and OFF-THE-PATH apply when they matched. The first of these circumstances occurred at states S2 and S3, when four moves were made that led off the solution path. One of these moves led to a loop from S2 back to S1, the original state. Comparing the good move from this point (from S2 to S4) to the bad move, SAGE's discrimination mechanism generated the variant TOH-1 that we considered earlier. The selection and rejection contexts for this learning situation were identical to those we have examined, except that SAGE compared two moves from state S2, rather than comparing one move from state S2 and another from state S3. As a result, the same differences were discovered, and the variant TOH-1 was constructed. The reader will recall that this rule contains a negated conjunction that prevents it from proposing a move that will reverse the move SAGE has just made. Some four other differences

were found, leading to four additional variants, but TOH-1 was the only rule that ever became strong enough to apply. An identical set of variants were created when the context for the move from S3 to S1 was compared to that for the move from S3 to S5, since these situations are completely symmetrical; this led each of the existing variants to be strengthened.

A different set of three variants resulted when the good move from S2 to S4 was compared to the bad move from S2 to S3 (and when the symmetrical moves were examined). In this case, the rule we are interested in is subtly different from the variant we described earlier:

TOH-2
If you have *disk* on *current-peg* in *current-state*,
  and you have some *other-peg* different from *current-peg*,
  and in *current-state* there is no *other-disk* on *current-peg* that is
    smaller than *disk*,
  and in *current-state* there is no *third-disk* on *other-peg* that is
    smaller than *disk*,
  and it is not the case that:
      prior-move led-from *prior-state* to *current-state*, and
      prior-move was a move of *disk* from *any-peg* to *current peg*,
then consider moving *disk* from *current-peg* to *other-peg*.

The new negated conjunction on this variant of TOH is nearly identical to that on TOH-1, but the difference is significant. TOH-2 states that it is acceptable to move a disk from its current peg to a new peg, provided on the previous move one did not move from *any* peg to the current peg. An example should help clarify this difference. Suppose we have disk-1 on peg-B, and since disk-1 is the smallest of the disks, we can move it to either Peg-A or peg-C without violating any of the task constraints. Further suppose that on the previous step, we moved disk-1 from peg-A to peg-B, so that TOH-1 will not propose moving the smallest disk back to peg-A (which would result in a loop). However, this variant would propose moving disk-1 to peg-C. In contrast, TOH-2 would not propose moving disk-1 to either peg-A or peg-C, since its negated condition forbids a move of the same disk twice in a row. Thus, the second variant is more conservative than the first, and as a result, it constrains the search process to a greater extent.

Upon comparing its moves from state S4 and S5, SAGE produced another set of variants on its initial proposer. When the discrimination process compared the context in which the desirable move from S4 to S6 was proposed to the context that led to the move from S4 to S7, some six new productions resulted. In this case, two of the rules are of interest:

TOH-3
If you have *disk* on *current-peg* in *current-state*,
  and you have some *other-peg* different from *current-peg*,
  and in *current-state* there is no *other-disk* on *current-peg* that is
    smaller than *disk*,

and in *current-state* there is no *third-disk* on *other-peg* that is
smaller than *disk*,
and it is not the case that:
prior-move led-from *prior-state* to *current-state*, and
earlier-move led-from *earlier-state* to *prior-state*, and
disk was on *other-peg* in *earlier-state*,
then consider moving *disk* from *current-peg* to *other-peg*.

and

TOH-4
If you have *disk* on *current-peg* in *current-state*,
and you have some *other-peg* different from *current-peg*,
and in *current-state* there is no *other-disk* on *current-peg* that is
smaller than *disk*,
and in *current-state* there is no *third-disk* on *other-peg* that is
smaller than *disk*,
and it is not the case that:
prior-move led-from *prior-state* to *current-state*, and
earlier-move led-from *earlier-state* to *prior-state*, and
earlier-move was a move of *disk* from *other-peg* to *current-peg*,
then consider moving *disk* from *current-peg* to *other-peg*.

In addition to helping direct search down profitable paths, these rules are interesting because they are syntactically different but semantically equivalent. The first refers to the *state* occupied two steps before the current state, while the second refers to the *move* made at that point. Yet both rules effectively keep one from moving a disk back to the position it was in two moves before, avoiding such nonoptimal moves as that from S4 to S7 and that from S5 to S8. Because of the structure of the task domain, these rules are always guaranteed to match together, and whenever one is learned, the other will also be learned. The possibility for syntactically distinct but semantically identical rules causes some extra search through the space of possible rules, but other than this, no harm is done.

So far, we have considered only the initial cases in which the above variants were constructed. However, each of these was relearned many times throughout the course of the first run. For example, the nonbackup variant TOH-1 was relearned and strengthened at each step along the way, since SAGE foolishly considered a backup at every point in its initial search tree. Similarly, the TOH-2 variant was strengthened whenever an attempt had been made to move the same disk twice in a row (other than simple backups). Thus, the bad moves from S2 to S3, from S6 to S7, and from S12 to S13 all resulted in an increase of this rule's strength, along with analogous faulty moves on the symmetrical path. Finally, the last two useful variants, TOH-3 and TOH-4, were learned whenever SAGE had considered moving a disk back to the position it had occupied two states earlier. Thus,

the bad moves from S4 to S7, from S10 to S13, and from S16 to S21 all reinforced these rules, increasing their likelihood of selection on the next run.

On the second run, the system's performance improved considerably, since TOH-1's strength had come to exceed that of the initial proposer. As a result, no backup moves were considered and the search process was considerably more directed. Unfortunately, neither this rule nor any of the other variants were sufficient by themselves to completely eliminate SAGE's search on the Tower of Hanoi problem, so more learning was required. Again the system chained back up its solution path, marking traces that led to the goal states, and began to compare the contexts of positive and negative instances in its search for useful variants. The learning process on this run was quite similar to the first, except that variants of TOH-1 were created (since only it had been applied), instead of variants of the original rule.

As one might expect, TOH-1 made exactly the same errors as its predecessor, except for the backup moves which its additional condition forbid. Thus, when at state S2, it considered moving to S3 as well as to S4, and when at state S4, it moved to S7 as well as to S6. As a result, the discrimination process generated variants of this production that were very similar to those created for its more general ancestor. When comparing the contexts that led from S2 to S4 and from S2 to S3, SAGE created a rule containing a "don't move the same disk twice in a row" condition, as well as the "don't back up" condition that was already present. Similarly, when comparing the moves from S4 to S6 and from S4 to S7, it constructed two variants with a "don't move a disk back where it was two states before" condition (again, these were syntactically different, but would always match against the same state of memory). These rules were relearned and strengthened at each of the points where their analogs were learned during the first run.

Since the new variants were more conservative than TOH-1, and since they had surpassed this rule in strength during the second learning run, they began to further direct the search process on the third pass. In fact, the "don't move the same disk twice in a row" variant (let us call it TOH-4) achieved the highest strength, so it was applied at each stage on this run. This rule avoided errors such as moving from S2 to S3 and from S6 to S7. However, it continued to make mistakes such as moving from S4 to S7, since it lacked the condition (contained in TOH-3) that would keep it from making such moves. Fortunately, once the solution paths had been found and the learning stage had begun, two (structurally different, but semantically equivalent) variants of TOH-4 were constructed that contained the "don't move a disk back to where it was two states before" condition. Once these two rules exceeded the strength of TOH-4 (as they had by the end of the run), SAGE had available to it a search heuristic that proposed moves lying on the solution path, but that ignored moves that would take it off that path. Indeed, when the system was presented the three-disk prob-

lem a fourth time, it successfully solved the problem without taking any false steps.

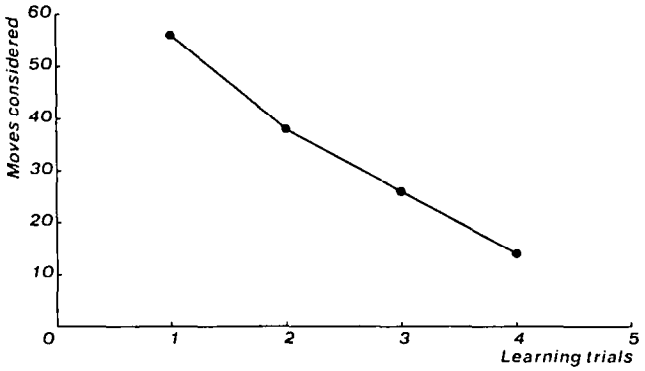

Figure 3. Learning curve for the three-disk Tower of Hanoi task.

Figure 3 presents the learning curve for SAGE.2 on the Tower of Hanoi task. The figure graphs the number of states considered during the search process against the number of times the problem had previously been attempted. As can be seen, the system shows a distinct improvement over time, until it eventually solves the task in the minimum number of steps. In addition, since the problem spaces for the four-disk and five-disk puzzles have the same basic structure as the simpler three-disk space, the learned heuristics were also useful in these more complex tasks. In fact, when presented with the standard four-disk and five-disk versions of the puzzle (in which all disks must be moved from one peg to a different peg), SAGE applied its heuristics to solve these problems without search as well. Thus, we can conclude that for this domain at least, the system is capable of transfer to scaled-up versions of a problem on which it has practiced.

While SAGE was able to transfer its acquired knowledge to other *standard* versions of the Tower of Hanoi task, the program would not have fared so well if it had been given a nonstandard problem. The heuristics that the system learns for this task are very good at directing search when all disks start on one peg and must be moved to another peg, but they are not adequate for moving from one arbitrary *configuration* to another. Later, we will have more to say about this type of transfer, and what would be required to accomplish it. However, let us first turn to the topic of learning while doing.

Learning While Doing

Although SAGE.2 is capable of learning from complete solution paths, it is not limited to this method. As we have seen, the system also includes heuris-

tics for learning from longer paths and loops, from dead ends, from illegal moves, and from a failure to make progress. The first two of these techniques[10] can be applied to the Tower of Hanoi puzzle to acquire search strategies identical to those described in the previous section. Let us consider this process of learning while doing and its relation to learning from complete solution paths.

As before, SAGE began the three-disk problem by carrying out a breadth-first search, moving from state S1 to states S2 and S3. Since these moves led to new states and since other moves could be made from them, none of the blame-assignment heuristics applied at this point. Since the two solution paths are symmetrical, we will focus on the left half of the space shown in Figure 1. From the state S2, three moves were possible—SAGE could move to S4, to S1, and to S3. The first of these was a new state, but S1 and S3 had been visited before. The move from S2 to S1 led to a loop, while the move from S1 through S2 to S3 was a longer path than that from S1 directly to S3. However, the NOTE-LONGER production does not make such distinctions (because it is concerned only with avoiding revisited states), so this rule applied, marking the moves from S2 to S1 and S3 as undesirable.

Given the information that these two moves should not have been made, the rule MARKED-BAD was applied to each in turn, calling on the discrimination mechanism. In both cases, it focused on the move from S2 to S4 as the positive instance, since this was the only move from S2 that was not labeled as an error. Upon comparing this move to the one from S2 to S1, SAGE constructed the variant TOH-1 that we saw before, along with four other variant productions that never became strong enough to apply. When the move from S2 to S4 was compared to that from S2 to S3, the variant TOH-2 was created (along with two other rules). Thus, up to this point, SAGE had assigned credit in precisely the same manner that it did when the complete solution path was available.

Next, having abandoned the revisited states, SAGE applied its initial proposer (which was still stronger than any of the variants) to the state S4. From this position, three moves were again possible—from S4 to S6, from S4 to S2, and from S4 to S7. The second of these led back to the previous state, and was labeled as undesirable by NOTE-LONGER. Given this judgment, MARKED-BAD applied twice, comparing this move both to that from S4 to S6 and to that from S4 to S7, since neither had been marked as bad. In both cases, the variant TOH-1 was recreated and strengthened, along with a number of other rules. Since SAGE did not yet have any reason

[10] In fact, the rules NOTE-LONGER and DEAD-END were used even in the described run in which credit was assigned after a solution had been found. However, their role in this run was only to tell SAGE when it had reached untenable positions, so the system could abandon search down certain paths and focus on others. Because the production MARKED-BAD was not present, the program could not learn using the information added to memory by these rules.

to suspect that the move from S4 to S7 was undesirable, it considered moves from both this state and from S6, which lay on the solution path.

Three moves were possible from S6, and all were carried out; these included a move from S6 to S10, from S6 to S4, and from S6 to S7. The last two of these operations led to revisited states, so NOTE-LONGER was applied in each case. MARKED-BAD compared each of these moves to that from S6 to S10, regenerating TOH-1 in one instance and TOH-2 in the other, along with a number of additional variants. Three moves could also be made from S7, to the states S6, S4, and S8. However, each of these states had been visited before, the last from the symmetrical search in the right side of the space. NOTE-LONGER was applied and marked each of the moves from S7 as undesirable, but since there were no good moves originating from S7 with which they could be compared, MARKED-BAD could not be applied. Meanwhile, NOTE-LONGER had also refocused SAGE's attention on S7, marking it as one of the states currently under consideration for expansion. Since no other moves could be made from this state, the rule DEAD-END applied, calling on the discrimination routine to compare the good move from S4 to S6 to the recently determined bad move. Two of the resulting variants were TOH-3 and TOH-4, which avoid moving a disk back to the position it occupied two states earlier.

By this point, SAGE's credit assignment had begun to lose ground to the strategy of learning from complete solution paths. Although NOTE-LONGER continued to notice revisited states and to lead MARKED-BAD to strengthen both TOH-1 and TOH-2, the dead-end noticing rule never had another chance to apply. As a result, the moves from S10 to S13 and from S16 to S21 were never classified as undesirable, and the two variants TOH-3 and TOH-4 were not relearned until the complete solution path was marked, and ON-PATH and OFF-PATH came into the picture. This did eventually occur, and the resulting events were identical to those described in the previous section, save that many of the variants already existed, and so by the end of the run they were considerably stronger than in the other case. After this, SAGE was given a second chance to solve the three-disk task, and events followed much the same route, except that backups were missing, so NOTE-LONGER was applied much less often. By the fifth run, the system was able to solve the problem without search, and to transfer its expertise to the four-disk puzzle. The learning curve for these runs was very similar to that shown in Figure 3. However, slightly less search was carried out in the early runs, since the useful variants were able to mask their predecessors before the run was complete.

The Importance of Goals

In our treatment of the Tower of Hanoi puzzle, we assumed two goal states and two symmetrical solution paths to these goals. It is much more common

to formulate the problem with a single goal peg, resulting in only one optimal solution path, and our use of multiple goals deserves some discussion. In the early stages of constructing SAGE.2, we made two design decisions that led us to state the Tower of Hanoi puzzle as we have done. First, we decided to treat the arguments of operators as insignificant during the discrimination process, as we described earlier. As a result, the system has difficulty in learning heuristics for moving disks toward one peg rather than another, and we avoided this issue by including two goal pegs. If we had chosen instead to treat pegs as significant symbols, SAGE would have learned more specific rules, but at least the system would have been able to acquire heuristics for moving disks to a specific peg. However, a more general and attractive alternative presents itself.

The second design decision involved assuming a procedural representation for the goal state, rather than a declarative one. The reader will recall that SAGE includes a production for recognizing when it has solved a problem, and which stops the search process when this occurs. Since goal information is not available for inspection by the discrimination mechanism, it cannot discover conditions that refer to the goal state. As a result, the search heuristics it learns are incapable of directing search down different paths depending on the goal. Note that this is not a limitation of the discrimination method itself, but is rather a limitation in the information accessible to the learning system. If we had chosen to include explicit information about the goal state in working memory, SAGE should have been able to learn rules that would move toward a single goal and still treat the arguments of its operators (such as pegs and disks) as insignificant symbols.

In addition, this approach opens the way for learning heuristics for solving nonstandard versions of the Tower of Hanoi puzzle, in which both the initial and goal states are arbitrary configurations of disks. Once the discrimination method has access to the goal state, it might well be able to acquire rules that would transfer between different initial and goal states, leading to a much more robust system. Although we have not yet tested SAGE in this manner on the Tower of Hanoi, we will later examine another task in which this approach does lead to the predicted forms of transfer. Since goals are so obviously important to problem solving, it may seem odd that we did not include declarative knowledge of goals at the outset of our research. Such judgments are all too easily made with the aid of hindsight. In defense, we can only note that very little of the other work on learning search heuristics deals with goals in this manner, so that SAGE is far from alone on this dimension.

APPLYING SAGE.2 TO OTHER DOMAINS

One important dimension on which AI systems are judged is their generality, and the most obvious test of a program's generality is to apply it to a

number of different domains. In this section, we summarize SAGE.2's behavior on five additional tasks. Some of these are puzzles similar to the Tower of Hanoi task, but others have quite different characteristics. In each case, we describe the problem or class of problems, consider the rules the program learns in the domain, and discuss the types of transfer that occur. After this, we examine the generality of the individual learning heuristics employed by the system.

The Slide-Jump Puzzle

In the Slide-Jump puzzle, one is presented with N quarters and N nickels placed in a row. The quarters are on the left, the nickels are on the right, and the two sets of coins are separated by a blank space. Legal moves include *sliding* into a blank space or *jumping* over another coin into a blank space. In addition, quarters can be moved only to the right, while nickels can be moved only to the left. The goal is to exchange the positions of the quarters and the nickels, so that the former occur on the right side of the blank and the latter occur on the left. For instance, given the initial state Q Q Q - N N N, one would attempt to generate the goal state N N N - Q Q Q. Like the Tower of Hanoi problem, the Slide-Jump puzzle has a relatively small search space, yet it is quite difficult for human problem solvers to master. Also like the Tower of Hanoi, it has two symmetric solution paths; however, since moves are not reversible, loops do not come into play in this task.

SAGE.2 was initially presented with the four-coin version of this puzzle, in which the positions of two quarters and two nickels must be exchanged. The program was given two initial proposers—one for suggesting slide moves and the other for suggesting jumps. After an initial breadth-first search in which both optimal solutions were found, the system attempted to learn from these paths. After some three runs through the problem, SAGE had generated (and sufficiently strengthened) the following variant of the initial slide rule:

SLIDE-1
If a *type-of-coin* is in *current-position* in *current state,*
  and *adjacent-position* is blank in *current-state,*
  and *adjacent-position* is to the *left-or-right* of *current-position,*
  and *type-of-coin* can move to the *left-or-right,*
  [and *prior-move* led-from *prior-state* to *current-state,*]
  [and *prior-move* was a jump of *type-of-coin* from *adjacent-position*
    to *other-position,*]
then consider sliding *type-of-coin* from *current-positions,* to *adjacent-position.*

This rule contains two conditions (enclosed in brackets) that were not present in the original slide-proposing production. These conditions allow the

variant to propose sliding a coin only if another coin of the same type was just jumped from the adjacent position. Five other variants of the original slide rule were constructed and contributed to directing the search process, while some 14 variants were based on spurious features of the problem, and were not learned enough times to affect behavior. One variant of the jump rule was also constructed, which avoided jumping one coin over another of the same type (which leads to to a dead end). However, this rule was learned only once before a variant of the slide rule caused SAGE to avoid this particular error.

In the learning-while-doing runs, the system proceeded in a very similar manner, except that some credit and blame was assigned during the search process. In this task, two credit-assignment heuristics contributed to learning. The DEAD-END rule produced a variant that avoided sliding the same type of coin twice in a row, while NOTE-LONGER generated the jump variant already mentioned. When SAGE was presented with the six-coin Slide-Jump puzzle, it successfully solved this problem without search, again indicating that the system can handle scaled-up transfer. Although the normal statement of the puzzle does not allow reversible moves, alternate initial and goal states can be formulated if they are allowed. However, in its current form, the program would not have been able to transfer its expertise to an arbitrary problem of this type, for the same reasons as the Tower of Hanoi version.

Tiles and Squares

Ohlsson (1982) has described the Tiles and Squares puzzle, in which one is presented with N tiles and $N+1$ squares on which they are placed. Each square is numbered from 1 to $N+1$, and each tile is labeled with a unique letter. Only one legal move is possible: moving a tile from its current position to the blank square. The goal is simple: Get all the tiles from the initial positions to some explicitly specified end position. For example, the initial configuration might be B C * A, while the goal configuration might be A * C B. Since *any* tile may be moved into the blank space, the moves are much less constrained than in most puzzles. One of the interesting features of this task is that while the branching factor of the search space is quite high (3 for three-tile tasks, 4 for four-tile tasks, etc.), two simple heuristics are sufficient to avoid search entirely. Indeed, one might even question whether the task is challenging enough to be called a puzzle. We have included it here primarily to clarify SAGE's ability to acquire disjunctive rules.

* The location of the asterisk between the A B C letter patterns indicates blank space positioning.

SAGE.2 was presented with this problem, as well as a single rule for proposing legal moves. Based on the two optimal solution paths it discovered for this task, the system generated (and sufficiently strengthened) seven variants for directing the search process, along with some 73 less useful rules. Two of the useful variants[11] may be paraphrased as:

TS-1
If you have a *tile* on *current-square* in *current-state,*
  and *other-square* is blank in *current-state,*
  [and in the final goal you want *tile* in *other-square,*]
then consider moving *tile* from *current-square* to *other-square.*

and

TS-2
If you have a *tile* on *current-square* in *current state,*
  and *other-square* is blank in *current-state,*
  [and in the final goal you want *other-tile* in *current-square,*]
  [and it is not the case that:
    prior-move led-from *prior-state* to *current-state,* and
    prior-move was a move of *tile* from *other-square* to *current-square,*]
then consider moving *tile* from *current-square* to *other-square.*

Note that these rules are *disjunctive* in that they cover different situations that arise in the problem. For example, the first variant is useful in suggesting that C be moved to the third position at the outset of the above problem, leading to the state B * C A. Once this has been done, the second rule is useful in proposing that either B or A be moved into the second square, leading to the states * B C A and B A C *. At this point the first rule again comes into play, proposing the move of A into square 1 or B into square 4, and finally, this same rule proposes moving B to 4 or A to 1, reaching the goal state. The point here is that neither of the above heuristics is sufficient to completely direct the search process by itself, but taken together they eliminate search. Thus, the ability of SAGE's discrimination process to consider disjunctive heuristics shows its potential in the Tiles and Squares puzzle.

Another interesting characteristic of this problem is that SAGE incorporated information about the goal state in the conditions it discovered. This was possible because the goal description was present in working memory, and so was considered during the condition-finding process. As a result, the heuristics the system learned from the above problem can be applied not only to more complex problems with longer solution paths, but to other problems in the same space with differing initial and goal states. Thus, SAGE's behavior on the Tiles and Squares task shows that the system

[11] The other five useful variants were semantically equivalent to TS-2 and proposed the same moves in all cases.

is capable of acquiring goal-sensitive heuristics, as we proposed earlier, provided information about the goal state is present in working memory.

In addition to learning from complete solution paths, the credit assignment heuristic for noting loops and longer paths was also applicable to this domain. The detection of longer paths led to TS-1, the first variant, which moves a tile into its goal square whenever possible. Similarly, the detection of loops led to an initial version of TS-2 that contained only the no-backup condition. However, none of the learning while doing heuristics were sufficient to learn the TS-2 condition "in the final goal you want *other-tile* in *current-square*." This was due to the fact that, whenever TS-2 is applicable, there are a number of equally good moves that lie along optimal solution paths. Moreover, other than backtracking moves, *all* of the legal moves in such situations are equally desirable. Since the learning while doing rule MARKED-BAD only compares instances originating from the same state, and since there are no bad moves from such states, SAGE can never master the complete form of TS-2 during the search process. As a result, the system fell back on its complete solution path strategy to learn the final version of this variant.

The Mattress Factory Puzzle

Like the Slide-Jump problem, the Mattress Factory puzzle requires two operators for moving through its search space. In this task, one is told that N employees are working at a mattress factory. Due to losses, the factory must be closed down, and so all the workers must be fired. However, union regulations require that hiring and firing follow certain rules. The least senior worker may be hired or fired at any time; this corresponds to the first operator. However, other workers may only be hired or fired if the person directly below them in seniority is currently employed, and furthermore, provided that no other person below them is also employed. This complex rule corresponds to the second operator. Since each of these operators is reversible, one can always immediately undo an action that was just taken. Thus, this task shares an abundance of possible loop moves with the Tower of Hanoi. Although this problem has an even smaller space than the Tower of Hanoi, it also gives human problem solvers considerable difficulty. Cahn (1977) has studied human learning on the Mattress Factory problem.

SAGE.2 was initially presented with the three-person version of the problem, along with rules for proposing the two types of moves described above. After finding the single solution path, it generated and sufficiently strengthened a straightforward variant of the original lowest worker rule:

MF-1
If you have a *worker* with *current-status* in *current-state*,
and *worker* is not senior to any *other-worker*,

and *current-status* is the opposite of *other-status*,
[and it is not the case that:
prior-move led-from *prior-state* to *current-state*, and
prior-move was a change of *worker* from *other-status* to
current-status,]
then consider changing *worker* from *current-status* to *other-status*.

In this production, the variables *current-status* and *other-status* match against the possible states in which a worker can find himself—either *employed* or *unemployed*. The additional negated conjunction on this rule simply prevents one from undoing the previous move. Together with a similar variant of the second operator, this production is nearly sufficient for directing search on the Mattress Factory puzzle.

However, one additional piece of information is required. If one avoids backups, then only two legal paths can be traversed in this problem space, and these paths are entirely determined by whether one initially fires the least senior worker or his immediate superior. In the three-worker problem, the correct choice is to fire the lowest person. SAGE acquires this strategy by weakening the variant on the second operator, so that the MF-1 rule shown above is preferred. This strategy transfers to scaled-up problems concerning five, seven, or any odd number of workers, but not to problems concerning even numbers of employees. If we had been willing to add to SAGE's memory the parity of the number of workers, this could conceivably have been learned as a condition across problem types.

A significant feature of this class of problems is that learning from complete solution paths does not provide any more accurate credit assignment information than does learning while doing. In the latter case, the majority of credit is assigned by the NOTE-LONGER rule in response to the large number of loop moves that are made. In addition, although SAGE explores both of the paths leading from the initial state, one of these eventually leads to a dead end. At this point, the DEAD-END rule chains back up the search tree, marking each state along the way as undesirable. However, no learning can occur until it reaches the two moves made from the initial state, since it requires both a positive and negative instance before learning can occur. Since different operators were applied at this point, no discriminations can result, but the rule proposing the move down the dead-end path is weakened, giving preference to the other operator.

Algebra

We have also presented SAGE.2 with algebra problems in one variable, such as $4x - 5 = 3$. The goal here is to simplify the expression, arriving at an equation with the variable on one side and a number on the other, such as $x = 2$. For this domain, the system was given a single operator for adding, sub-

tracting, multiplying, or dividing both sides of an equation by the same number. Moreover, the initial proposer for this operator required that any numeric arguments to these functions occur somewhere within the current expression. In addition, SAGE was provided with a domain-specific credit assignment heuristic; this informed the program that expressions which were not simpler in form than the previous expression were no closer to the goal, and so were undesirable.

Given this information, the system's behavior when learning while doing was identical to that when learning from complete solution paths. During both runs, SAGE arrived at a variant of its original proposer that would always direct it to an optimal solution. This rule can be stated as:

ALGEBRA-1
If you see a *number* as the argument of *function* in *current-state,*
  and *other-function* is a function,
  [and *function* is the inverse of *other-function,*]
  [and *function* occurs at the top level of the expression in *current-state,*]
then consider applying *other-function* to both sides with *number* as its argument.

This production contains two conditions beyond those in the initial rule, both of which are enclosed in brackets. The first of these constrains attention to functions that are the inverses of functions occurring in the expression. For example, given the expression $4x - 5 = 3$, ALGEBRA-1 would consider adding a number (since addition is the inverse of subtraction), or dividing by a number (since division is the inverse of multiplication), but not subtracting or multiplying. The second condition further constrains the function that is selected. SAGE represents such expressions as trees or list structures with forms like (= (- (* 4 x) 5) 3). Since subtraction occurs at the top level of the structure, it would bind against the variable *function,* so that adding 5 to both sides would be suggested.

Since algebra problems such as the above always assume similar goals, transfer to problems with different goals is not appropriate for this domain. However, scaled-up transfer is possible, and the variant SAGE generated for the above problem can be used to solve more complex problems, such as $(3\ (x + 1) - 5)\ /2 = 2$. Obviously, it can also be used to solve different problems of the same complexity involving different functions. In principle, we could have given SAGE four different proposers at the outset—one for addition, one for subtraction, and so forth. If we had not given the system information about the inverses of functions, it would still have been able to learn not to add unless subtraction occurred in an expression, and analogous rules with similar conditions. However, given a problem like $4x - 5 = 3$ on which to practice, the system would then have only partial transfer to a problem like $2x + 1 = 7$, in which there occurred only one of the operators

with which it had experience. This form of transfer is similar to that studied by Mitchell, et al. (1983) in their work on symbolic integration.

Seriation

Seriation behavior has been widely studied by developmental psychologists, starting with Piaget (1952), and production system models of children's behavior on this task have been constructed by Young (1976) and by Baylor, Gascon, Lemoyne, and Pother (1973). In one version of this task, the child is presented with a set of blocks in a pile and is asked to line them up in order of descending height (say from left to right). As simple as this may sound, young children have considerable difficulty with this sorting task, and many adults do not solve the problem very efficiently. Since this class of problems was somewhat different from the others SAGE had been given, we felt it would be useful to include it in our tests of the system.

In this case, the program was given a single operator for moving a block from the pile to the end of the current line (or to the first position in the line, if none existed). Also, SAGE was given a domain-specific rule for determining illegal states. This stated that if a taller block had been set to the right of a shorter block, the move that led to this state was undesirable. For example, suppose the system were presented with four blocks—A, B, C, and D—where A is the tallest and D is the shortest. Further suppose that on the first move, SAGE moved D into the line. On the next move, the program could move any of A, B, or C next to D, but each of these moves would immediately be classified as illegal.

SAGE.2 was presented with four blocks and given the goal of ordering them according to height. Learning from complete solution paths (and using only the illegal move detector to constrain the initial search), the system generated 1 useful variant, along with some 67 others. The useful production exceeded the original rule in strength after a single learning run, and led to the perfect behavior on the second time through the problem; it can be stated as:

SERIATE-1
If you have a *block* in the pile in *current-state,*
  [and it is not the case that:
    there is some *other-block* in the pile in *current-state,*
    and *other-block* is taller than *block,*]
then consider moving *block* to the end of the line.

This production contains a single new condition that is stated as a negated conjunction. Effectively, it says that one should move a block only if there is no other block in the pile that is taller than that piece. This constraint is related to conditions in the illegal state detector, since the SERIATE-1

variant will never place a taller block to the right of a shorter one. However, one can imagine a rule that would never propose illegal moves, and yet would still start off down the wrong path, say by placing the smallest block in the line first. Such a variant was generated during the seriation run, but did not become as strong as the rule shown above. Thus, while SERIATE-1 incorporates the test for illegal states in its condition side, it incorporates look-ahead information as well, so that it avoids moves that lead to dead ends.

SAGE.2 was also capable of learning during the initial search on this task. In addition to the rule for noting illegal states, the DEAD-END heuristic also came into play. Consider again our example in which block D is placed first in the line. In this situation, the system attempted moving each of A, B, and C next to the smallest block, and each move was marked as illegal. However, since no other moves were possible from this state, the DEAD-END rule applied, marking the initial D move as undesirable. Since the three other moves considered at the outset were still acceptable (the B and C moves did not lead to dead ends until later), the D move was compared to each of these moves by MARKED-BAD. The resulting call on discrimination led to the SERIATE-1 rule shown above. Later dead ends led to similar comparisons, and this rule was strengthened, until it came to efficiently direct the search process before an initial solution had been found.

DISCUSSION

Now that we have examined SAGE and its behavior on a number of tasks, we can begin to evaluate the program. In the case of a learning system, one of the most important dimensions is generality. One way to test a system's generality is to run it in a number of domains, and as we have seen, SAGE fares well on this criterion. However, one could in principle construct a program that employed one heuristic for one domain, a different heuristic for another domain, and so forth. In other words, one must also test the *components* of a system for generality. On this dimension, SAGE's discrimination/strengthening strategy passes with flying colors, since it played a central role in each of the runs we have described. However, the situation with respect to the credit-assignment heuristics is more complex, so let us consider it in more detail.

Table 5 presents the five credit assignment rules used in SAGE.2, along with the six task domains in which the system was tested. As can be seen from the table, and as has been apparent throughout the paper, the complete solution path heuristic is very general, and was (or could have been) applied on each of the tasks. The other heuristics were less useful, but still showed

TABLE V
Generality of SAGE.2's Credit Assignment Heuristics

	Solution	*Longer*	*Deadends*	*Illegal*	*No Progress*
Tower of Hanoi	X	X	X		
Slide-jump	X	X	X		
Tiles and squares	X	X			
Mattress factory	X	X	X		
Algebra	X				X
Seriation	X		X	X	

evidence of generality. Both the loop move/longer path rule and the dead-end rule led to learning in four of the six problem classes.

The illegal state detector was stated in a domain-specific manner and was used only in the seriation task. However, one can imagine versions of the Tower of Hanoi, Mattress Factory, and Slide-Jump puzzles in which the conditions for legal moves must be learned along with the conditions for good moves. It might even be possible to state these constraints as elements in SAGE's working memory, so that a quite general illegal state detector could be implemented. Finally, the no-progress rule was used only in the algebra domain, but one can imagine a version of SAGE that always computed the distance between the current state and the goal state, and a very general no-progress heuristic that matched off the results of this computation.

Another issue relates to the *form* of the acquired heuristics. As we have seen, the discrimination approach is in principle capable of learning disjunctive rules, and this potential proved useful on the Titles and Squares task. Since disjunctive heuristics are likely to occur in a significant fraction of task domains, the ability to acquire them is certainly desirable, and SAGE fares well on this count. On the other hand, we found that on most tasks, SAGE was not able to learn heuristics that incorporated information about the goal state. Such rules are important, since they would let the system to transfer its acquired expertise to problems with different initial and goal states than those on which it practiced.

The one area in which the system did achieve such transfer was the Tiles and Squares problem, and the key is this case was the *explicit* representation in working memory of the goal state toward which the system was working. Since this information was available for inspection by the discrimination mechanism, it could be included in the conditions on variants spawned by this process. As a result, variants containing such conditions could direct the search in different directions, depending on the particular goal that was being sought. Presumably, before SAGE can be expected to manage similar transfers for other domains, its representation for these tasks must be augmented to include explicit representations of their goal

states. Whether such an addition will be sufficient or merely necessary is a question that can best be answered experimentally.

A second natural extension relates to the search stragegy that SAGE employs. Many problems (such as winning a chess game) are so complex that they can only be solved by breaking the task up into manageable components. One such approach involves setting up *subgoals,* each of which must be solved before the supergoal is accomplished. If SAGE's search control were augmented to allow the introduction of subgoals, then the heuristic for assigning credit based on complete solution paths could undergo an important but subtle alteration. Rather than requiring solutions to an entire problem, the method could be applied whenever a particular subgoal had been achieved. Variants learned from this path would be specific to that subgoal; that is, they would include a description of the current subgoal as an extra condition, in addition to the other conditions found through discrimination. Even if SAGE later determined that this subgoal was not particularly desirable in the current context, the rules that had been learned might still prove useful in satisfying the subgoal in some other situation at a later date. This approach would also require the system to learn the conditions under which various subgoals should be set, but this could be handled by the existing mechanisms for learning the conditions on operators.

In summary, the existing version of SAGE has a number of desirable features, but our understanding of the strategy learning process is far from complete, and more work remains to be done. In our future research, we plan to restructure the system's problem solving and learning methods to take advantage of information about goals. In addition, SAGE has so far been tested only on problems with relatively small search spaces, and we are now ready to explore the system's behavior on more complex tasks. Undoubtedly, our experiences in these domains will lead to additional insights into SAGE's limitations, and to further revisions that, hopefully, will lead to a more powerful and robust system for learning search heuristics.

REFERENCES

Anderson, J. R. (1976). *Language, memory, and thought.* Hillsdale, NJ: Erlbaum.

Anderson, J. R., & Kline, P. J. (1979). A learning system and its psychological implications. *Proceedings of the Sixth International Joint Conference on Artificial Intelligence* (pp. 16-21). Tokyo, Japan.

Anderson, J. R. (1981). Tuning the search of the problem space for geometry proofs. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence.*

Anzai, Y. (1978). Learning strategies by computer. *Proceedings of the Canadian Society for Computational Studies of Intelligence* (pp. 181-190). Toronto, Ontario, Canada.

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). Hamburg, West Germany.

Bundy, A., & Silver, B. (1982). A critical survey of rule learning programs. *Proceedings of the European Conference on Artificial Intelligence* (pp. 151-157). Orsay, France.

Cahn, A. (1977). *A puzzle with a goal recursive strategy: The mattress factory.* Unpublished master's thesis, Department of Psychology, Carnegie-Mellon University, Pittsburgh, PA.

Carbonell, J. G. (1983). Learning by analogy: Formulating and generalizing plans from past experience. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach.* Palo Alto, CA: Tioga Press.

Hagert, G. (1982). On procedural learning and its relation to memory and attention. *Proceedings of the European Conference on Artificial Intelligence* (pp. 261-266). Orsay, France.

Hayes-Roth, F., & McDermott, J. (1978). An interference matching technique for inducing abstractions. *Communications of the ACM, 21,* 401-410.

Iba, G. A. (1979). *Learning disjunctive concepts from examples.* Unpublished master's thesis. Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Keller, R. M. (1982). *A survey of research in strategy acquisition.* (Tech. Rep. DCS-TR-115). Dept. of Computer Science, Rutgers University, New Brunswick, NJ.

Korf, R. E. (1982). A program that learns to solve Rubik's cube. *Proceedings of the National Conference on Artificial Intelligence* (pp. 164-167). Pittsburgh, PA.

Langley, P., Neches, R., Neves, D., & Anzai Y. (1980). A domain-independent framework for learning procedures. *International Journal of Policy Analysis and Information Systems, 4,* 163-197.

Langley, P., & Neches, R. (1981). *Prism user's manual.* (Tech. Rep.) Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, PA.

Langley, P. (1982a). Strategy acquisition governed by experimentation. *Proceedings of the European Conference on Artificial Intelligence* (pp. 171-176). Orsay, France.

Langley, P. (1982b). Language acquisition through error recovery. *Cognition and Brain Theory, 5,* 211-255.

Langley, P. (1983). Learning search strategies through discrimination. *International Journal of Man-Machine Studies, 18,* 513-541.

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). Cambridge, MA.

Mitchell, T. M., Utgoff, P., & Banerji, R. B. (1983). Learning problem solving heuristics by experimentation. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine learning: An artificial intelligence approach,* Palo Alto, CA: Tioga Press.

Neches, R. (1981). A computational formalism for heuristic procedure modification. *Proceedings of the Seventh International Joint Conference on Artificial Intelligence* (pp. 283-288). Vancouver, B.C., Canada

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). Toronto, Ontario, Canada.

Newell, A., & Simon, H. A. (1972). *Human problem solving.* Englewood Cliffs, NJ: Prentice-Hall.

Nilsson, N. J. (1971). *Problem solving methods in artificial intelligence.* New York: McGraw-Hill.

Ohlsson, S. (1982). *Transfer of training in procedural learning: A matter of conjectures and refutations?* (Tech. Rep. 13). Computing Science Department, University of Uppsala, Uppsala, Sweden.

Ohlsson, S. (1983). A constrained mechanism for procedural learning. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (pp. 426–428). Karlsruhe, West Germany.

Piaget, J. (1952). *The child's conception of number.* New York: Humanities Press.

Rendell, L. A. (1983). A learning system which accommodates feature interactions. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence* (pp. 469 472). Karlsruhe, West Germany.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development, 3,* 210–229.

Simon, H. A., & Reed, S. K. (1976). Modeling strategy shifts in a problem-solving task. *Cognitive Psychology, 8,* 86–97.

Sleeman, D., Langley, P., & Mitchell, T. (1982). Learning from solution paths: An approach to the credit assignment problem. *AI Magazine, 3,* 48–52.

Vere, S. A. (1975). Induction of concepts in the predicate calculus. *Proceedings of the Fourth International Joint Conference on Artificial Intelligence* (pp. 281–287). Tbilisi, Union of the Soviet Socialist Republics.

Winston, P. H. (1970). *Learning structural descriptions from examples.* (Tech. Rep. AI-TR-231). Massachusetts Institute of Technology, Cambridge, MA.

Winston, P. H. (1975). Learning structural descriptions from examples. In P. H. Winston (Ed.), *The Psychology of Computer Vision.* New York: McGraw-Hill.

Young, R. M. (1976). *Seriation by children: An artificial intelligence analysis of a Piagetian task.* Basel, Switzerland: Birkhauser.