CMU-CS-85-154

Chunking in Soar: The Anatomy of a General Learning Mechanism

John E. Laird
Intelligent Systems Laboratory
Xerox Palo Alto Research Center
3333 Coyote Hill Rd.
Palo Alto, CA 94304

Paul S. Rosenbloom
Departments of Computer Science and Psychology
Stanford University
Stanford, CA 94305

Allen Newell
Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

August 1985

Table of Contents

1. Introduction

The goal of the *Soar* project is to build a system capable of general intelligent behavior. We seek to understand what mechanisms are necessary for intelligent behavior, whether they are adequate for a wide range of tasks — including search-intensive tasks, knowledge-intensive tasks, and algorithmic tasks — and how they work together to form a general cognitive architecture. One necessary component of such an architecture, and the one on which we focus in this paper, is a general learning mechanism. Intuitively, a general learning mechanism should be capable of learning all that needs to be learned. To be a bit more precise, assume that we have a general performance system capable of solving any problem in a broad set of domains. Then, a general learning mechanism for that performance system would possess the following three properties.[1]

- *Task generality.* It can improve the system's performance on all of the tasks in the domains. The scope of the learning component should be the same as that of the performance component.

- *Knowledge generality.* It can base its improvements on any knowledge available about the domain. This knowledge can be in the form of examples, instructions, hints, its own experience, etc.

- *Aspect generality.* It can improve all aspects of the system. Otherwise there would be a *wandering-bottleneck problem* (Mitchell, 1983), in which those aspects not open to improvement would come to dominate the overall performance effort of the system.

These properties relate to the scope of the learning, but they say nothing concerning the generality and effectiveness of what is learned. Therefore we add a fourth property.

- *Transfer of learning.* What is learned in one situation will be used in other situations to improve performance. It is through the transfer of learned material that *generalization*, as it is usually studied in artificial intelligence, reveals itself in a learning problem solver.

Generality thus plays two roles in a general learning mechanism: in the scope of application of the mechanism and the generality of what it learns.

There are many possible organizations for a general learning mechanism, each with different behavior and implications. Some of the possibilities that have been investigated within AI and psychology include:

- *A Multistrategy Learner.* Given the wide variety of learning mechanisms currently being investigated in AI and psychology, one obvious way to achieve a general learner is to build a system containing a combination of these mechanisms. The best example of this to date is Anderson's (1983a) ACT* system which contains six learning mechanisms.

- *A Deliberate Learner.* Given the breadth required of a general learning mechanism, a natural way to build one is as a problem solver that deliberately devises modifications that will improve performance. The modifications are usually based on analyses of the tasks to be accomplished,

---

[1] These properties are related to, but not isomorphic with, the three dimensions of variation of learning mechanisms described in Carbonell, Michalski, and Mitchell (1983) — application domain, underlying learning strategy, and representation of knowledge.

the structure of the problem solver, and the system's performance on the tasks. Sometimes this problem solving is done by the performance system itself, as in Lenat's *AM* (1976) and ***Eurisko*** (1983) programs, or in a production system that employs a *build* operation (Waterman, 1975) — whereby productions can themselves create new productions — as in Anzai & Simon's (1979) work on learning by doing. Sometimes the learner is constructed as a separate *critic* with its own problem solver (Smith, Mitchell, Chestek, & Buchanan, 1977; Rendell, 1983), or as a set of critics as in Sussman's (1977) ***Hacker*** program.

- *A Simple Experience Learner.* There is a single learning mechanism that bases its modifications on the experience of the problem solver. The learning mechanism is fixed, and does not perform any complex problem solving. Examples of this approach are memo functions (Michie, 1968; Marsh, 1970), macro-operators in ***Strips*** (Fikes, Hart and Nilsson, 1972), production composition (Lewis, 1978; Neves & Anderson, 1981), and knowledge compilation (Anderson, 1983b).

The third approach, the simple experience learner, is the one adopted in ***Soar***. In some ways it is the most parsimonious of the three alternatives: it makes use of only one learning mechanism, in contrast to a multistrategy learner; it makes use of only one problem solver, in contrast to a critic-based deliberate learner; and it requires only problem solving about the actual task to be performed, in contrast to both kinds of deliberate learner. Counterbalancing the parsimony is that it is not obvious a priori that a simple experience learner can provide an adequate foundation for the construction of a general learning mechanism. At first glance, it would appear that such a mechanism would have difficulty learning from a variety of sources of knowledge, learning about all aspects of the system, and transferring what it has learned to new situations.

The hypothesis being tested in the research on ***Soar*** is that *chunking*, a simple experience-based learning mechanism, can form the basis for a general learning mechanism.[2] Chunking is a mechanism originally developed as part of a psychological model of memory (Miller, 1956). The concept of a chunk — a symbol that designates a pattern of other symbols — has been much studied as a model of memory organization. It has been used to explain such phenomena as why the span of short term memory is approximately constant, independent of the complexity of the items to be remembered (Miller, 1956), and why chess masters have an advantage over novices in reproducing chess positions from memory (Chase & Simon, 1973).

Newell and Rosenbloom (1981) proposed chunking as the basis for a model of human practice and used it to model the ubiquitous power law of practice — that the time to perform a task is a power-law function of the number of times the task has been performed. The model was based on the idea that practice improves performance via the acquisition of knowledge about patterns in the task environment, that is, chunks. When the model was implemented as part of a production-system architecture, this idea was instantiated with chunks relating patterns of goal parameters to patterns of goal results (Rosenbloom, 1983; Rosenbloom &

[2] For a comparison of chunking to other simple mechanisms for learning by experience, see Rosenbloom and Newell (1985).

Newell, 1985). By replacing complex processing in subgoals with chunks learned during practice, the model could improve its speed in performing a single task or set of tasks.

To increase the scope of the learning beyond simple practice, a similar chunking mechanism has been incorporated into the *Soar* problem-solving architecture (Laird, Newell, & Rosenbloom, 1985). In previous work we have demonstrated how chunking can improve *Soar's* performance on a variety of tasks and in a variety of ways (Laird, Rosenbloom & Newell, 1984). In this article we focus on presenting the details of how chunking works in *Soar* (Section 3), and describe a new application involving the acquisition of macro-operators similar to those reported by Korf (1985a) (Section 4). This demonstration extends the claims of generality, and highlights the ability of chunking to transfer learning between different situations.

Before proceeding to the heart of this work — the examination of the anatomy of chunking and a demonstration of its capabilities — it is necessary to make a fairly extensive digression into the structure and performance of the *Soar* architecture (Section 2). In contrast to systems with multistrategy or deliberate learning mechanisms, the learning phenomena exhibited by a system with only a simple experience-based learning mechanism is a function not only of the learning mechanism itself, but also of the problem-solving component of the system. The two components are closely coupled and mutually supportive.

2. Soar — An Architecture for General Intelligence

Soar is an architecture for general intelligence that has been applied to a variety of tasks (Laird, Newell, & Rosenbloom, 1985; Rosenbloom, Laird, McDermott, Newell, & Orciuch, 1985): many of the classic AI toy tasks such as the Tower of Hanoi, and the Blocks World; tasks that appear to involve non-search-based reasoning, such as syllogisms, the three-wise-men puzzle, and sequence extrapolation; and large tasks requiring expert-level knowledge, such as the *R1* computer configuration task (McDermott, 1982). In this section we briefly review the *Soar* architecture and present an example of its performance in the Eight Puzzle.

2.1. The Architecture

Performance in *Soar* is based on the *problem space-hypothesis*: all goal-oriented behavior occurs as search in problem spaces (Newell, 1980). A problem space for a task domain consists of a set of *states* representing possible situations in the task domain and a set of *operators* that transform one state into another one. For example, in the chess domain the states are configurations of pieces on the board, while the operators are the legal moves, such as P-K4. In the computer-configuration domain the states are partially configured computers, while the operators add components to the existing configuration (among other actions). Problem solving in a problem space consists of starting at some given *initial state*, and applying operators (yielding intermediate states) until a *desired state* is reached that is recognized as achieving the goal.

In *Soar*, each goal has three slots, one each for a current problem space, state, and operator. Together these four components — a goal along with its current problem space, state and operator — comprise a *context*. Goals can have subgoals (and associated contexts), which form a strict goal-subgoal hierarchy. All objects (such as goals, problem spaces, states, and operators) have a unique *identifier*, generated at the time the object was created. Further descriptions of an object are called *augmentations*. Each augmentation has an identifier, an attribute, and a value. The value can either be a constant value, or the identifier of another object. All objects are connected via augmentations (either directly, or indirectly via a chain of augmentations) to one of the objects in a context, so that the identifiers of objects act as nodes of a semantic network, while the augmentations represent the arcs or links.

Throughout the process of satisfying a goal, *Soar* makes *decisions* in order to select between the available problem spaces, states, and operators. Every problem-solving episode consists of a sequence of decisions and these decisions determine the behavior of the system. Problem solving in pursuit of a goal begins with the selection of a problem space for the goal. This is followed by the selection of an initial state, and then an operator to apply to the state. Once the operator is selected, it is applied to create a new state. The new state can then be selected for further processing (or the current state can be kept, or some previously generated state can be selected), and the process repeats as a new operator is selected to apply to the selected state. The weak methods can be represented as knowledge for controlling the selection of states and operators (Laird & Newell, 1983a). The knowledge that controls these decisions is collectively called *search control*. Problem solving without search control is possible in *Soar*, but it leads to an exhaustive search of the problem space.

Figure 1 shows a schematic representation of a series of decisions. To bring the available search-control knowledge to bear on the making of a decision, each decision involves a monotonic *elaboration phase*. During the elaboration phase, all *directly* available knowledge relevant to the current situation is brought to bear. This is the act of retrieving knowledge from memory to be used to control problem solving. In *Soar*, the long-term memory is structured as a production system, with all directly available knowledge represented as productions.[3] The elaboration phase consists of one or more cycles of production execution in which all of the eligible productions are fired in parallel. The contexts of the goal hierarchy and their augmentations serve as the working memory for these productions. The information added during the elaboration phase can take one of two forms. First, existing objects may have their descriptions elaborated (via augmentations) with new or existing objects, such as the addition of an evaluation to a state. Second, data structures called *preferences* can be created that specify the desirability of an object for a slot in a context. Each preference indicates the context in which it is relevant by specifying the appropriate goal, problem space, state and operator.

---

[3] We will use the terms production and rule interchangeably throughout this paper.

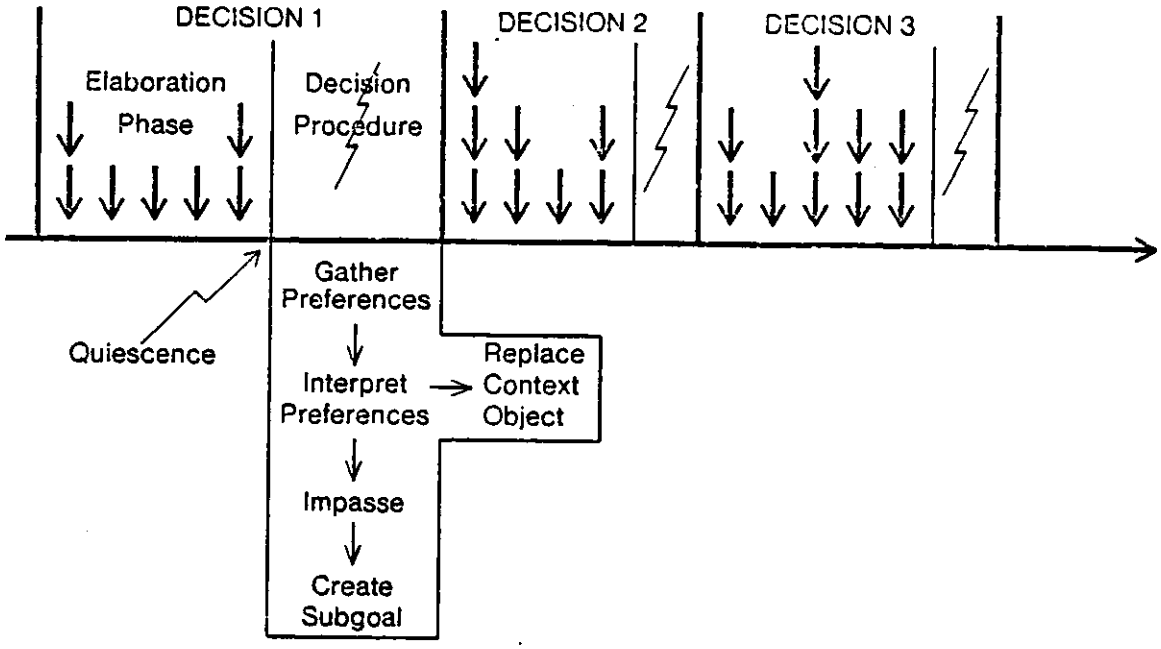

Figure 1: The *Soar* decision cycle.

When the elaboration phase reaches quiescence — when no more productions are eligible to fire — a fixed *decision procedure* is run that gathers and interprets the preferences provided by the elaboration phase to produce a specific decision. Preferences of type *acceptable* and *reject* determine whether or not an object is a candidate for a context. Preferences of type *better*, *equal*, and *worse* determine the relative worth of objects. Preferences of type *best*, *indifferent* and *worst* make absolute judgements about the worth of objects.[4] Starting from the oldest context, the decision procedure uses the preferences to determine if the current problem space, state, or operator in any of the contexts should be changed. The problem space is considered first, followed by the state and then the operator. A change is made if one of the candidate objects for the slot dominates (based on the preferences) all of the others, or if a set of equal objects dominates all of the other objects. In the latter case, a random selection is made between the equal objects. Once a change has been made, the subordinate positions in the context (state and operator if a problem space is changed) are initialized to **undecided**, all of the more recent contexts in the stack are discarded, the decision procedure terminates, and a new decision commences.

If sufficient knowledge is available during the search to uniquely determine a decision, the search proceeds unabated. However, in many cases the knowledge encoded into productions may be insufficient to allow the

[4] There is also a *parallel* preference that can be used to assert that two operators should execute simultaneously.

direct application of an operator or the making of a search-control decision. That is, the available preferences do not determine a unique, uncontested change in a context, causing an *impasse* in problem solving to occur (Brown & VanLehn, 1980). Four classes of impasses can arise in ***Soar:*** (1) *no-change* (the elaboration phase ran to quiescence without suggesting any changes to the contexts), (2) *tie* (no single object or group of equal objects was better than all of the other candidate objects), (3) *conflict* (two or more candidate objects were better than each other), and (4) *rejection* (all objects were rejected, even the current one). All types of impasse can occur for any of the three context slots associated with a goal — problem space, state, and operator — and a no-change impasse can occur for the goal. For example, a state tie occurs whenever there are two or more competing states and no directly available knowledge to compare them. An operator no-change occurs whenever no context changes are suggested after an operator is selected (usually because not enough information is directly available to allow the creation of a new state).

Soar responds to an impasse by creating a subgoal (and an associated context) to resolve the impasse. Once a subgoal is created, a problem space must be selected, followed by an initial state, and then an operator. If an impasse is reached in any of these decisions, another subgoal will be created to resolve it, leading to the hierarchy of goals in ***Soar***. By generating a subgoal for each impasse, the full problem-solving power of ***Soar*** can be brought to bear to resolve the impasse. These subgoals correspond to all of the types of subgoals created in standard AI systems (Laird, Newell, & Rosenbloom, 1985). This capability to generate automatically all subgoals in response to impasses and to open up all aspects of problem-solving behavior to problem solving when necessary is called *universal subgoaling* (Laird, 1984).

Because all goals are generated in response to impasses, and each goal can have at most one impasse at a time, the goals (contexts) in working memory are structured as a stack, referred to as the *context stack.* A subgoal terminates when its impasse is resolved. For example, if a tie impasse arises, the subgoal generated for it will terminate when sufficient preferences have been created so that a single object (or set of equal objects) dominates the others. When a subgoal terminates, ***Soar*** pops the context stack, removing from working memory all augmentations created in that subgoal that are not connected to a prior context, either directly or indirectly (by a chain of augmentations), and preferences whose context objects do not match objects in prior contexts. Those augmentations and preferences that are not removed are the *results* of the subgoal.

Default knowledge (in the form of productions) exists in ***Soar*** to cope with any of the subgoals when no additional knowledge is available. For some subgoals (those created for all types of rejection impasses and no-change impasses for goals, problem-spaces, and states) this involves simply backing up to a prior choice in the context, but for other subgoals (those create for tie, conflict and operator no-change impasses), this involves searches for knowledge that will resolve the subgoal's impasse. If additional non-default knowledge

is available to resolve an impasse, it dominates the default knowledge (via preferences) and controls the problem solving within the subgoal.

2.2. An Example Problem Solving Task

Consider the Eight Puzzle, in which there are eight numbered, movable tiles set in a 3x3 frame. One cell of the frame is always empty (the blank), making it possible to move an adjacent tile into the empty cell. The problem is to transform one configuration of tiles into a second configuration by moving the tiles. The states of the **eight-puzzle** problem space are configurations of the numbers 1-8 in a 3x3 grid. There is a single general operator to move adjacent tiles into the empty cell. For a given state, an instance of this operator is created for each of the cells adjacent to the empty cell. Each of these operator instances is instantiated with the empty cell and one of the adjacent cells. To simplify our discussion, we will refer to these instantiated operators by the direction they move a tile into the empty cell: **up**, **down**, **left**, or **right**. Figure 2 shows an example of the initial and desired states of an Eight Puzzle problem.

Figure 2: Example initial and desired states of the Eight Puzzle.

To encode this task in *Soar*, one must include productions that propose the appropriate problem space, create the initial state of that problem space, implement the operators of the problem space, and detect the desired state when it is achieved. If no additional knowledge is available, an exhaustive depth-first search occurs as a result of the default processing for tie impasses. Tie impasses arise each time an operator has to be selected. In response to the subgoals for these impasses, alternatives are investigated to determine the best move. Whenever another tie impasse arises during the investigation of one of the alternatives, an additional subgoal is generated, and the search deepens. If additional search-control knowledge is added to provide an evaluation of the states, the search changes to steepest-ascent hill climbing. As more or different search-control knowledge is added, the behavior of the search changes in response to the new knowledge. One of the properties of *Soar* is that the weak methods, such as generate and test, means-ends analysis, depth-first search and hill climbing, do not have to be explicitly selected, but instead emerge from the structure of the task and

the available search-control knowledge (Laird & Newell, 1983a; Laird & Newell, 1983b; Laird, 1984).

Another way to control the search in the Eight puzzle is to break it up into a set of subgoals to get the individual tiles into position. We will look at this approach in some detail because it forms the basis for the use of macro-operators for the Eight Puzzle. Means-ends analysis is the standard technique for solving problems where the goal can be decomposed into a set of subgoals, but it is ineffective for problems such as the Eight Puzzle that have *non-serializable* subgoals — tasks for which there exist no ordering of the subgoals such that successive subgoals can be achieved without undoing what was accomplished by earlier subgoals (Korf, 1985a). Figure 3 shows an intermediate state in problem solving where tiles 1 and 2 are in their desired positions. In order to move tile 3 into its desired position, tile 2 must be moved out of its desired position. Non-serializable subgoals can be tractable if they are *serially decomposable* (Korf, 1985a). A set of subgoals is serially decomposable if there is an ordering of them such that the solution to each subgoal depends only on that subgoal and on the preceding ones in the solution order. In the Eight Puzzle the subgoals are, in order: (1) have the blank in its correct position; (2) have the blank and the first tile in their correct positions; (3) have the blank and the first two tiles in their correct positions; and so on through the eighth tile. Each subgoal depends only on the positions of the blank and the previously placed tiles. Within one subgoal a previous subgoal may be undone, but if it is, it must be re-achieved before the current subgoal is completed.

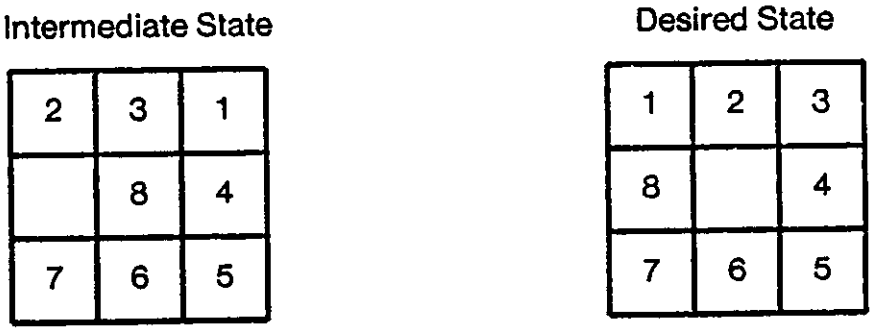

Figure 3: Non-serializable subgoals in the Eight Puzzle

Adopting this approach does not result in new knowledge for directly controlling the selection of operators and states in the eight-puzzle problem space. Instead it provides knowledge about how to structure and decompose the puzzle. This knowledge consists of the set of serially decomposable subgoals, and the ordering of those subgoals. To encode this knowledge in *Soar*, we have added a second problem space, eight-puzzle-sd,

with a set of nine operators corresponding to the nine subgoals.[5] For example, the operator place-2 will place tile 2 in its desired position, while assuring that the blank and the first tile will also be in position. The ordering of the subgoals is encoded as search-control knowledge that creates preferences for the operators.

Figure 4 shows a trace of the decisions for a short problem-solving episode for the initial and desired states from Figure 2. This example is heavily used in the remainder of the paper, so we shall go through it in some detail. To start problem solving, the current goal is initialized to be **solve-eight-puzzle** (in decision 1). The goal is represented in working memory by an identifier, in this case G1. Problem solving begins in the **eight-puzzle-sd** problem space. Once the initial state, S1, is selected, preferences are generated that order the operators so that **place-blank** is selected. Application of this operator, and all of the **eight-puzzle-sd** operators, is complex, often requiring extensive problem solving. Because the problem-space hypothesis implies that such problem solving should occur in a problem space, the operator is not directly implemented as rules. Instead, a no-change impasse leads to a subgoal to implement **place-blank**, which will be achieved when the blank is in its desired position. The **place-blank** operator is then implemented as a search in the **eight-puzzle** problem space for a state with the blank in the correct position. This search can be carried out using any of the weak methods described earlier, but for this example, let us assume there is no additional search-control knowledge.

Once the initial state is selected (decision 7), a tie impasse occurs among the operators that move the three adjacent tiles into the empty cell (**left**, **up** and **down**). A resolve-tie subgoal (G3) is automatically generated for this impasse, and the **tie** problem space is selected. Its states are sets of objects being considered, and its operators evaluate objects so that preferences can be created. One of these **evaluate-object** operators (O5) is selected to evaluate the operator that moves tile 8 to the left, and a resolve-no-change subgoal (G4) is generated because there are no productions that directly compute an evaluation of the **left** operator for state S1. Default search-control knowledge attempts to implement the **evaluate-object** operator by applying the **left** operator to state S1. This is accomplished in the subgoal (decisions 13-16), yielding the desired state (S3). Because the **left** operator lead to a solution for the goal, a preference is returned for it that allows it to be selected immediately for state S1 (Decision 17) in goal G2, flushing the two lower subgoals (G3 and G4). If this state were not the desired state, another tie impasse would arise and the tie problem space would be selected for this new subgoal. The subgoal combination of a resolve-tie followed by a resolve-no-change on an **evaluate-object** operator would recur, giving a depth-first search.

Applying the **left** operator to state S1 yields state S4, which is the desired result of the **place-blank** operator

[5] Both **place-7** and **place-8** are always no-ops because once the blank and tiles 1-6 are in place, either tiles 7 and 8 must also be in place, or the problem is unsolvable. They can therefore be safely ignored.

```
1  G1 solve-eight-puzzle
2  P1 eight-puzzle-sd
3  S1
```

2	3	1
	8	4
7	6	5

```
4  O1 place-blank
5  ==>G2 (resolve-no-change)
6     P2 eight-puzzle
7     S1
8     ==>G3 (resolve-tie operator)
9        P3 tie
10       S2 {left, up, down}
11       O5 evaluate-object(O2(left))
12       ==>G4 (resolve-no-change)
13          P2 eight-puzzle
14          S1
15          O2 left
16          S3
```

2	3	1
8		4
7	6	5

```
17     O2 left
18     S4
19 S4
20 O8 place-1
```

Figure 4: A problem-solving trace for the Eight Puzzle. Each line of the trace includes, from left to right, the decision number, the identifier of the object selected, and possibly a short description of the object.

in goal G1 above. The **place-1** operator (O8) is then selected as the current operator. As with **place-blank**, **place-1** is implemented by a search in the **eight-puzzle** problem space. It succeeds when both tile 1 and the blank are in their desired positions. With this problem-solving strategy, each tile is moved into place by one of the operators in the **eight-puzzle-sd** problem space. In the subgoals that implement the **eight-puzzle-sd** operators, many of the tiles already in place might be moved out of place, however, they must be back in place for the operator to terminate successfully.

3. Chunking in Soar

Soar was originally designed to be a general (non-learning) problem solver. Nevertheless, its problem-solving and memory structures support learning in a number of ways. The structure of problem solving in *Soar* determines when new knowledge is needed, what that knowledge might be, and when it can be acquired.

- *Determining when new knowledge is needed.* In *Soar*, impasses occur if and only if the directly available knowledge is either incomplete or inconsistent. Therefore, impasses indicate when the system should attempt to acquire new knowledge.

- *Determining what to learn.* While problem solving within a subgoal, *Soar* can discover information that will resolve an impasse. This information, if remembered, can avert similar impasses in future problem solving.

- *Determining when new knowledge can be acquired.* When a subgoal completes, because its impasse has been resolved, an opportunity exists to add new knowledge that was not already explicitly known.

Soar's long-term memory, which is based on a production system and the workings of the elaboration phase, supports learning in two ways:

- *Integrating new knowledge.* Productions provide a modular representation of knowledge, so that the integration of new knowledge only requires adding a new production to production memory and does not require a complex analysis of the previously stored knowledge in the system (Newell, 1973; Waterman, 1975; Davis & King, 1976; Anderson, 1983b).

- *Using new knowledge.* Even if the productions are syntactically modular, there is no guarantee that the information they encode can be integrated together when it is needed. The elaboration phase of *Soar* brings all appropriate knowledge to bear, with no requirement of synchronization (and no conflict resolution). The decision procedure then integrates the results of the elaboration phase.

Chunking in *Soar* takes advantage of this support to create rules that summarize the processing of a subgoal, so that in the future, the costly problem solving in the subgoal can be replaced by direct rule application. When a subgoal is generated, a learning episode begins that could lead to the creation of a chunk. During problem solving within the subgoal, information accumulates on which a chunk can be based. When the subgoal terminates, a chunk can be created. Each chunk is a rule (or set of rules) that gets added to the production memory. Chunked knowledge is brought to bear during the elaboration phase of later decisions. In the remainder of this section we look in more detail at the process of chunk creation, evaluate the scope of chunking as a learning mechanism, and examine the sources of chunk generality.

3.1. Constructing Chunks

Chunks are based on the working memory elements that are either examined or created during problem solving within a subgoal. The conditions consist of those aspects of the situation that existed prior to the goal, and which were examined during the processing of the goal, while the actions consist of the results of the goal.

When the subgoal terminates,[6] the collected working-memory elements are converted into the conditions and actions of one or more productions.[7] In this subsection, we describe in detail the three steps in chunk creation: (1) the collection of conditions and actions, (2) the variabilization of identifiers, and (3) chunk optimization.

3.1.1. Collecting Conditions and Actions

The conditions of a chunk should test those aspects of the situation existing prior to the creation of the goal that are relevant to the results that satisfy the goal. In *Soar* this corresponds to the working-memory elements that were matched by productions that fired in the goal (or one of its subgoals), but that existed before the goal was created. These are the elements that the problem solving implicitly deemed to be relevant to the satisfaction of the subgoal. This collection of working-memory elements is maintained for each active goal in the goal's *referenced-list*.[8] *Soar* allows productions belonging to any goal in the context stack to execute at any time, so updating the correct referenced-list requires determining for which goal in the stack the production fired. This is the most recent of the goals matched by the production's conditions. The production's firing affects the chunks created for that goal and all of its supergoals, but because the firing is independent of the more recent subgoals, it has no effect on the chunks built for those subgoals. No chunk is created if the subgoal's results were not based on prior information; for example, when an object is input from the outside, or when an impasse is resolved by domain-independent default knowledge.

The actions of a chunk are based on the results of the subgoal for which the chunk was created. No chunk is created if there are no results. This can happen, for example, when a result produced in a subgoal leads to the termination of a goal much higher in the goal hierarchy. All of the subgoals that are lower in the hierarchy will also be terminated, but they may not generate results.

For an example of chunking in action, consider the terminal subgoal (G4) from the problem-solving episode in Figure 4. This subgoal was created as a result of a no-change impasse for the evaluate-object

---

[6] The default behavior for *Soar* is to create a chunk *always;* that is, every time a subgoal terminates. The major alternative to creating chunks for all terminating goals is to chunk *bottom-up*, as was done in modeling the power law of practice (Rosenbloom, 1983). In bottom-up chunking, only terminal goals — goals for which no subgoals were generated — are chunked. As chunks are learned for subgoals, the subgoals need no longer be generated (the chunks accomplish the subgoals' tasks before the impasses occur), and higher goals in the hierarchy become eligible for chunking. It is unclear whether chunking always or bottom-up will prove more advantageous in the long run, so to facilitate experimentation, both options are available in *Soar*.

[7] *Production composition* (Lewis, 1978) has also been used to learn productions that summarize goals (Anderson, 1983b). It differs most from chunking in that it examines the actual definitions of the productions that fired in addition to the working-memory elements referenced and created by the productions.

[8] If a fired production has a negated condition — a condition testing for the absence in working memory of an element matching its pattern — then the negated condition is instantiated with the appropriate variable bindings from the production's positive conditions. If the identifier of the instantiated condition existed prior to the goal, then the instantiated condition is included in the referenced-list.

operator that should evaluate the operator that will move tile 8 to the left. The problem solving within goal G4 must implement the **evaluate-object** operator. Figure 5 contains a graphic representation of part of the working memory for this subgoal near the beginning of problem solving (A) and just before the subgoal is terminated (B). The working memory that existed before the subgoal was created consisted of the augmentations of the goal to resolve the tie between the **eight-puzzle** operators, G3, and its supergoals (G2 and G1, not shown). The tie problem space is the current problem space of G3, while state S2 is the current state and the **evaluate-object** operator (O5) is the current operator. D1 is the desired state of having the blank in the middle, but with no constraint on the tiles in the other cells (signified by the X's in the figure). All of these objects have further descriptions, some only partially shown in the figure.

The purpose of goal G4 is to evaluate operator O2, that will move tile 8 to the left in the initial state (S1). The first steps are to augment the goal with the desired state (D1) and then select the **eight-puzzle** problem space (P2), the state to which the operator will be applied (S1), and finally the operator being evaluated (O2). To do this, the augmentations from the **evaluate-object** operator (O5) to these objects are accessed and therefore added to the referenced list (the highlighted arrows in part (A) of Figure 5). Once operator O2 is selected, it is applied by a production that creates a new state (S3). The application of the operator depends on the exact representation used for the states of the problem space. State S1 and desired state D1, which were shown only schematically in Figure 5, are shown in detail in Figure 6. The states are built out of *cells* and *tiles* (only some of the cells and tiles are shown in Figure 6). The nine cells (C1-C9) represent the structure of the Eight Puzzle frame. They form a 3x3 grid in which each cell points to its adjacent cells. There are eight numbered tiles (T2-T9), and one blank (T1). Each tile points to its name, 1 through 8 for the numbered tiles and 0 for the blank. Tiles are associated with cells by objects called *bindings*. Each state contains 9 bindings, each of which associates one tile with the cell where it is located. The bindings for the desired state, D1, are L1-L9, while the bindings for state S1 are B1-B9. The fact that the blank is in the center of the desired state is represented by binding L2, which points to the blank tile (T1) and the center cell (C5). All states (and desired states) in both the **eight-puzzle** and **eight-puzzle-sd** problem spaces share this same cell structure.

To apply the operator and create a new state, a new state symbol is created (S3) with two new bindings, one for the moved tile and one for the blank. The binding for the moved tile points to the tile (T9) and to the cell where it will be (C4). The binding for the blank points to the blank (T1) and to the cell that will be empty (C5). All the other bindings are then copied to the new state. This processing accessing the relative positions of the blank and the moved tile, and the bindings for the remaining tiles in current state (S1). The augmentations of the operator are tested for the cell that contains the tile to be moved.

Once the new state (S3) is selected, a production generates the operators that can apply to the new state. All

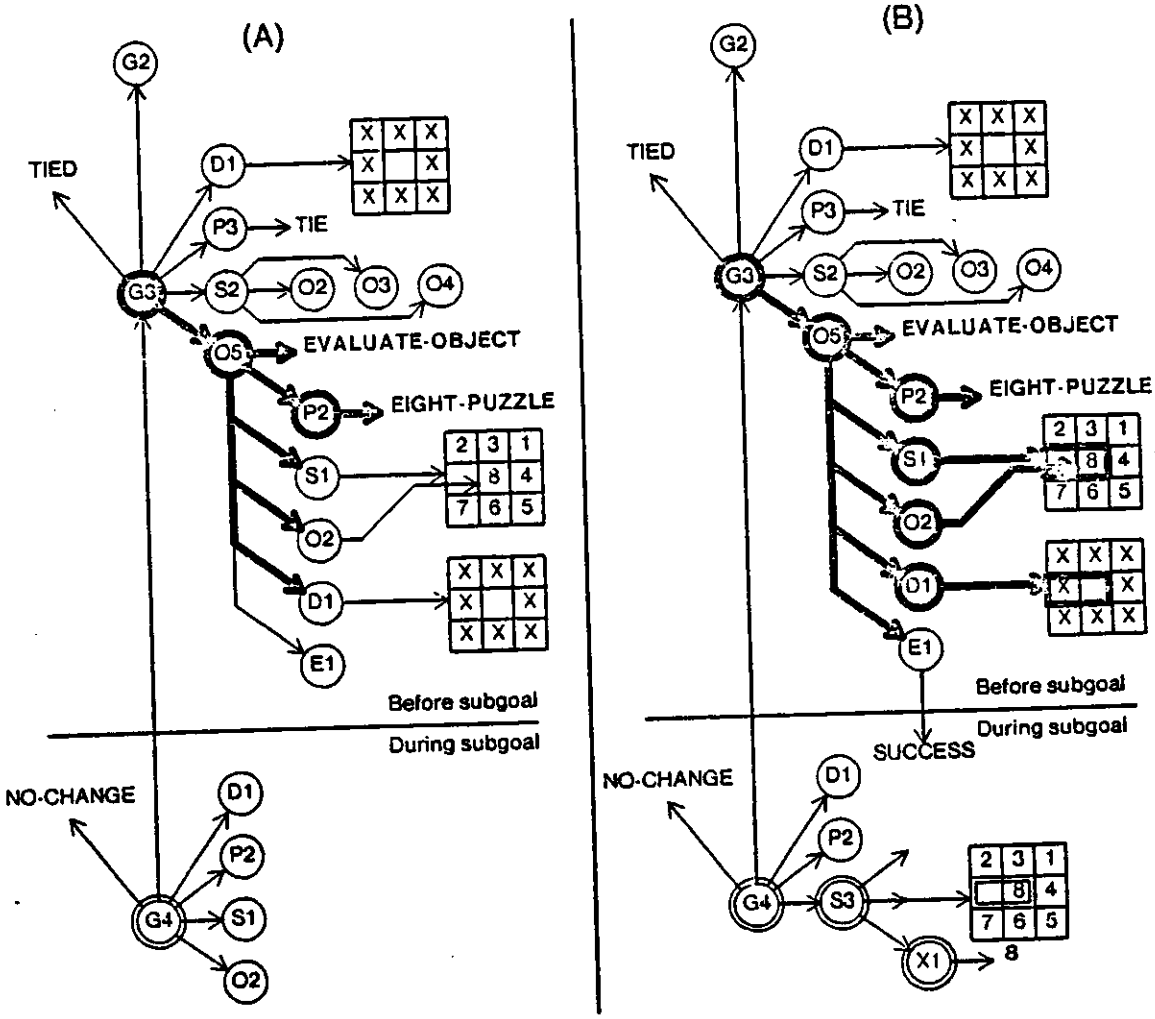

Figure 5: An example of the working-memory elements used to create a chunk. (A) shows working memory near the beginning of the subgoal to implement the evaluate-object operator. (B) shows working memory at the end of the subgoal. The circled symbols represent identifiers and the arrows represent augmentations. The identifiers and augmentations above the horizontal lines existed before the subgoal was created. Below the lines, the identifiers marked by doubled circles, and all of the augmentations, are created in the subgoal. The other identifiers below the line are not new; they are actually the same as the corresponding ones above the lines. The highlighted augmentations were referenced during the problem solving in the subgoal and will be the basis of the conditions of the chunk. The augmentation that was created in the subgoal but originates from an object existing before the subgoal (E1->SUCCESS) will be the basis for the action of the chunk.

cells that are adjacent to the blank cell (C2, C4, C6, and C8) are used to create operators. This requires testing the structure of the board as encoded in the connections between the cells. Following the creation of the operators that can apply to state S3, the operator that would undo the previous operator is rejected so that unnecessary backtracking is avoided. During the same elaboration phase, the state is tested to determine

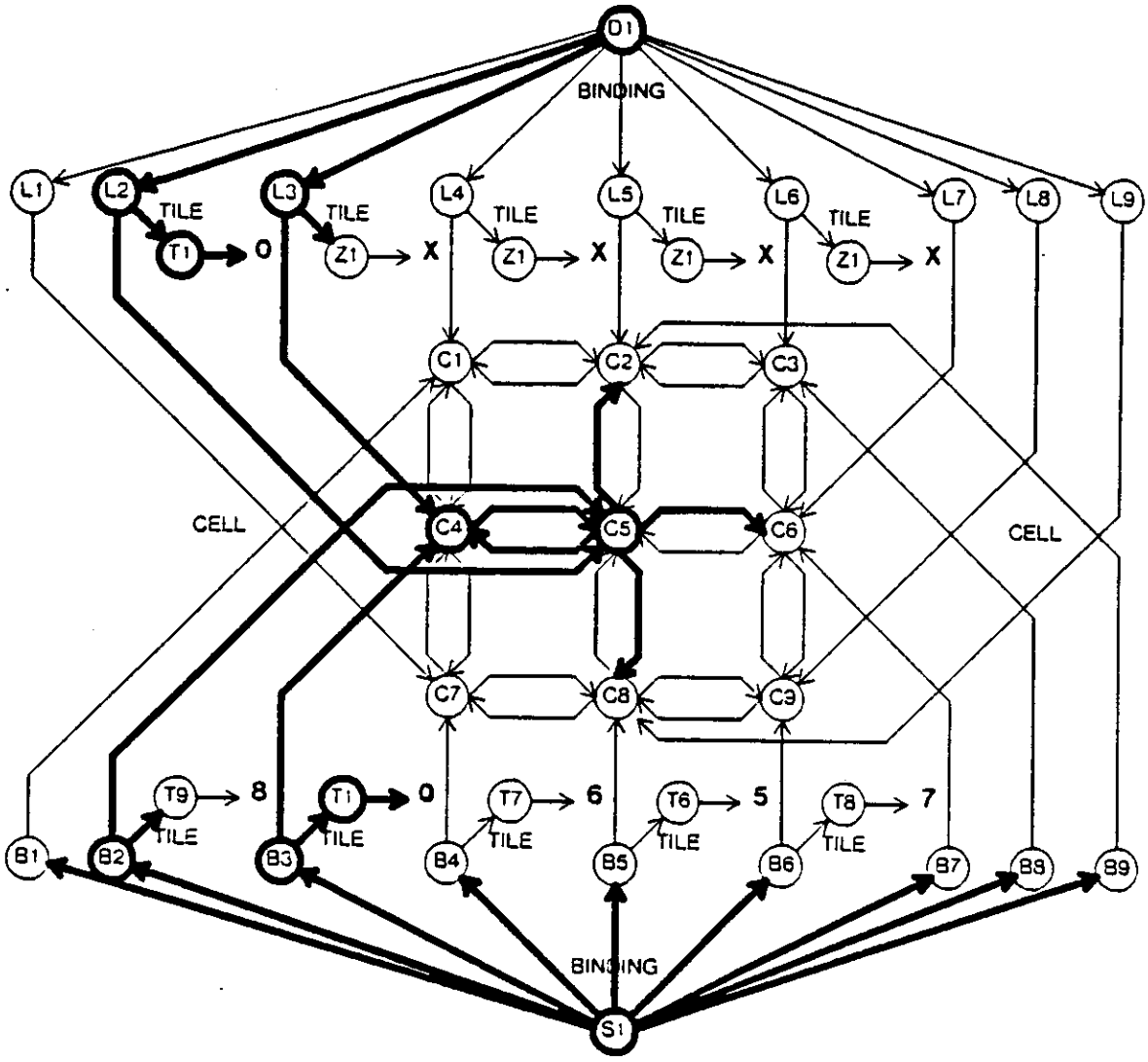

Figure 6: Example of working-memory elements representing the state used to create a chunk. The highlighted augmentations were referenced during the the subgoal.

whether a tile was just moved in or out of its correct position. This information is used to generate an evaluation based on the sum of the number of tiles that do not have to be in place and the number of tiles that both have to be in place and are in place. This computation, whose result is represented by the object X1 with a value of 8 in Figure 5, results in the accessing of those aspects of the desired state highlighted in Figure 6. The value of 8 means that the goal is satisfied, so the evaluation (E1) for the operator has the value success. Because E1 is an identifier that existed before the subgoal was created and the success augmentation was created in the subgoal, this augmentation becomes an action. If success had further augmentations, they would also be included as actions. The augmentations of the subgoal (G4), the new state (S3), and its sub-object (X1) that point to objects created before the subgoal are not included as actions because they are not augmentations, either directly or indirectly, of an object that existed prior to the creation of the subgoal.

When goal G4 terminates, the initial set of conditions and actions have been determined for the chunk. The conditions test that there exists an evaluate-object operator whose purpose is to evaluate the operator that moves the blank into its desired location, and that all of the tiles are either in position or irrelevant for the current eight-puzzle-sd operator. The action is to mark the evaluation as successful, meaning that the operator being evaluated will achieve the goal. This chunk should apply in similar future situations, directly implementing the evaluate-object operator, and avoiding the no-change impasse and the resulting subgoal.

3.1.2. Identifier Variabilization

Once the conditions and actions have been determined, all of the identifiers are replaced by production (pattern-match) variables, while the constants, such as evaluate-object, eight-puzzle, and 0 are left unchanged. An identifier is a label by which a particular instance of an object in working memory can be referenced. It is a short-term symbol that lasts only as long as the object is in working memory. Each time the object reappears in working memory it is instantiated with a new identifier. If a chunk that is based on working-memory elements is to reapply in a later situation, it must not mention specific identifiers. In essence the variabilization process is like replacing an "eq" test in *Lisp* (which requires pointer identity) with an "equal" test (which only requires value identity).

All occurrences of a single identifier are replaced with the same variable and all occurrences of different identifiers are replaced by different variables. This assures that the chunk will match in a new situation only if there is an identifier that appears in the same places in which the original identifier appeared. The production is also modified so that no two variables can match the same identifier. Basically, *Soar* is guessing which identifiers must be equal and which must be distinct, based only on the information about equality and inequality in working memory. All identifiers that are the same are assumed to require equality. All identifiers that are not the same are assumed to require inequality. Biasing the generalization in these ways assures that the chunks will not be overly general (at least because of these modifications), but they may be overly specific. The only problem this causes is that additional chunks may need to be learned if the original ones suffer from overspecialization. In practice, these modifications have not led to overly specific chunks.

3.1.3. Chunk Optimization

At this point in the chunk-creation process the semantics of the chunk are determined. However, three additional processes are applied to the chunks to increase the efficiency with which they are matched against working memory (all related to the use in *Soar* of the *Ops5* rule matcher (Forgy, 1981)). The first process is to remove conditions from the chunk that provide (almost) no constraint on the match process. A condition is removed if it has a variable in the value field of the augmentation that is not bound elsewhere in the rule (either in the conditions or the actions). This process recurses, so that a long linked-list of conditions will be removed if the final one in the list has a variable that is unique to that condition. For the chunk based on

Figures 5 and 6, the bindings and tiles that were only referenced for copying (B1, B4, B5, B6, B7, B8, B9, and T9) and the cells referenced for creating operator instantiations (C2, C6, and C8) are all removed. The evaluation object, E1, in Figure 5 is not removed because it is included in the action. Eliminating the bindings does not increase the generality of the chunk, because all states must have nine bindings. However, the removal of the cells does increase the generality, because they (along with the test of cell C4) implicitly test that there must be four cells adjacent to the one to which the blank will be moved. Only the center has four adjacent cells, so the removal of these conditions does increase the generality. This does increase slightly the chance of the chunk being over-general, but in practice it has never caused a problem, and it can significantly increase the efficiency of the match by removing unconstrained conditions.

The second optimization is to eliminate potential combinatorial matches in the conditions of productions whose actions are to copy a set of augmentations from an existing object to a new object. A common strategy for implementing operators in subgoals is to create a new state containing only the new and changed information, and then to copy over pointers to the rest of the previous state. The chunks built for these subgoals contain one condition for each of the copied pointers. If, as is usually the case, a set of similar items are being copied, then the copy conditions end up differing only in the names of variables. Each augmentation can match each of these conditions independently, generating a combinatorial number of instantiations. This problem would arise if a subgoal were used to implement the eight-puzzle operators instead of the rules used in our current implementation. A single production would be learned that created new bindings for the moved tile and the blank, and also copied all of the other bindings. There would be seven conditions that tested for the bindings, but each of these conditions could match any of the bindings that had to be copied, generating 7! (5040) instantiations. This problem is solved by collapsing the set of similar copy conditions down to one. All of the augmentations can still be copied over, but it now occurs via multiple instantiations (seven of them) of the simpler rule. Though this reduces the number of rule instantiations to linear in the number of augmentations to be copied, it still means that the other non-copying actions are done more than once. This problem is solved by splitting the chunk into two productions. One production does everything the subgoal did except for the copying. The other production just does the copying. If there is more than one set of augmentations to be copied, each set is collapsed into a single condition and a separate rule is created for each.[9]

The final optimization process consists of applying a condition-reordering algorithm to the chunk productions. The efficiency of the Rete-network matcher (Forgy, 1982) used in *Soar* is sensitive to the order in which conditions are specified. By taking advantage of the known structure of *Soar's* working memory, we

[9]The inelegance of this solution leads us to believe that we do not yet have the right assumptions about how new objects are to be created from old ones.

have developed a static reordering algorithm that significantly increases the efficiency of the match. Execution time is sometimes improved by more than an order of magnitude, almost duplicating the efficiency that would be achieved if the reordering was done by hand. This reordering process preserves the existing semantics of the chunk.

3.2. The Scope of Chunking

In Section 1 we defined the scope of a general learning mechanism in terms of three properties: task generality, knowledge generality, and aspect generality. Below we briefly discuss each of these with respect to chunking in *Soar*.

Task generality. *Soar* provides a single formalism for all behavior — heuristic search of problem spaces in pursuit of goals. This formalism has been widely used in Artificial Intelligence (Feigenbaum and Feldman, 1963; Nilsson, 1980; Rich, 1983) and it has already worked well in *Soar* across a wide variety of problem domains (Laird, Newell, & Rosenbloom, 1985). If the problem-space hypothesis (Newell, 1980) does hold, then this should cover all problem domains for which goal-oriented behavior is appropriate. Chunking can be applied to all of the domains for which *Soar* is used. Though it remains to be shown that useful chunks can be learned for this wide range of domains, our preliminary experience suggests that the combination of *Soar* and chunking has the requisite generality.[10]

Knowledge generality. Chunking learns from the experiences of the problem solver. At first glance, it would appear to be unable to make use of instructions, examples, analogous problems, or other similar sources of knowledge. However, by using such information to help make decisions in subgoals, *Soar* can learn chunks that incorporate the new knowledge. This approach has worked for a simple form of user direction, and is under investigation for learning by analogy. The results are preliminary, but it establishes that the question of knowledge generality is open for *Soar*.

Aspect generality. Three conditions must be met for chunking to be able to learn about all aspects of *Soar's* problem solving. The first condition is that all aspects must be open to problem solving. This condition is met because *Soar* creates subgoals for all of the impasses it encounters during the problem solving process. These subgoals allow for problem solving on any of the problem solver's functions: creating a problem space, selecting a problem space, creating an initial state, selecting a state, selecting an operator, and applying an operator. These functions are both necessary and sufficient for *Soar* to solve problems. So far chunking has been demonstrated for the selection and application of operators (Laird, Rosenbloom & Newell, 1984); that

---

[10] For demonstrations of chunking in *Soar* on the Eight Puzzle, Tic-Tac-Toe, and the *R1* computer-configuration task, see Laird, Rosenbloom, & Newell (1984), Rosenbloom, Laird, McDermott, Newell, & Orciuch (1985), and van de Brug, Rosenbloom, & Newell (1985).

is, strategy acquisition (Langley, 1983; Mitchell, 1983) and operator implementation. However, demonstrations of chunking for the other types of subgoals remain to be done.[11]

The second condition is that the chunking mechanism must be able to create the long-term memory structures in which the new knowledge is to be represented. *Soar* represents all of its long-term knowledge as productions, and chunking acquires new productions. By restricting the kinds of condition and action primitives allowed in productions (while not losing Turing equivalence), it is possible to have a production language that is coextensive syntactically with the types of rules learned by chunking; that is, the chunking mechanism can create rules containing all of the syntactic constructs available in the language.

The third condition is that the chunking mechanism must be able to acquire rules with the requisite content. In *Soar*, this means that the problem solving on which the requisite chunks are to be based must be understood. The current biggest limitations on coverage stem from our lack of understanding of the problem solving underlying such aspects as problem-space creation and change of representation (Hayes and Simon, 1976; Korf, 1980; Lenat, 1983; Utgoff, 1984).

3.3. Chunk Generality

One of the critical questions to be asked about a simple mechanism for learning from experience is the degree to which the information learned in one problem can transfer to other problems. If generality is lacking, and little transfer occurs, the learning mechanism is simply a caching scheme. The variabilization process described in Section 3.1.2 is one way in which chunks are made general. However, this process would by itself not lead to chunks that could exhibit non-trivial forms of transfer. All it does is allow the chunk to match another instance of the same exact situation. The principal source of generality is the *implicit generalization* that results from basing chunks on only the aspects of the situation that were referenced during problem solving. In the example in Section 3.1.1, only a small percentage of the augmentations in working memory ended up as conditions of the chunk. The rest of the information, such as the identity of the tile being moved and its absolute location, and the identities and locations of the other tiles, was not examined during problem solving, and therefore had no effect on the chunk.

Together, the representation of objects in working memory and the knowledge used during problem solving, combine to form the *bias* for the implicit generalization process (Utgoff, 1984); that is, they determine which generalizations are embodied in the chunks learned. The object representation defines a language for the implicit generalization process, bounding the potential generality of the chunks that can be learned. The

---

[11] In part this issue is one of rarity. For example, selection of problem spaces is not yet problematical, and conflict impasses have not yet been encountered.

problem solving determines (indirectly, by what it examines) which generalizations are actually embodied in the chunks.

Consider the state representation used in Korf's (1985a) work on the Eight Puzzle (recall Section 2.2). In his implementation, the state of the board was represented as a vector containing the positions of each of the tiles. Location 0 contained the coordinates of the position that was blank, location 1 contained the coordinates of the first tile, and so on. This is a simple and concise representation, but because aspects of the representation are overloaded with more than one functional concept, it provides poor support for implicit generalization (or for that matter, any traditional condition-finding method). For example, the vector indices have two functions: they specify the identity of the tile, and they provide access to the tile's position. When using this state representation it is impossible to access the position of a tile without looking at its identity. Therefore, even when the problem solving is only dependent on the locations of the tiles, the chunks learned would test the tile identities, thus failing to apply in situations in which they rightly could. A second problem with the representation is that some of the structure of the problem is implicit in the representation. Concepts that are required for good generalizations, such as the relative positions of two tiles, cannot be captured in chunks because they are not explicitly represented in the structure of the state. Potential generality is maximized if an object is represented so that functionally independent aspects are explicitly represented and can be accessed independently. For example, the Eight Puzzle state representation shown in Figure 6 breaks each functional role into separate working-memory objects. This representation, while not predetermining what generalizations are to be made, defines a class of possible generalizations that include good ones for the Eight Puzzle.

The actual generality of the chunk is maximized (within the constraints established by the representation) if the problem solver only examines those features of the situation that are absolutely necessary to the solution of the problem. When the problem solver knows what it is doing, everything works fine, but generality can be lost when information that turns out to be irrelevant is accessed. For example, whenever a new state is selected, productions fire to suggest operators to apply to the state. This preparation goes on in parallel with the testing of the state to see if it matches the goal. If the state does satisfy the goal, then the preparation process was unnecessary. However, if the preparation process referenced aspects of the prior situation that were not accessed by previous productions, then irrelevant conditions will be added to the chunk. Another example occurs when false paths — searches that lead off of the solution path — are investigated in a subgoal. The searches down unsuccessful paths may reference aspects of the state that would not have been tested if only the successful path were followed.[12]

---

[12] An experimental version of chunking has been implemented that overcomes these problems by performing a dependency analysis on traces of the productions that fired in a subgoal. The production traces are used to determine which conditions were necessary to produce results of the subgoal. All of the results of this paper are based on the version of chunking without the dependency analysis.

4. A Demonstration — Acquisition of Macro-Operators

In this section we provide a demonstration of the capabilities of chunking in *Soar* involving the acquisition of macro-operators in the Eight Puzzle for serially decomposable goals (see Section 2). We begin with a brief review of Korf's (1985a) original implementation of this technique. We follow this with the details of its implementation in *Soar*, together with an analysis of the generality of the macro-operators learned. This demonstration of macro-operators in *Soar* is of particular interest because: we are using a general problem solver and learner instead of special-purpose programs developed specifically for learning and using macro-operators; and because it allows us to investigate the generality of the chunks learned in a specific application.

4.1. Macro Problem Solving

Korf (1985a) has shown that problems that are serially decomposable can be efficiently solved with the aid of a table of *macro-operators*. A macro-operator (or *macro* for short) is a sequence of operators that can be treated as a single operator (Fikes, Hart and Nilsson, 1972). The key to the utility of macros for serially decomposable problems is to define each macro so that after it is applied, all subgoals that had been previously achieved are still satisfied, and one new subgoal is achieved. Means-ends analysis is thus possible when these macro-operators are used. Table 1 shows Korf's (1985a) macro table for the Eight Puzzle task of getting all of the tiles in order, clockwise around the frame, with the 1 in the upper left hand corner, and the blank in the middle (the desired state in Figure 3). Each column contains the macros required to achieve one of the subgoals of placing a tile. The rows give the appropriate macro according to the current position of the tile, where the positions are labeled A-I as in Figure 7. For example, if the goal is to move the blank (tile 0) into the center, and it is currently in the top left corner (location B), then the operator sequence *ul* will accomplish it.

		Tiles						
		0	1	2	3	4	5	6
Positions	A							
	B	ul						
	C	u	rdlu					
	D	ur	dlurrdlu	dlur				
	E	r	ldrurdlu	ldru	rdllurdrul			
	F	dr	uldrurdldrul	lurdldru	ldrulurddlru	lurd		
	G	d	urdldrul	ulddru	urddluldrrul	uldr	rdlluurdldrrul	
	H	dl	rulddrul	druuldrdlu	ruldrdluldrrul	urdluldr	uldrurdllurd	urdl
	I	l	drul	rulldru	rdluldrrul	rulldr	uldrruldlurd	ruld

Table 1: Macro table for the Eight Puzzle (from Korf, 1985, Table 1). The primitive operators move a tile one step in a particular direction: *u* (up), *d* (down), *l* (left), and *r* (right).

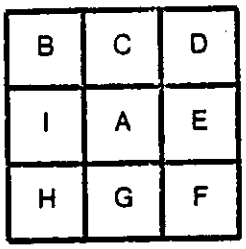

Figure 7: The positions (A-I) in the Eight Puzzle frame.

Korf's implementation of macro problem solving used two programs: a problem solver and a learner. The problem solver could use macro tables acquired by the learner to solve serially decomposable problems efficiently. Using Table 1, the problem-solving program could solve any Eight Puzzle problem with the same desired state (the initial state may vary). The procedure went as follows: (a) the position of the blank was determined; (b) the appropriate macro was found by using this position to index into the first column of the table; (c) the operators in this macro were applied to the state, moving the blank into position; (d) the position of the first tile was determined; (e) the appropriate macro was found by using this position to index into the second column of the table; (f) the operators in this macro were applied to the state, moving the first tile (and the blank) into position; and so on until all of the tiles were in place.

To discover the macros, the learner started with the desired state, and performed an iterative-deepening search (for example, see Korf, 1985b) using the elementary tile-movement operators.[13] As the search progressed, the learner detected sequences of operators that left some of the tiles invariant, but moved others. When an operator sequence was found that left an initial sequence of the subgoals invariant — that is, for some tile k, the operator moved that tile while leaving tiles 1 through k-1 where they were — the operator sequence was added to the macro table in the appropriate column and row. In a single search from the desired state, all macros could be found. Since the search used iterative-deepening, the first macro found was guaranteed to be the shortest for its slot in the table.

4.2. Macro Problem Solving in Soar

Soar's original design criteria did not include the ability to employ serially decomposable subgoals or to acquire and use macro-operators to solve problems structured by such subgoals. However, ***Soar's*** generality allows it to do so with no changes to the architecture (including the chunking mechanism). Using the implementation of the Eight Puzzle described in Sections 2.2 and 3.1.1, ***Soar's*** problem solving and learning capabilities work in an integrated fashion to learn and use macros for serially decomposable subgoals.

---

[13] For very deep searches, other more efficient techniques such as bidirectional search and macro-operator composition were used.

The opportunity to learn a macro-operator exists each time a goal for implementing one of the eight-puzzle-sd operators, such as **place-5**, is achieved. When the goal is achieved there is a stack of subgoals below it, one for each of the choice points that led up to the desired state in the eight-puzzle problem space. As described in Section 2, all of these lower subgoals are terminated when the higher goal is achieved. As each subgoal terminates, a chunk is built that tests the relevant conditions and produces a preference for one of the operators at the choice point.[14] This set of chunks encodes the path that was successful for the eight-puzzle-sd operator. In future problems, these chunks will act as search-control knowledge, leading the problem solver directly to the solution without any impasses or subgoals. Thus, ***Soar*** learns macro-operators, not as monolithic data structures, but as sets of chunks that determine at each point in the search which operator to select next. This differs from previous realizations of macros where a single data structure contains the macro, either as a list of operators, as in Korf's work, or as a triangle table, as in ***Strips*** (Fikes, Hart and Nilsson, 1972). Instead, for each operator in the macro-operator sequence, there is a chunk that causes it to be selected (and therefore applied) at the right time. On later problems (and even the same problem), these chunks control the search when they can, giving the appearance of macro problem solving, and when they cannot, the problem solver resorts to search. When the latter succeeds, more chunks are learned, and more of the macro table is covered. By representing macros as sets of independent productions that are learned when the appropriate problem arises, the processes of learning, storing, and using macros become both incremental and simplified.

Figure 8 shows the problem solving and learning that ***Soar*** does while performing iterative-deepening searches for the first three eight-puzzle-sd operators of an example problem. The figure shows the searches for which the depth is sufficient to implement each operator. The first eight-puzzle-sd operator, **place-blank**, moves the blank to the center. Without learning, this yields the search shown in the left column of the first row. During learning (the middle column), a chunk is first learned to avoid an operator that does not achieve the goal within the current depth limit (2). This is marked by a "−" and the number 1 in the figure. The unboxed numbers give the order that the chunks are learned, while the boxed numbers show where the chunks are used in later problem solving. Once the goal is achieved, signified by the darkened circle, a chunk is learned that prefers the first move over all other alternatives, marked by "+" in the figure. No chunk is learned for the final move to the goal since the only other alternative at that point has already been rejected, eliminating any choice, and thereby eliminating the need to learn a chunk. The right column shows that on a second attempt, chunk 2 applied to select the first operator. After the operator applied, chunk 1 applied to reject the operator that did not lead to the goal. This leaves only the operator that leads to the goal, which is selected and applied. In this scheme, the chunks control the problem solving within the subgoals that

[14] Additional chunks are created for the subgoals resulting from no-change impasses on the evaluate-object operators, such as the example chunk in Section 3.1.1, but these become irrelevant for this task once the rules that embody preferences are learned.

implement the eight-puzzle-sd operator, eliminating search, and thereby encoding a macro-operator.

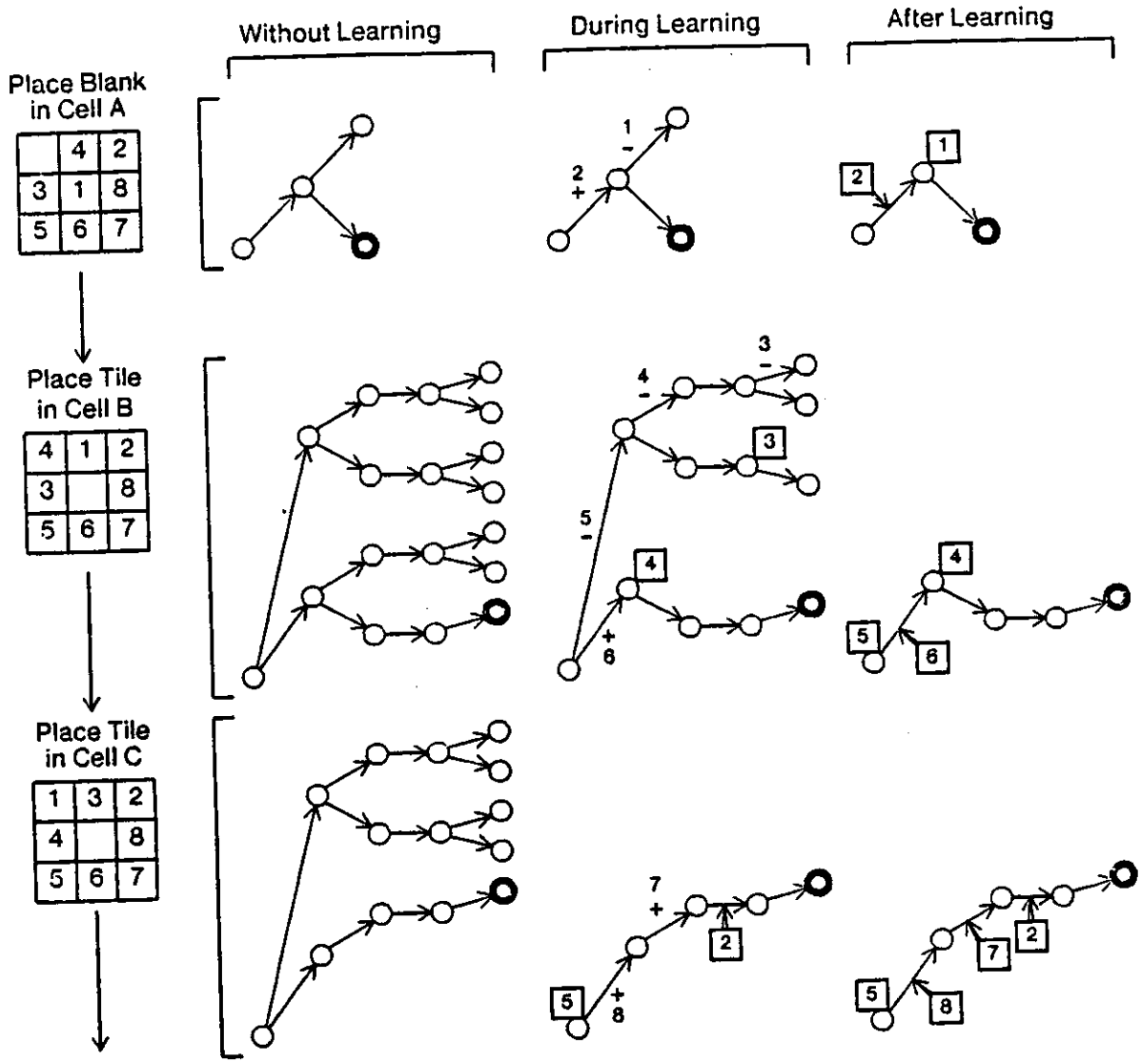

Figure 8: Searches performed for the first three **eight-puzzle-sd** operators in an example problem. The left column shows the search without learning. The horizontal arrows represent points in the search where no choice (and therefore no chunk) is required. The middle column shows the search during learning. A "+" signifies that a chunk was learned that preferred a given operator. A "–" signifies that a chunk was learned to avoid an operator. The boxed numbers show where a previously learned chunk was applied to avoid search during learning. The right column shows the search after learning.

The examples in the second and third rows of Figure 8 show more complex searches and demonstrate how the chunks learned during problem solving for one **eight-puzzle-sd** operator can reduce the search both within

that operator and within other operators. In all of these examples, a macro-operator is encoded as a set of chunks that are learned during problem solving and that will eliminate the search the next time a similar problem is presented.

In addition to learning chunks for each of the operator-selection decisions, *Soar* can learn chunks that directly implement instances of the operators in the **eight-puzzle-sd** problem space. They directly create a new state where the tiles have been moved so that the next desired tile is in place, a process that usually involves many Eight Puzzle moves. These chunks would be ideal macro-operators if it were not necessary to actually apply each **eight-puzzle** operator to a physical puzzle in the real world. As it is, the use of such chunks can lead to illusions about having done something that was not actually done. We have not yet implemented in *Soar* a general solution to the problem posed by such chunks. One possible solution — whose consequences we have not yet analyzed in depth — is to have chunking automatically turned off for any goal in which an action occurs that affects the outside world. For this work we have simulated this solution by disabling chunking for the **eight-puzzle** problem space. Only search-control chunks (generated for the **tie** problem space) are learned.

The searches within the **eight-puzzle** problem space can be controlled by a variety of different problem solving strategies, and any heuristic knowledge that is available can be used to avoid a brute-force search. Both iterative-deepening and breadth-first search[15] strategies were implemented and tested. Only one piece of search control was employed — do not apply an operator that will undo the effects of the previous operator. Unfortunately, *Soar* is too slow to be able to generate a complete macro table for the Eight Puzzle by search. *Soar* was unable to learn the eight macros in columns three and five in Figure 1. These macros require searches to at least a depth of eight.[16]

The actual searches used to generate the chunks for a complete macro table were done by having a user lead *Soar* down the path to the correct solution. At each resolve-tie subgoal, the user specified which of the tied operators should be evaluated first, insuring that the correct path was always tried first. Because the user specified which operator should be evaluated first, and not which operator should actually be applied, *Soar* proceeded to try out the choice by selecting the specified **evaluate-object** operator and entering an subgoal in which the relevant **eight-puzzle** operator was applied. *Soar* verified that the choice made by the user was correct by searching until the choice led to either success or failure. During the verification, the appropriate objects were automatically referenced so that a correct chunk was generated. This is analogous to the

---

[15] This was actually a *parallel* breadth-first search in which the operators at each depth were executed in parallel.

[16] Although some of the macros are fourteen operators long, not every operator selection requires a choice (some are forced moves) and, in addition, *Soar* is able to make use of transfer from previously learned chunks (Section 4.3).

explanation-based learning approach (for example, see De Jong, 1981 or Mitchell, Keller, & Kedar-Cabelli (1986)), though the explanation and learning processes differ.

Soar's inability to search quickly enough to complete the macro table autonomously is the one limitation on a claim to have replicated Korf's (1985a) results for the Eight Puzzle. This, in part, reflects a trade-off between speed (Korf's system) and generality (***Soar***). But it is also partially a consequence of our not using the fastest production-system technology available. Significant improvements in ***Soar's*** performance should be possible by reimplementing it using the software technology developed for ***Ops83*** (Forgy, 1984).

4.3. Chunk Generality and Transfer

Korf's (1985a) work on macro problem solving shows that a large class of problems — for example, all Eight Puzzle problems with the same desired state — can be solved efficiently using a table with a small number of macros. This is possible only because the macros ignore the positions of all tiles not yet in place. This degree of generality occurs in ***Soar*** as a direct consequence of implicit generalization. If the identities of the tiles not yet placed are not examined during problem solving, as they need not be, then the chunks will also not examine them. However, this does not tap all of the possible sources of generality in the Eight Puzzle. In the remainder of this subsection we will describe two additional forms of transfer available in the ***Soar*** implementation.

4.3.1. Different Goal States

One limitation on the generality of the macro table is that it can only be used to solve for the specific final configuration in Figure 3. Korf (1985a) described one way to overcome this limitation. For other desired states with the blank in the center it is possible to use the macro table by renumbering the tiles in the desired state to correspond to the ordering in Figure 3, and then using the same transformation for the initial state. In the ***Soar*** implementation this degree of generality occurs automatically as a consequence of implicit generalization. The problem solver must care that a tile is in its desired location, but it need not care which tile it actually is. The chunks learned are therefore independent of the exact numbering on the tiles. Instead they depend on the relationship between where the tiles are and where they should be.

For desired states that have the blank in a different position, Korf (1985a) described a three-step solution method. First find a path from the initial state to a state with the blank in the center; second, find a path from the desired state to the same state with the blank in the middle; and third, combine the solution to the first problem with the inverse of the solution to the second problem — assuming the inverse of every operator is both defined and known — to yield a solution to the overall problem. In ***Soar*** this additional degree of generality can be achieved with the learning of only two additional chunks. This is done by solving the problem using the following subgoals (see Figure 9 below): (a) get the blank in the middle, (b) get the first six

tiles into their correct positions, and (c) get the blank in its final position. The first 7 moves can be performed directly by the chunks making up the macro table, while the last step requires 2 additional chunks.

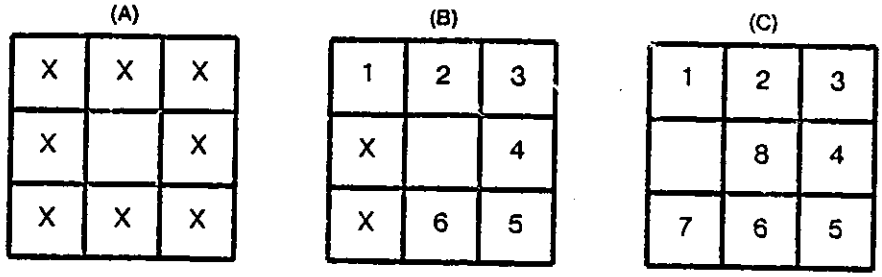

Figure 9: Problems with different goals states, with different positions of the blank, can be solved by: (a) moving the blank into the center, (b) moving the first six tiles into position, and (c) moving the blank into its desired position.

4.3.2. Transfer Between Macro-Operators

In addition to the transfer of learning between desired states, we can identify four different levels of generality that are based on increasing the amount of transfer that occurs *between* the macro-operators in the table: *no transfer, simple transfer, symmetry transfer (within column)*, and *symmetry transfer (across column)*. The lowest level, *no transfer*, corresponds to the generality provided directly by the macro table. It uses macro-operators quite generally, but shows no transfer between the macro operators. Each successive level has all of the generality of the previous level, plus one additional variety of transfer. The actual runs were done for the final level, which maximizes transfer. The number of chunks required for the other cases were computed by hand. Let us consider each of them in turn.

No transfer. The no-transfer situation is identical to that employed by Korf (1985a). There is no transfer of learning between macro-operators. In *Soar*, a total of 230 chunks would be required for this case.[17] This is considerably higher than the number of macro-operators (35) because one chunk must be learned for each operator in the table (if there is no search control) rather than for each macro-operator. If search control is available to avoid undoing the previous operator, only 170 chunks must be learned.

Simple transfer. Simple transfer occurs when two entries in the same column of the macro table end in exactly the same set of moves. For example, in the first column of Table 1, the macro that moves the blank to the center from the upper-right corner uses the macro-operator *ur* (column 0, row D in the table). The chunk learned for the second operator in this sequence, which moves the blank to the center from the position to the right of the center (by moving the center tile to the right), is dependent on the state of the board following the first operator, but independent of what the first operator actually was. Therefore, the chunk for the last half

[17] These numbers include only the chunks for the resolve-tie subgoals. If the chunks generated for the evaluate-object operators were included, the chunk counts given in this section would be doubled.

of this macro-operator is exactly the chunk/macro-operator in column 0, row E of the table. This type of transfer is always available in *Soar*, and reduces the number of chunks needed to encode the complete macro table from 170 to 112. The amount of simple transfer is greater than a simple matching of the terminal sequences of operators in the macros in Table 1 would predict because different macro operators of the same length as those in the table can be found that provide greater transfer.

Symmetry transfer (within column). Further transfer can occur when two macro-operators for the same subgoal are identical except for rotations or reflections. Figure 10 contains two examples of such transfer. The desired state for both is to move the 1 to the upper left corner. The X's represent tiles whose values are irrelevant to the specific subgoal and the arrow shows the path that the blank travels in order to achieve the subgoal. In (a), a simple rotation of the blank is all that is required, while in (b), two rotations of the blank must be made. Within both examples the pattern of moves remains the same, but the orientation of the pattern with respect to the board changes. The ability to achieve this type of transfer by implicit generalization is critically dependent upon the representation of the states (and operators) discussed in Section 3.3. The representation allows the topological relationships among the affected cells (which cells are next to which other cells) and the operators (which cells are affected by the operators) to be examined while the absolute locations of the cells and the names of the operators are ignored. This type of transfer reduces the number of required chunks from 112 to 83 over the simple-transfer case.

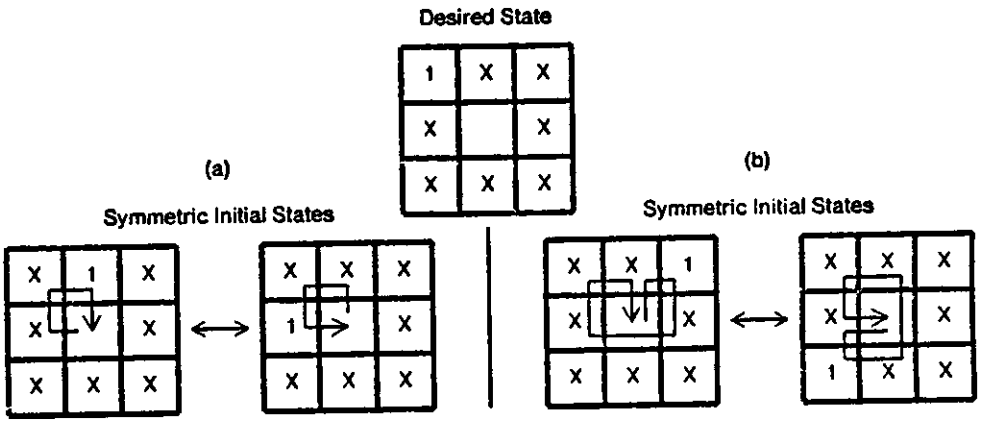

Figure 10: Two examples of within-column symmetry transfer.

Symmetry transfer (across column). The final level of transfer involves the carryover of learning between different subgoals. As shown by the example in Figure 11, this can involve far from obvious similarities between two situations. What is important in this case is: (1) that a particular three cells are not affected by the moves (the exact three cells can vary); (2) the relative position of the tile to be placed with respect to where it should be; and (3) that a previously placed piece that is affected by the moves gets returned to its

original position. Across-column symmetry transfer reduces the number of chunks to be learned from 83 to 61 over the within-column case.[18] Together, the three types of transfer make it possible for *Soar* to learn the complete macro table in only three carefully selected trials.

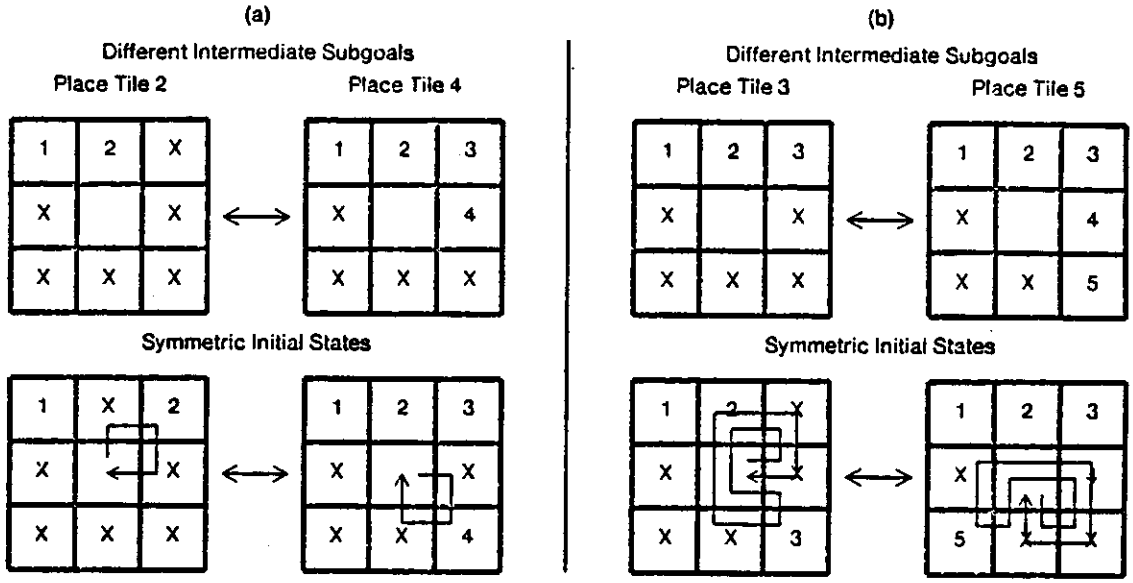

Figure 11: An example of across-column symmetry transfer.

Table 2 contains the macro-table structure of the chunks learned when all three levels of transfer are available (and search control to avoid undoing the previous operator is included). In place of operator sequences, the table contains numbers for the chunks that encode the macros. There is no such table actually in *Soar* — all chunks (productions) are simply stored, unordered, in production memory. The purpose of this table is to show the actual transfer that was achieved for the Eight Puzzle.

The order in which the subgoals are presented has no effect on the collection of chunks that are learned for the macro table, because if a chunk will transfer to a new situation (a different place in the macro table) the chunk that would have been learned in the new situation would be identical to the one that applied instead. Though this is not true for all tasks, it is true in this case. Therefore, we can just assume that the chunks are learned starting in the upper left corner, going top to bottom and left to right. The first chunk learned is number 1 and the last chunk learned is number 61. When the number for a chunk is highlighted, it stands for all of the chunks that followed in its first unhighlighted occurrence. For example, for tile 1 in position F, the chunks listed are 13, 12, 11, ***10***. However, ***10*** signifies the sequence beginning with chunk 10: 10, 9, 8, ***4***. The

[18] The number of chunks can be reduced further, to 54, by allowing the learning of macros that are not of minimum length. This increases the total path length by 2 for 14% of the problems, by 4 for 26% of the problems and 6 for 7% of the problems.

terminal *4* in this sequence signifies the sequence beginning with chunk 4: 4, 3, *1*. Therefore, the entire sequence for this macro is: 13, 12, 11, 10, 9, 8, 4, 3, 1.

		Tiles					
		0	1	2	3	4	5	6
Positions	A							
	B	2,1						
	C	*1*	4,3,*1*					
	D	*2*	7,6,5,*4*	15,14,*1*				
	E	*1*	10,9,8,*4*	18,17,16	34,33,32,31,30,29,*1*			
	F	*2*	13,12,11,*10*	21,20,19,*18*	40,39,38,37,36,35,*30*	*15*		
	G	*1*	*10*	23,22,*17*	46,45,44,43,42,41,*30*	*18*	61,60,59,58,56,55,*29*	
	H	*2*	*7*	26,25,24,*23*	54,53,52,51,50,49,48,47,46,*29*	*21*	*40*	*15*
	I	*1*	*4*	28,27,*22*	*51*	*23*	*46*	*18*

Table 2: Structure of the chunks that encode the macro table for the Eight Puzzle.

The abbreviated macro format used in Table 2 is more than just a notational convenience; it directly shows the transfer of learning between the macro-operators. Simple transfer and within-column symmetry transfer show up as the use of a macro that is defined in the same column. For example, the sequence starting with chunk 51 is learned in column 3 row H, and used in the same column in row I. The extreme case is column 0, where the chunks learned in the top row can be used for all of the other rows. Across-column symmetry transfer shows up as the reoccurrence of a chunk in a later column. For example, the sequence starting with chunk 29 is learned in column 3 (row E) and used in column 5 (row G). The extreme examples of this are columns 4 and 6 where all of the macros were learned in earlier columns of the table.

4.4. Other Tasks

The macro technique can also be used in the Tower of Hanoi (Korf, 1985a). The three-peg, three-disk version of the Tower of Hanoi has been implemented as a set of serially decomposable subgoals in *Soar*. In a single trial (moving three disks from one peg to another), *Soar* learns eight chunks that completely encode Korf's (1985a) macro table (six macros). Only a single trial was required because significant within and across column transfer was possible. The chunks learned for the three-peg, three-disk problem will also solve the three-peg, two-disk problem. These chunks also transfer to the final moves of the three-peg, N-disk problem when the three smallest disks are out of place. Korf (1985a) demonstrated the macro table technique on three additional tasks: the Fifteen Puzzle, Think-A-Dot and Rubik's Cube. The technique for learning and using macros in *Soar* should be applicable to all of these problems. However, the performance of the current implementation would require user-directed searches for the Fifteen Puzzle and Rubik's Cube because of the

size of the problems.

5. Conclusion

In this article we have laid out how chunking works in *Soar*. It is a learning mechanism that is based on the acquisition of rules from goal-based experience. As such, it is related to a number of other learning mechanisms. However, it obtains extra scope and generality from its intimate connection with a sophisticated problem solver (*Soar*) and the memory organization of the problem solver (a production system). This is the most important lesson of this research. The problem solver provides many things: the opportunities to learn, direction as to what is relevant (biases) and what is needed, and a consumer for the learned information. The memory provides a means by which the newly learned information can be integrated into the existing system and brought to bear when it is relevant.

In previous work we have demonstrated how the combination of chunking and *Soar* could acquire search-control knowledge (strategy acquisition) and operator implementation rules in both search-based puzzle tasks and knowledge-based expert systems tasks (Laird, Rosenbloom & Newell, 1984; Rosenbloom, Laird, McDermott, Newell, & Orciuch, 1985). In this paper we have provided a new demonstration of the capabilities of chunking in the context of the macro-operator learning task investigated by Korf (1985a). This demonstration shows how: (1) the macro-operator technique can be used in a general, learning problem solver without the addition of new mechanisms; (2) the learning can be incremental during problem solving rather than requiring a preprocessing phase; (3) the macros can be used for any goal state in the problem; and (4) additional generality can be obtained via transfer of learning between macro-operators, provided an appropriate representation of the task is available.

Although chunking displays many of the properties of a general learning mechanism, it has not yet been demonstrated to be a truly general learning mechanism. It can not yet learn new problem spaces or new representations, nor can it yet make use of the wide variety of potential knowledge sources, such as examples or analogous problems. Our approach to all of these insufficiences will be to look to the problem solving. Goals will have to occur in which new problem spaces and representations are developed, and in which different types of knowledge can be used. The knowledge can then be captured by chunking.

Acknowledgement

We would like to thank Pat Langley and Richard Korf for their comments on an earlier draft of this paper.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under contracts F33615-81-K-1539 and N00039-83-

C-0136, and by the Personnel and Training Research Programs, Psychological Sciences Division, Office of Naval Research, under contract number N00014-82C-0067, contract authority identification number NR667-477. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the Office of Naval Research, or the US Government.

References

Anderson, J. R. *The Architecture of Cognition.* Cambridge: Harvard University Press, 1983.

Anderson, J. R. Knoweldge compilation: The general learning mechanism. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Proceedings of the 1983 Machine Learning Workshop.*, 1983.

Anzai, Y. and Simon, H. A. The Theory of Learning by Doing. *Psychological Review*, 1979, *86*(2), 124-140.

Brown, J. S., & VanLehn, K. Repair theory: A generative theory of bugs in procedural skills. *Cognitive Science*, 1980, *4*, 379-426.

Carbonell, J. G., Michalski, R. S., & Mitchell, T. M. An overview of machine learning. In R. S. Michalski, J. G. Carbonell, T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach.* Palo Alto, CA: Tioga, 1983.

Chase, W. G. & Simon, H. A. Perception in chess. *Cognitive Psychology*, 1973, *4*, 55-81.

Davis, R. and King, J. An overview of production systems. In E. W. Elcock & D. Michie (Ed.), *Machine Intelligence 8.* New York: American Elsevier, 1976.

DeJong, G. Generalizations based on explanations. In *Proceedings of IJCAI-81.*, 1981.

Feigenbaum, E. A. and Feldman, J. (Eds.). *Computers and Thought.* New York: McGraw-Hill, 1963.

Fikes, R. E., Hart, P. E. and Nilsson, N. J. Learning and executing generalized robot plans. *Artificial Intelligence*, 1972, *3*(4), 251-288.

Forgy, C. L. *OPS5 Manual.* Computer Science Department, Carnegie-Mellon University, 1981.

Forgy, C. L. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 1982, *19*, 17-37.

Forgy, C. L. *The OPS83 Report* (Tech. Rep. #84-133). Carnegie-Mellon University Computer Science Department, May 1984.

Hayes, J. R. and Simon, H. A. Understanding complex task instructions. In Klahr, D. (Ed.), *Cognition and Instruction.* Hillsdale, NJ: Erlbaum, 1976.

Korf, R. E. Toward a model of representation changes. *Artificial Intelligence*, 1980, *14*, 41-78.

Korf, R. E. Macro-operators: A weak method for learning. *Artificial Intelligence*, 1985, *26*, 35-77.

Korf, R. E. Depth-first iterative-deepening: An optimal admissable tree search. *Artificial Intelligence*, 1985. In press.

Laird, J. E. *Universal Subgoaling.* Doctoral dissertation, Computer Science Department, Carnegie-Mellon

University, 1984.

Laird, J. E. and Newell, A. A universal weak method: Summary of results. In *Proceedings of IJCAI-83.* Los Altos, CA: Kaufmann, 1983.

Laird, J. E., and Newell, A. *A Universal Weak Method* (Tech. Rep. #83-141). Carnegie-Mellon University Computer Science Department, June 1983.

Laird, J. E., Newell, A., and Rosenbloom, P. S. Soar: An Architecture for General Intelligence. 1985. In preparation.

Laird, J. E., Rosenbloom, P. S., and Newell, A. Towards Chunking as a General Learning Mechanism. In *Proceedings of AAAI-84. National Conference on Artificial Intelligence.* Austin: American Association for Artificial Intelligence, 1984.

Langley, P. Learning Effective Search Heuristics. In *Proceedings of IJCAI-83.* Los Altos, CA: Kaufmann, 1983.

Lenat, D. *AM: An Artificial Intelligence Approach to Discovery in Mathematics as Heuristic Search.* Doctoral dissertation, Computer Science Department, Stanford University, 1976.

Lenat, D. B. Eurisko: A program that learns new heuristics and domain concepts. *Artificial Intelligence,* 1983, *21,* 61-98.

Lewis, C. H. *Production system models of practice effects.* Doctoral dissertation, University of Michigan, 1978.

Marsh, D. Memo functions, the graph traverser, and a simple control situation. In B. Meltzer & D. Michie (Ed.), *Machine Intelligence 5.* New York: American Elsevier, 1970.

McDermott, J. R1: A rule-based configurer of computer systems. *Artificial Intelligence,* 1982, *19,* 39-88.

Michie, D. "Memo" functions and machine learning. *Nature,* 1968, *218,* 19-22.

Miller, G. A. The magic number seven, plus or minus two: Some limits on our capacity for processing information. *Psychological Review,* 1956, *63,* 81-97.

Mitchell, T. M. Learning and Problem Solving. In *Proceedings of IJCAI-83.* Los Altos, CA: Kaufmann, 1983.

Mitchell, T. M., Keller, R. M., Kedar-Cabelli, S. T. Explanation-based generalization: A unifying view. *Machine Learning,* 1986, Vol. *1.* In press.

Neves, D. M. & Anderson, J. R. Knowledge compilation: Mechanisms for the automatization of cognitive skills. In Anderson, J. R. (Ed.), *Cognitive Skills and their Acquisition.* Hillsdale, NJ: Erlbaum, 1981.

Newell, A. Production Systems: Models of Control Structures. In Chase, W. (Ed.), *Visual Information Processing.* New York: Academic, 1973.

Newell, A. Reasoning, problem solving and decision processes: The problem space as a fundamental category. In R. Nickerson (Ed.), *Attention and Performance VIII.* Hillsdale, N.J.: Erlbaum, 1980. (Also available as CMU CSD Technical Report, Aug 79).

Newell, A. & Rosenbloom, P. S. Mechanisms of skill acquisition and the law of practice. In J. R. Anderson (Ed.), *Cognitive Skills and Their Acquisition.* Hillsdale, NJ: Erlbaum, 1981. (Also available as

Carnegie-Mellon University Computer Science Tech. Rep. #80-145).

Nilsson, N. *Principles of Artificial Intelligence.* Palo Alto, CA: Tioga, 1980.

Rendell, L. A. A new basis for state-space learning systems and a successfull implementation. *Artificial Intelligence*, 1983, *20*(4), 369-392.

Rich, E. *Artificial Intelligence.* New York: McGraw-Hill, 1983.

Rosenbloom, P. S. *The Chunking of Goal Hierarchies: A Model of Practice and Stimulus-Response Compatibility.* Doctoral dissertation, Carnegie-Mellon University, 1983. (Available as Carnegie-Mellon University Computer Science Tech. Rep. #83-148).

Rosenbloom, P. S., & Newell, A. The chunking of goal hierarchies: A generalized model of practice. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach, Volume II.* Los Altos, CA: Morgan Kaufmann Publishers, Inc., 1985. In press (Also available in *Proceedings of the Second International Machine Learning Workshop*, Urbana: 1983).

Rosenbloom, P. S., Laird, J. E., McDermott, J., Newell, A., and Orciuch, E. R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1985. In press (Also available in *Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems*, Denver: IEEE Computer Society, 1984, and as part of Carnegie-Mellon University Computer Science Tech. Rep. #85-110).

Smith, R. G., Mitchell, T. M., Chestek, R. A., and Buchanan, B. G. A Model for Learning Systems. In *Proceedings of IJCAI-77.*, 1977.

Sussman, G. J. *A Computer Model of Skill Acquisition.* New York: Elsevier, 1977.

Utgoff, P. E. *Shift of bias for inductive concept learning.* Doctoral dissertation, Rutgers University, October 1984.

van de Brug, A., Rosenbloom, P. S., & Newell, A. *Some Experiments with R1-Soar* (Tech. Rep.). Carnegie-Mellon University Computer Science Department, 1985. In preparation.

Waterman, D. A. Adaptive Production Systems. In *Proceedings of IJCAI-83.*, 1975.