CMU-CS-86-171

Soar: An Architecture for General Intelligence

John E. Laird
Department of Electrical Engineering and Computer Science
University of Michigan
(this work was done while at
Intelligent Systems Laboratory
Xerox Palo Alto Research Center)

Allen Newell
Department of Computer Science
Carnegie-Mellon University

Paul S. Rosenbloom
Knowledge Systems Laboratory,
Departments of Computer Science and Psychology
Stanford University

3 December 1986

Abstract

The ultimate goal of work in cognitive architecture is to provide the foundation for a system capable of general intelligent behavior. That is, the goal is to provide the underlying structure that would enable a system to perform the full range of cognitive tasks, employ the full range of problem-solving methods and representations appropriate for the tasks, and learn about all aspects of the tasks and its performance on them. In this article we present Soar, an implemented proposal for such an architecture. We describe its organizational principles, the system as currently implemented, and demonstrations of its capabilities.

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. Additional partial support was provided by the Sloan Foundation and some computing support was supplied by the SUMEX-AIM facility (NIH grant number RR-00785). 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, the Sloan Foundation, the National Institute of Health, or the US Government.

Table of Contents

List of Figures

Soar: An Architecture for General Intelligence[1]

Soar is an architecture for a system that is to be capable of general intelligence. Soar is to be able to: (1) work on the full range of tasks, from highly routine to extremely difficult open-ended problems; (2) employ the full range of problem-solving methods and representations required for these tasks; and (3) learn about all aspects of the tasks and its performance on them. Soar has existed since mid 1982 as an experimental software system (in Ops5 and Lisp), initially as Soar 1 [31, 32], then as Soar 2 [29, 35], and currently as Soar 4 [30]. Soar realizes the capabilities of a general intelligence only in part, with significant aspects still missing. But enough has been attained to make worthwhile an exposition of the current system.

Soar is one of many artificial intelligence (AI) systems that have attempted to provide an appropriate organization for intelligent action. It is to be compared with other organizations that have been put forth, especially recent ones: MRS [22]; Eurisko [38, 39]; blackboard architectures [4, 16, 24, 56]; Pam/Pandora [79] and Nasl [40]. Soar is also to be compared with machine learning systems that involve some form of problem solving [10, 15, 37, 45, 46]. Especially important are existing systems that engage in some significant form of both problem solving and learning, such as: ACT* [2]; and Repair theory [8], embodied in a system called Sierra [77]. ACT* and Repair theory are both psychological theories of human cognition. Soar, whose antecedents have layed a strong role in cognitive theories, is also intended as the basis for a psychological theory, but this aspect is not yet well developed and is not discussed further.

Soar has its direct roots in a continuous line of research that starts back in 1956 with the Logic Theorist [53] and list processing (the IPLs) [55]. The line goes through GPS [17, 54], the general theory of human problem solving [51] and the development of production systems, PSG [48], Psanls [66] and the Ops series [20, 21]. Its roots include the emergence of the concept of cognitive architecture [48], the Instructable Production System project [67, 68] and the extension of the concept of problem spaces to routine behavior [49]. They also include research on cognitive skill and its acquisition [11, 35, 50, 63]. Soar is the current culmination of all this work along the dimension of architectures for intelligence.

Soar's behavior has already been studied over a range of tasks and methods (Figure 1), which sample its intended range, though unsystematically. Soar has been run on most of the standard AI toy problems [29, 31]. These tasks elicit knowledge-lean, goal-oriented behavior. Soar has been run on a small number of routine, essentially algorithmic, tasks, such as matching forms to objects, doing elementary syllogisms, and searching for a root of a quadratic equation. Soar has been run on knowledge-intensive tasks that are typical of current expert systems. The tactic has been to do the same task as an existing AI expert system, using the same

[1]We would like to thank David Steier and Danny Bobrow for their helpful comments on earlier drafts of this article, and Randy Gobbel for assistance in the final preparation of the manuscript.

knowledge. The main effort has been R1-Soar [65], which showed how Soar would realize a classical expert system, R1, which configures Vax and PDP-11 computers at Digital Equipment Corporation [3, 41]. R1 is a large system and R1-Soar was only carried far enough in its detailed coverage (about 25% of the functionality of R1) to make clear that it could be extended to full coverage if the effort warranted [75]. In addition, Soar versions of other substantial systems are operational although not complete: Neomycin [13], which itself is a reworking of the classical expert system, Mycin [71]; and Designer [26], an AI system for designing algorithms. Soar has also been given some tasks that have played important roles in the development of artificial intelligence: natural-language parsing, concept learning, and predicate-calculus theorem proving. In each case the performance and knowledge of an existing system has been adopted as a target in order to learn as much as possible by comparison: Dypar [6], Version Spaces [44] and Resolution [60]. These have so far been small demonstration systems; developing them to full-scale performance has not seemed profitable.

A variety of different representations for tasks and methods can be realized within Soar's architecturally given procedural and declarative representations. Essentially all the familiar weak methods [47] have been realized with Soar and used on several tasks [31]. In larger tasks, such as R1-Soar, different weak methods occur in different subparts of the task. Alternative decompositions of a task into subtasks [75] and alternative basic representations of a task have also been explored [31], but not intensively.

Soar has a general mechanism for learning from experience [33, 36] which applies to any task it performs. Thus, it can improve its performance in all of the tasks listed. Detailed studies of its learning behavior have been done on several tasks of varying characteristics of size and task-type (games, puzzles, expert-system tasks). This single learning mechanism produces a range of learning phenomena, such as improvement in related tasks (across-task transfer); improvement even within the learning trial (within-trial transfer); and the acquisition of new heuristics, operator implementations and macro-operators.

Several basic mechanisms of cognition have not yet been demonstrated with Soar. Potentially, each such mechanism could force the modification of the architecture, although we expect most of them to be realized without major extension. Some of the most important missing aspects are deliberate planning, as developed in artificial-intelligence systems [69]; the automatic acquisition of new tasks [23]; the creation of new task representations [1, 27]; extension to additional types of learning (e.g., by analysis, instruction, example, reading); and the ability to recover from errors in learning (which in Soar occurs by overgeneralization [34]). It is useful to list these lacunae, not just to indicate present limitations on Soar, but to establish the intended scope of the system. Soar is to operate throughout the entire spectrum of cognitive tasks.

The first section of this paper gives a preview of the features of Soar. The second section describes the Soar architecture in detail. The third section discusses some examples in order to make clear Soar's structure and operation. The final section concludes with a list of the principal hypotheses underlying the design of Soar.

Small, knowledge-lean tasks (typical AI toy tasks):
Blocks world, eight puzzle, eight queens, labeling line drawings (constraint satisfaction), magic squares, missionaries and cannibals, monkey and bananas, picnic problem, robot location-finding, three wizards problem, tic-tac-toe, Tower of Hanoi, water-jug task

Small routine tasks:
Expression unification, root finding, sequence extrapolation, syllogisms, Wason verification task

Knowledge-intensive expert-system tasks:
R1-Soar: 3300 rule industrial expert system (25% coverage)
Neomycin: Revision of Mycin (initial version)
Designer: Designs algorithms (initial version)

Miscellaneous AI tasks:
Dypar-Soar: Natural language parsing program (small demo)
Version-spaces: Concept formation (small demo)
Resolution theorem-prover (small demo)

Multiple weak methods with variations, most used in multiple small tasks:
Generate and test, AND/OR search, hill climbing (simple and steepest-ascent), means-ends analysis, operator subgoaling, hypothesize and match, breadth-first search, depth-first search, heuristic search, best-first search, A*, progressive deepening (simple and modified), B* (progressive deepening), minimax (simple and depth-bounded), alpha-beta, iterative deepening, B*

Multiple organizations and task representations:
Eight puzzle, picnic problem, R1-Soar

Learning:
Learns on all tasks it performs by a uniform method (chunking)
  Detailed studies on eight puzzle, R1-Soar, tic-tac-toe, Korf macro-operators
Types of learning:
  Improvement with practice, within-task transfer, across-task transfer, strategy acquisition, operator implementation, macro-operators, explanation-based generalization

Major aspects still missing:
Deliberate planning, automatic task acquisition, creating representations, varieties of learning, recovering from overgeneralization, interaction with external task environment

Figure 1: Summary of Soar performance scope.

1. Preview

In common with the mainstream of problem-solving and reasoning systems in AI, Soar has an explicit symbolic representation of its tasks, which it manipulates by symbolic processes. It encodes its knowledge of the task environment in symbolic structures and attempts to use this knowledge to guide its behavior. It has a general scheme of goals and subgoals for representing what the system wants to achieve, and for controlling its behavior.

Beyond these basic communalities, Soar embodies mechanisms and organizational principles that express distinctive hypotheses about the nature of the architecture for intelligence. These hypotheses are shared by other systems to varying extents, but taken together they determine Soar's unique position in the space of possible architectures. We preview here these main distinctive characteristics of Soar. The full details of all these features will be given in the next section on the architecture.

1.1. Uniform task representation by problem spaces

In Soar, every task of attaining a goal is formulated as finding a desired state in a *problem space* (a space with a set of operators that apply to a current state to yield a new state) [49]. Hence, all tasks take the form of heuristic search. Routine procedures arise, in this scheme, when enough knowledge is available to provide complete search control, i.e., to determine the correct operator to be taken at each step. In AI, problem spaces are commonly used for genuine problem solving [18, 51, 57, 58, 59, 72], but procedural representations are commonly used for routine behavior. For instance, problem-space operators are typically realized by Lisp code. In Soar, on the other hand, complex operators are implemented by problem spaces (though sufficiently simple operators can be realized directly by rules). The adoption of the problem space as the fundamental organization for *all* goal-oriented symbolic activity (called the *Problem Space Hypothesis* [49]) is a principal feature of Soar.

Figure 1-1 provides a schematic view of the important components of a problem-space search for the *eight puzzle*. The lower, triangular portion of the figure represents the search in the eight puzzle problem space, while the upper, rectangular portion represents the knowledge involved in the definition and control of the search. In the eight puzzle, there are eight numbered tiles and one space on a three-by-three board. The states are different configurations of the tiles on the board. The operators are the movements of an adjacent tile into the space (up, down, left and right). In the figure, the states are represented by schematic boards and the operators are represented by arrows.

Problem-space search occurs in the attempt to attain a goal. In the eight puzzle the goal is a desired state representing a specific configuration of the tiles — the darkened board at the right of the figure. In other tasks, such as chess, where checkmate is the goal, there are many disparate desired states, which may then be represented by a test procedure. Whenever a new goal is encountered in solving a problem, the problem solver begins at some initial state in the new problem space. For the eight puzzle, the initial state is just a particular configuration of the tiles. The problem-space search results from the problem solver's application of operators in an attempt to find a way of moving from its initial state to one of its desired states.

Only the current position (in Figure 1-1, it is the board pointed to by the downward arrow from the knowledge box) exists on the physical board, and Soar can generate new states only by applying the operators.

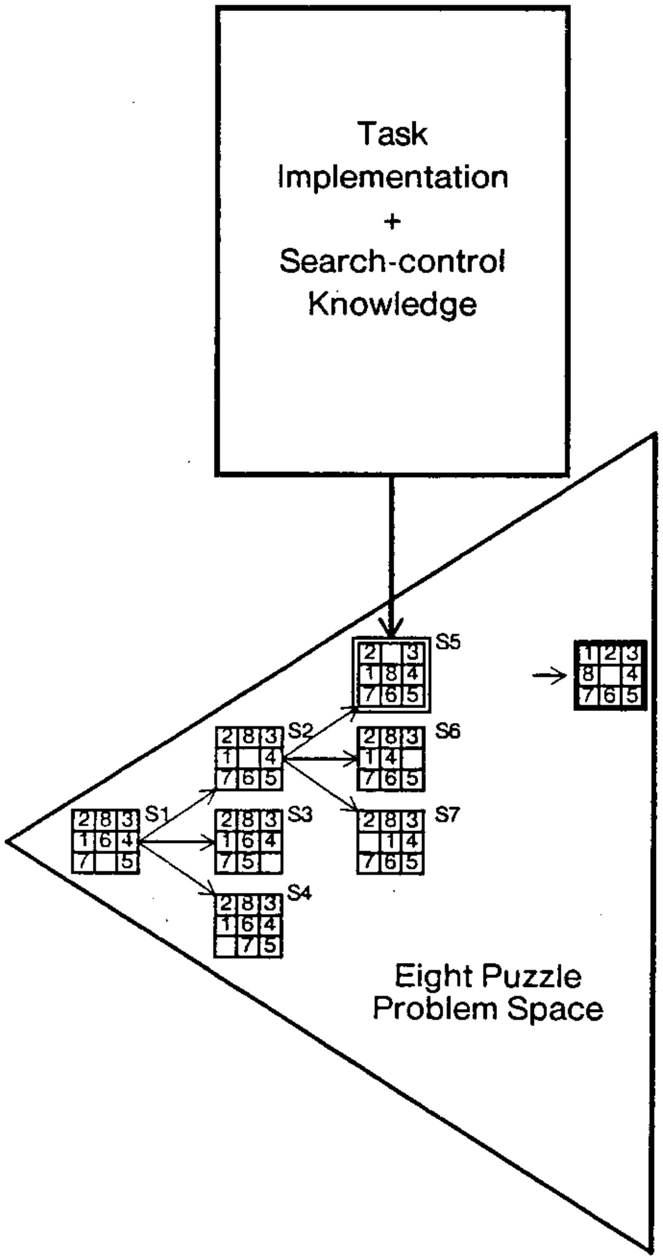

Task Implementation + Search-control Knowledge
S1
S2
S3
S4
S5
S6
S7
Eight Puzzle Problem Space

Figure 1-1: The structure of problem-space search for the eight puzzle.

Likewise, the states in a problem space, except the current state and possibly a few remembered states, do not preexist as data structures in the problem solver, so they must be generated by applying operators to states that do exist.

1.2. Any decision can be an object of goal-oriented attention

All decisions in Soar relate to searching a problem space (selection of operators, selection of states, etc.). The box in Figure 1-1 represents the knowledge that can be immediately brought to bear to make the decisions in a particular space. However, a subgoal can be set up to make any decision for which the immediate knowledge is insufficient. For instance, looking back to state S1, three moves were possible: moving a tile adjacent to the blank left, right or down. If the knowledge was not available to select which move to try, then a subgoal to select the operator would have been set up. Or, if the operator to move a tile left had been selected, but it was not known immediately how to perform that operator, then a subgoal would have been set up to do that. (The moves in the eight puzzle are too simple to require this, but many operators are more complex, e.g., an operator to factor a polynomial in an algebraic task.) Or, if the left operator had been applied and Soar attempted to evaluate the result, but the evaluation was too complicated to compute directly, then a subgoal would have been set up to obtain the evaluation. Or, to take just one more example, if Soar had attempted to apply an operator that was illegal at state S1, say to move tile 1 to the position of tile 2, then it could have set up a subgoal to satisfy the preconditions of the operator (that the position of tile 2 be blank).

In short, a subgoal can be set up for any problematic decision, a property we call *universal subgoaling*. Since setting up a goal means that a search can be conducted for whatever information is needed to make the decision, Soar can be described as having no fixed bodies of knowledge to make any decision (as in writing a specific Lisp function to evaluate a position or select among operators). The ability to search in subgoals also implies that further subgoals can be set up within existing subgoals so that the behavior of Soar involves a tree of subgoals and problem spaces (Figure 1-2). Because many of these subgoals address how to make control decisions, this implies that Soar can reflect [73] on its own problem-solving behavior, and do this to arbitrary levels [64].

1.3. Uniform representation of all long-term knowledge by a production system

There is only a single memory organization for all long-term knowledge, namely, a production system [9, 14, 25, 42, 78]. Thus, the boxes in Figures 1-1 and 1-2 are filled in with a uniform production system. Productions deliver control knowledge, as when a production action rejects an operator that leads back to the prior position. Productions also provide procedural knowledge for simple operators, such as the eight-puzzle moves, which can be accomplished by two productions, one to create the new state and put the changes in place and one to copy the unchanged tiles. (As noted above, more complex operators are realized by operating in an implementation problem space.) The data structures examinable by productions — that is, the pieces of knowledge in declarative form — are all in the production system's short-term working memory. However, the long-term storage of this knowledge is in productions which have actions that generate the data structures.

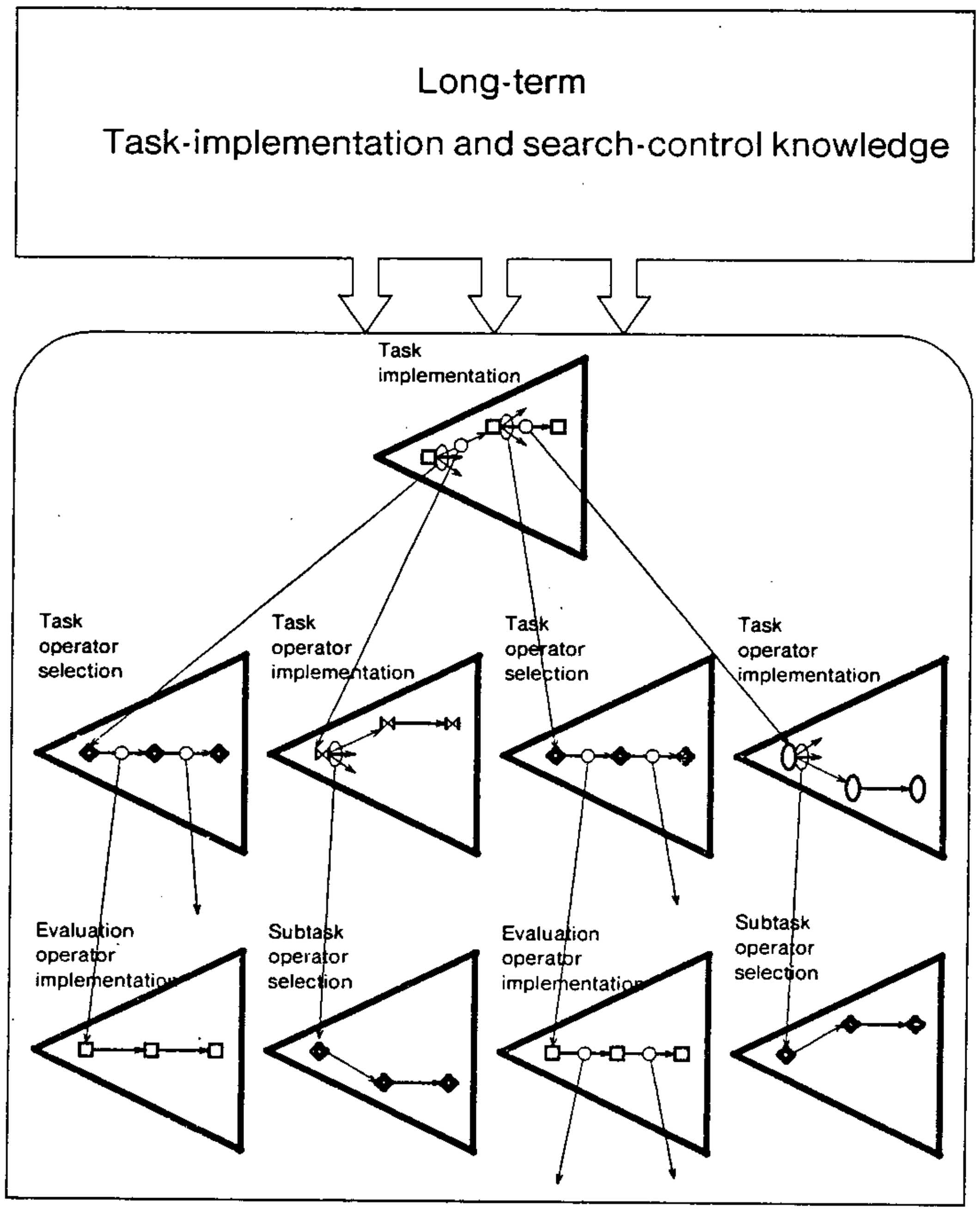

Long-term
Task-implementation and search-control knowledge
Task implementation
Task operator selection
Task operator implementation
Task operator selection
Task operator implementation
Evaluation operator implementation
Subtask operator selection
Evaluation operator implementation
Subtask operator selection

Figure 1-2: The tree of subgoals and their problem spaces.

Soar employs a specialized production system (a modified version of Ops5 [20]). All satisfied productions are fired in parallel, without conflict resolution. Productions can only add data elements to working memory. All modification and removal of data elements is accomplished by the architecture.

1.4. Knowledge to control search expressed by preferences

Search-control knowledge is brought to bear by the additive accumulation (via production firings) of data elements in working memory. One type of data element, the *preference*, represents knowledge about how Soar should behave in its current situation (as defined by a current goal, problem space, state and operator). For instance, the rejection of the move that simply returns to the prior state (in the example above) is encoded as a rejection preference on the operator. The preferences admit only a few concepts: acceptability, rejection, better (best, worse and worst), and indifferent. The architecture contains a fixed *decision procedure* for interpreting the set of accumulated preferences to determine the next action. This fixed procedure is simply the embodiment of the semantics of these basic preference concepts and contains no task-dependent knowledge.

1.5. All goals arise to cope with impasses

Difficulties arise, ultimately, from a lack of knowledge about what to do next (including of course knowledge that problems cannot be solved). In the immediate context of behaving, difficulties arise when problem solving cannot continue — when it reaches an *impasse*. Impasses are detectable by the architecture, because the fixed decision procedure concludes successfully only when the knowledge of how to proceed is adequate. The procedure fails otherwise (i.e., it detects an impasse). At this point the architecture creates a goal for overcoming the impasse. For example, each of the subgoals in Figure 1-2 is evoked because some impasse occurs: the lack of sufficient preferences between the three task operators creates a tie impasse; the failure of the productions in the task problem space to carry out the selected task operator leads to a no-change impasse; and so on.

In Soar, goals are created only in response to impasses. Although there are only a small set of architecturally distinct impasses (four), this suffices to generate all the types of subgoals. Thus, all goals arise from the architecture. This principle of operation, called *automatic subgoaling*, is the most novel feature of the Soar architecture, and it provides the basis for many other features.

1.6. Continuous monitoring of goal termination

The architecture continuously monitors for the termination of all active goals in the goal hierarchy. Upon detection, Soar proceeds immediately from the point of termination. For instance, in trying to break a tie between two operators in the eight puzzle, a subgoal will be set up to evaluate the operators. If in examining the first operator a preference is created that rejects it, then the decision at the higher level can, and will, be made immediately. The second operator will be selected and applied, cutting off the rest of the evaluation and comparison process. All of the working-memory elements local to the terminated goals are automatically removed.

Immediate and automatic response to the termination of any active goal is rarely used in AI systems because of its expense. Its (efficient) realization in Soar depends strongly on automatic subgoaling.

1.7. The basic problem-solving methods arise directly from knowledge of the task

Soar realizes the so-called weak methods, such as hill climbing, means-ends analysis, alpha-beta search, etc., by adding search-control productions that express, in isolation, knowledge about the task (i.e., about the problem space and the desired states). The structure of Soar is such that there is no need for this knowledge to be organized in separate procedural representations for each weak method (with a selection process to determine which one to apply). For example, if knowledge exists about how to evaluate the states in a task, and the consequences of evaluation functions are understood (prefer operators that lead to states with higher evaluations), then Soar exhibits a form of hill climbing. This general capability is another novel feature of Soar.

1.8. Continuous learning by experience through chunking

Soar learns continuously by automatically and permanently caching the results of its subgoals as productions. Thus, consider the tie-impasse between the three task operators in Figure 1-2, which leads to a subgoal to break that tie. The ultimate result of the problem solving in this subgoal is a preference (or preferences) that resolves the tie impasse in the top space and terminates the subgoal. Then a production is automatically created that will deliver that preference (or preferences) again in relevantly similar situations. If the system ever again reaches a similar situation, no impasse will occur (hence no subgoal and no problem solving in a subspace) because the appropriate preferences will be generated immediately.

This mechanism is directly related to the phenomenon called *chunking* in human cognition [63], whence its name. Structurally, chunking is a limited form of practice learning. However, its effects turn out to be wide-ranging. Because learning is closely tied to the goal scheme and universal subgoaling — which provide an extremely fine-grained, uniformly structured, and comprehensive decomposition of tasks on which the learning can work — Soar learns both operator implementations and search control. In addition, the combination of the fine-grained task decomposition with an ability to abstract away all but the relevant features allows Soar to exhibit significant transfer of learning to new situations, both within the same task and between similar tasks. This ability to combine learning and problem solving has produced the most striking experimental results so far in Soar [33, 36, 62].

2. The Soar Architecture

In this section we describe the Soar architecture systematically from scratch, depending on the preview primarily to have established the central role of problem spaces and production systems. We will continue to use the eight puzzle as the example throughout.

2.1. The Architecture for Problem Solving

Soar is a *problem-solving architecture*, rather than just an architecture for symbolic manipulation within which problem solving can be realized by appropriate control. This is possible because Soar accomplishes all of its tasks in problem spaces.

To realize a task as search in a problem space requires a fixed set of *task-implementation* functions, involving the retrieval or generation of: (1) problem spaces, (2) problem-space operators, (3) an initial state representing the current situation, and (4) new states that result from applying operators to existing states. To control the search requires a fixed set of *search-control* functions, involving the selection of: (1) a problem space, (2) a state from those directly available, and (3) an operator to apply to the state. Together, the task implementation and search-control functions are sufficient for problem-space search to occur. The quality and efficiency of the problem solving will depend on the nature of the selection functions.

The task-implementation and search-control functions are usually interleaved. Task implementation generates (or retrieves) new problem spaces, states and operators; and then search control selects among the alternatives generated. Together they completely determine problem-solving behavior in a problem space. Thus, as Figure 2-1 shows, the behavior of Soar on the eight puzzle can be described as a sequence of such acts. Other important functions must be performed for a complete system: goal creation, goal selection, goal termination, memory management and learning. None of these are included in Soar's search-control or task-implementation acts. Instead, they are handled automatically by the architecture, and hence are not objects of volition for Soar. They are described at the appropriate places below.

The deliberative acts of search-control together with the knowledge for implementing the task are the locus of intelligence in Soar. As indicated earlier in Figure 1-1, search-control and task-implementation knowledge is brought to bear on each step of the search. Depending on how much search-control knowledge the problem solver has and how effectively it is employed, the search in the problem space will be narrow and focused. or broad and random. If focused enough, the behavior is routine.

Figure 2-2 shows a block diagram of the architecture that generates problem-space search behavior. There is a *working memory* that holds the complete processing state for problem solving in Soar. This has three components: (1) a context stack that specifies the hierarchy of active goals, problem spaces, states and

```
[Retrieve the eight-puzzle problem space]
Select eight-puzzle as problem space
[Generate S1 as the initial state]
Select S1 as state
[Retrieve the operators Down, Left, Right]
Select Down as operator
[Apply operator (generate S2)]
Select Left as operator
[Apply operator (generate S3)]
Select Right as operator
[Apply operator (generate S4)]
Select S2 as state
[Retrieve the operators Down, Left, Right]
Select Down as operator
[Apply operator (generate S5)]
Select Left as operator
[Apply operator (generate S6)]
Select Right as operator
[Apply operator (generate S7)]
Select S7 as state
...
```

Figure 2-1: Problem-space trace in the eight puzzle. (Task implementation steps are bracketed.)

operators; (2) objects, such as goals and states (and their subobjects); and (3) preferences that encode the procedural search-control knowledge. The processing structure has two parts. One is the *production memory*, which is a set of productions that can examine any part of working memory, add new objects and preferences, and augment existing objects, but cannot modify the context stack. The second is a fixed *decision procedure* that examines the preferences and the context stack, and changes the context stack. The productions and the decision procedure combine to implement the search-control functions. Two other fixed mechanisms are shown in the figure: a *working-memory manager* that deletes elements from working memory, and a *chunking mechanism* that adds new productions.

Soar is embedded within Lisp. It includes a modified version of the Ops5 production system language plus additional Lisp code for the decision procedure, chunking, the working-memory manager, and other Soar-specific features. The Ops5 matcher has been modified to significantly improve the efficiency determining satisfied productions [70]. The total amount of Lisp code involved, measured in terms of the size of the source code, is approximately 255 kilobytes — 70 kilobytes of unmodified Ops5 code, 30 kilobytes of modified Ops5 code, and 155 kilobytes of Soar code. Soar runs in CommonLisp, FranzLisp, Interlisp and ZetaLisp on most of the appropriate hardware (Unix Vax, VMS Vax, Xerox D-machines, Symbolics 3600s, TI Explorers, IBM RTPCs, Apollo and Sun workstations).

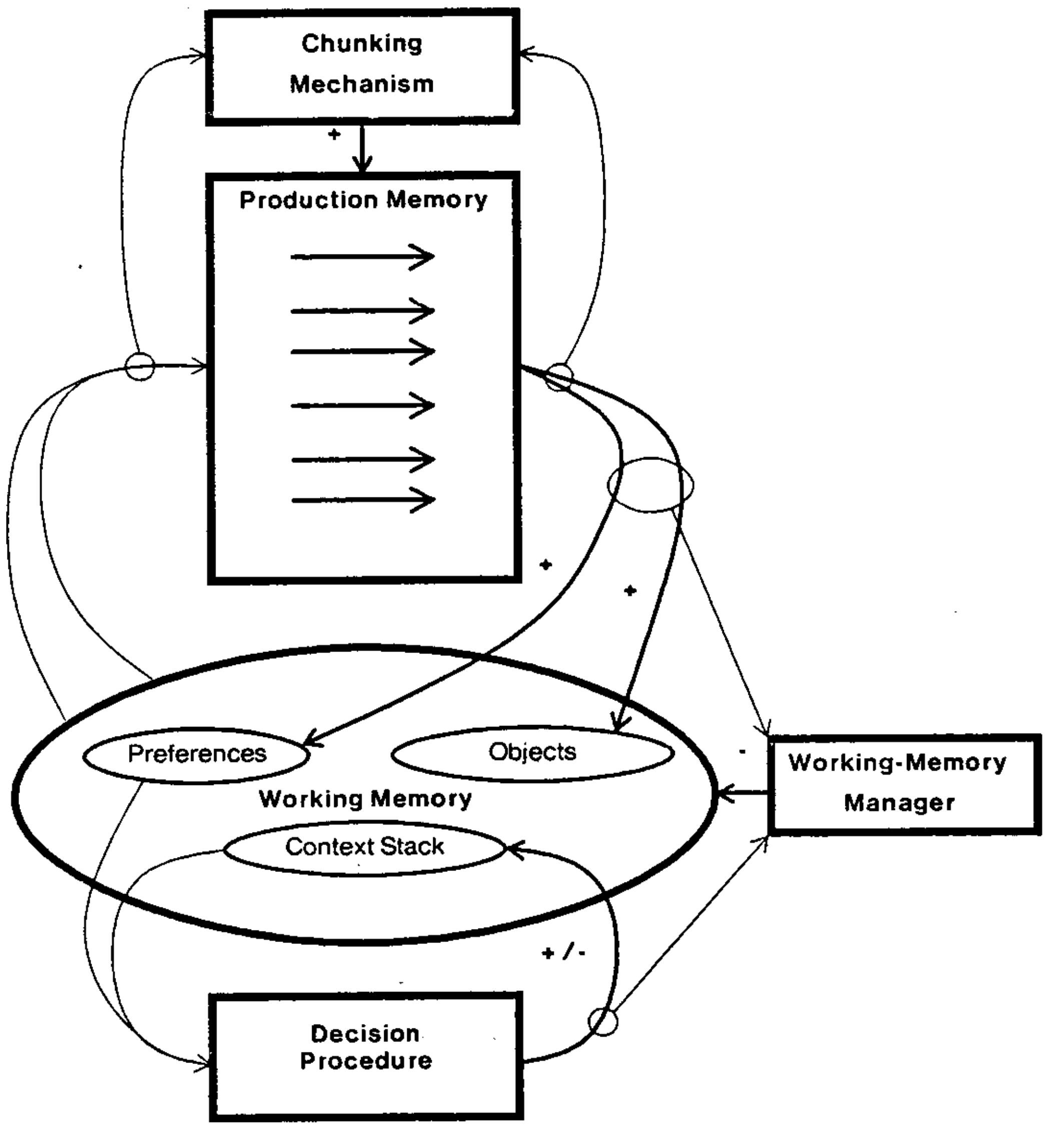


Figure 2-2: Architectural structure of Soar.

2.2. The Working Memory

Working memory consists of a *context stack*, a set of *objects* linked to the context stack, and *preferences*. Figure 2-3 shows a graphic depiction of a small part of working memory during problem solving on the eight puzzle. The context stack contains the hierarchy of active contexts (the boxed structures). Each context contains four *slots*, one for each of the different *roles:* goal, problem space, state and operator. Each slot can be occupied either by an object or by the symbol *undecided*, the latter meaning that no object has been selected for that slot. The object playing the role of the goal in a context is the current goal for that context; the object playing the role of the problem-space is the current problem space for that context and so on. The top context contains the highest goal in the hierarchy. The goal in each context below the top context is a subgoal of the context above it. In the figure, G1 is the current goal of the top context, P1 is the current problem space, S1 is the current state, and the current operator is undecided. In the lower context, G2 is the current goal (and a subgoal of G1). Each context has only one goal for the duration of its existence, so the context stack doubles as the goal stack.

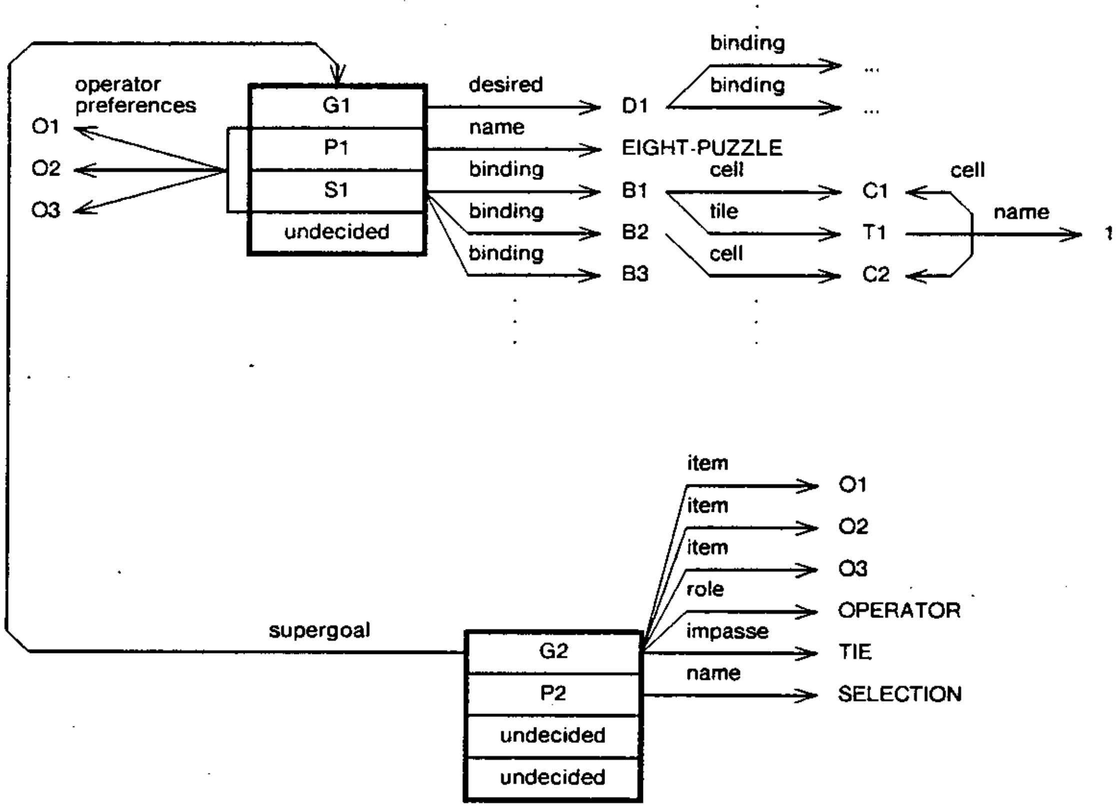


Figure 2-3: Snapshot of fragment of working memory.

The basic representation is object-centered. An object, such as a goal or a state, consists of a symbol, called its *identifier*, and a set of augmentations. An augmentation is a labeled relation (the *attribute*) between the object (the *identifier*) and another symbol (the *value*), i.e., an identifier-attribute-value triple. In the figure, G1 is augmented with a desired state, D1, which is itself an object that has its own augmentations (augmentations are directional, so G1 is not in an augmentation of D1, even though D1 is in an augmentation of G1). The attribute symbol may also be specified as the identifier of an object. Typically, however, situations are characterized by a small fixed set of attribute symbols — here, impasse, name, operator, binding, item, and role — that play no other role than to provide discriminating information. An object may have any number of augmentations, and the set of augmentations may change over time.[2]

A preference is a more complex data structure with a specific collection of eight architecturally-defined relations between objects. Three preferences are shown in the figure, one each for objects O1, O2, and O3. The preferences in the figure do not show their full structure (shown later in Figure 2-7), only the context in which they are applicable (any context containing problem space P1 and state S1).

The actual representation of objects in working memory is shown in Figure 2-4.[3] Working memory is a set — attempting to add an existing element does not change working memory. Each element in working memory represents a single augmentation. To simplify the description of objects, we group together all augmentations of the same object into a single expression. For example, the first line of Figure 2-4 contains a single expression for the four augmentations of goal G1. The first component of an object is a *class name* that distinguishes different types of objects. For example, goal, desired, problem-space, and state are the class names of the first four objects in Figure 2-4. Class names do not play a semantic role in Soar, although they allow the underlying matcher to be more efficient. Following the class-name is the identifier of the object. The goal has the current goal as its identifier. Following the identifier is an unordered list of attribute-value pairs, each attribute being prefaced by an up-arrow (↑). An object may have more than one value for a single attribute, as does state S1 in Figure 2-4, yielding a simple representation of sets.

The basic attribute-value representation in Soar leaves open how to represent task states. As we shall see later, the representation plays a key role in determining the generality of learning in Soar. The generality is maximized when those aspects of a state that are functionally independent are represented independently. In the eight puzzle, both the structure of the board and the actual tiles do not change from state to state in the real world. Only the location of a tile on the board changes, so the representation should allow a tile's location to change without changing the structure of the board or the tiles. Figure 2-5 contains a detailed graphic example of one representation of a state in the eight puzzle that captures this structure. The state it represents is shown in the lower left-hand corner. The board in the eight puzzle is represented by nine *cells* (the 3x3 square at the bottom of the figure), one for each of the possible locations for the tiles. Each cell is connected via augmentations of type cell to its neighboring cells (only a few labels in the center are actually filled in). In addition, there are nine *tiles* (the horizontal sequence of objects just above the cells), named 1-8, and blank (represented by a small box in the figure). The connections between the tiles and cells are specified by objects called *bindings*. A given state, such as S1 at the top of the figure, consists of a set of nine bindings (the

[2]The extent of the memory structure is necessarily limited by the physical resources of the problem solver, but currently this is assumed not to be a problem and mechanisms have not been created to deal with it.

[3]Some basic notation and structure is inherited from Ops5.

```
(goal G1 ↑problem-space P1 ↑state S1 ↑operator undecided ↑desired D1)
  (desired D1 ↑binding DB1 ↑binding DB2 ...)
 (problem-space P1 ↑name eight-puzzle)
(state S1 ↑binding B1 B2 B3 ...)
 (binding B1 ↑cell C1 ↑tile T1)
  (cell C1 ↑cell C2 ...)
  (tile T1 ↑name 1)
 (binding B2 ↑cell C2 ...)
  (cell C2 ↑cell C1 ...)
 (binding B3 ...)
 ...
(preference ↑object O1 ↑role operator ↑value acceptable
        ↑problem-space P1 ↑state S1)
(preference ↑object O2 ↑role operator ↑value acceptable
        ↑problem-space P1 ↑state S1)
(preference ↑object O3 ↑role operator ↑value acceptable
        ↑problem-space P1 ↑state S1)
 (operator O1 ...)
 (operator O2 ...)
 (operator O3 ...)

(goal G2 ↑problem-space P2 ↑state undecided ↑operator undecided
        ↑supergoal G1 ↑role operator ↑impasse tie
        ↑item O3 ↑item O2 ↑item O1)
 (problem-space P2 ↑name selection)
```

Figure 2-4: Working memory representation of the structure in Figure 2-3.

horizontal sequence of objects above the tiles). Each binding points to a tile and a cell; each tile points to its value; and each cell points to its adjacent cells. Eight puzzle operators manipulate only the bindings, the representation of the cells and tiles does not change.

Working memory can be modified by: (1) productions, (2) the decision procedure, and (3) the working-memory manager. Each of these components has a specific function. Productions only add augmentations and preferences to working memory. The decision procedure only modifies the context stack. The working-memory manager only removes irrelevant contexts and objects from working memory.

2.3. The Processing Structure

The processing structure implements the functions required for search in a problem space — bringing to bear task-implementation knowledge to generate objects, and bringing to bear search-control knowledge to select between alternative objects. The search-control functions are all realized by a single generic control act: the *replacement* of an object in a slot by another object from the working memory. The representation of a problem is changed by replacing the current problem space with a new problem space. Returning to a prior state is accomplished by replacing the current state with a preexisting one in working memory. An operator is selected by replacing the current operator (often undecided) with the new one. A step in the problem space occurs when the current operator is applied to the current state to produce a new state, which is then selected to replace the current state in the context.

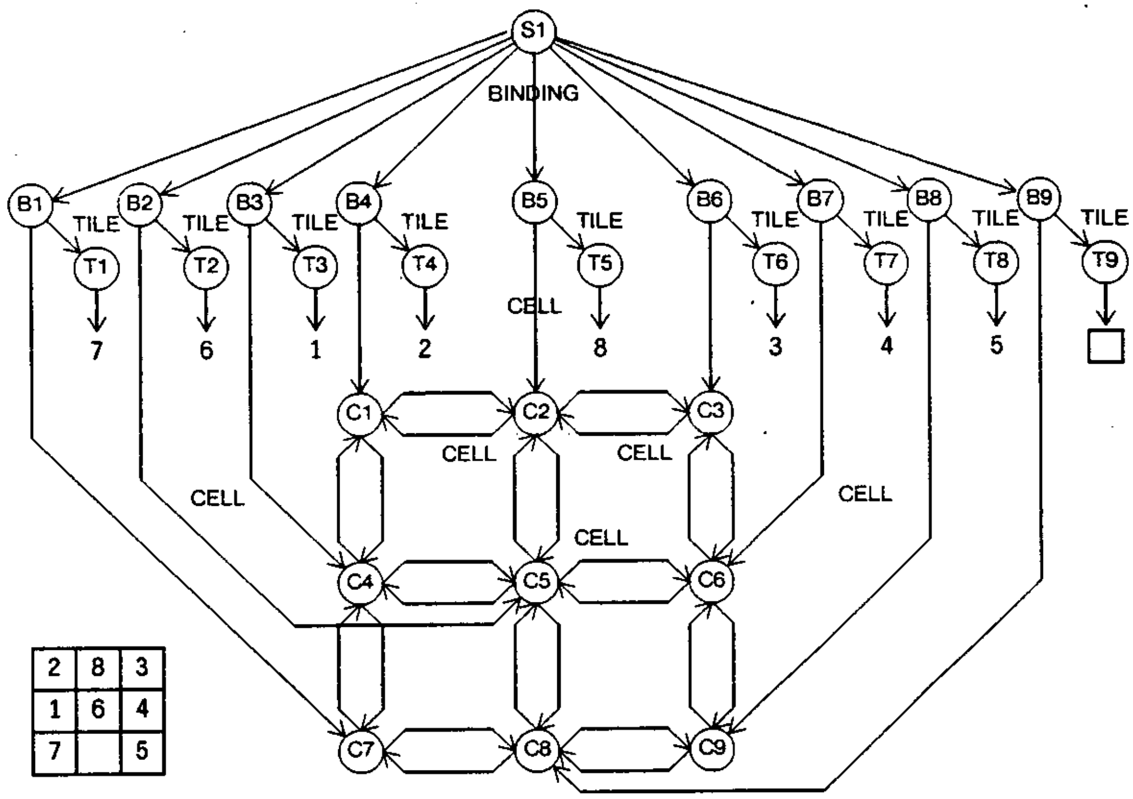


Figure 2-5: Graphic representation of an eight puzzle state.

A replacement can take place anywhere in the context stack, e.g., a new state can replace the state in any of the contexts in the stack, not just the lowest or most immediate context but any higher one as well. When an object in a slot is replaced, all of the slots below it in the context are reinitialized to undecided. Each lower slot depends on the values of the higher slots for its validity: a problem space is set up in response to a goal; a state functions only as part of a problem space; and an operator is to be applied at a state. Each context below the one where the replacement took place is terminated because it depends on the contents of the changed context for its existence (recall that lower contexts contain subgoals of higher contexts).

The replacement of context objects is driven by the *decision cycle*. Figure 2-6 shows three cycles, with the first one expanded out to reveal some of the inner structure. Each cycle involves two distinct parts. First, during the *elaboration phase*, new objects, new augmentations of old objects, and preferences are added to working memory. Then the *decision procedure* examines the accumulated preferences and the context stack, and either it replaces an existing object in some slot, i.e., in one of the roles of a context in the context stack, or it creates a subgoal in response to an impasse.

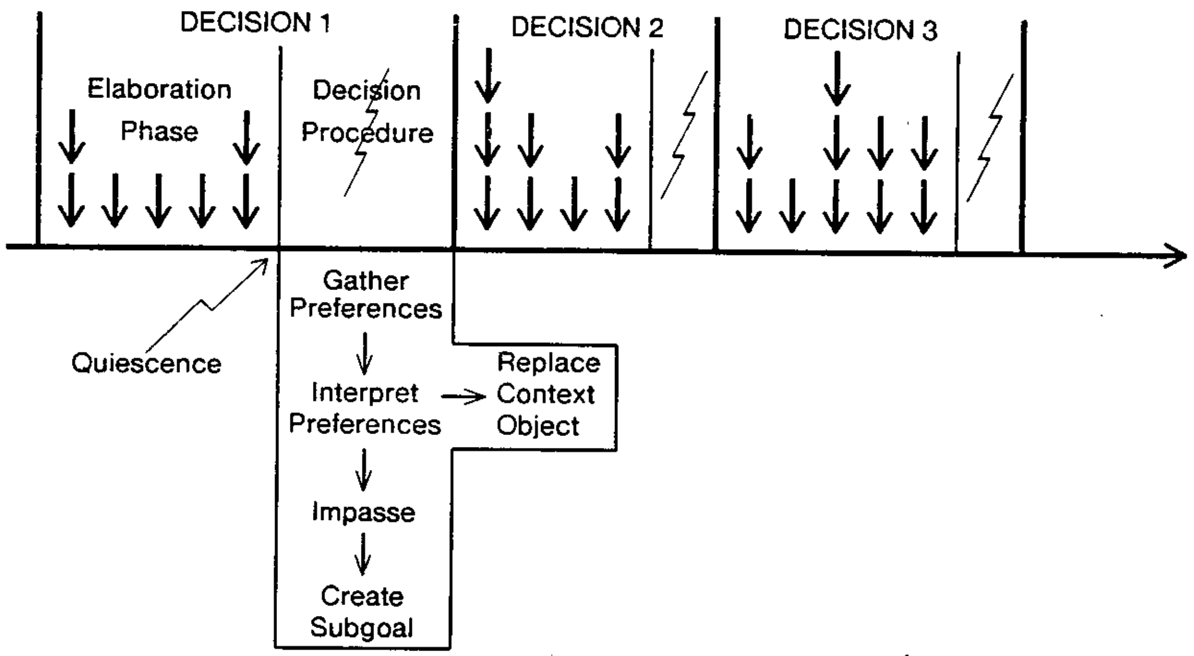


Figure 2-6: A sequence of decision cycles.

2.3.1. The elaboration phase

Based on the current contents of working memory, the elaboration phase adds new objects, augmentations of existing objects, and preferences. Elaborations are generated in parallel (shown by the vertical columns of arrows in Figure 2-6) but may still require multiple steps for completion (shown by the horizontal sequences of elaborations in the figure) because information generated during one step may allow other elaborations to be made on subsequent steps. This is a monotonic process (working-memory elements are not deleted or modified) that continues until quiescence is reached because there are no more elaborations to be generated.[4] The monotonic nature of the elaboration phase assures that no synchronization problems will occur during the parallel generation of elaborations. However, because this is only syntactic monotonicity — data structures are not modified or deleted — it leaves open whether semantic conflicts or non-monotonicity will occur.

The elaboration phase is encoded in Soar as *productions* of the form:

If C_1 *and* C_2 *and ... and* C_m *then add* $A_1, A_2 \ldots A_n$

The C_i are *conditions* that examine the context stack and the rest of the working memory, while the A_i are *actions* that add augmentations or preferences to memory. Condition patterns are based on constants, variables, negations, pattern-ands, and disjunctions of constants (according to the conventions of Ops5 productions). Any object in working memory can be accessed as long as there exists a chain of augmentations

[4] In practice, the elaboration phase reaches quiescence quickly (less than ten cycles), however, if quiescence is not reached after a prespecified number of iterations (typically 100), the elaboration phase terminates and the decision procedure is entered.

and preferences from the context stack to the object. An augmentation can be a link in the chain if its identifier appears either in a context or in a previously linked augmentation or preference. A preference can be a link in the chain if all the identifiers in its context fields (defined in Section 2.3.2) appear in the chain. This property of *linked access* plays an important role in working-memory management, subgoal termination, and chunking, by allowing the architecture to determine which augmentations and preferences are accessible from a context, independent of the specific knowledge encoded in elaborations.

A production is successfully *instantiated* if the conjunction of its conditions is satisfied with a consistent binding of variables. There can be any number of concurrently successful instantiations of a production. All successful instantiations of all productions fire concurrently (simulated) during the elaboration phase. The only conflict-resolution principle in Soar is refractory inhibition — an instantiation of a production is fired only once. Rather than having control exerted at the level of productions by conflict resolution, control is exerted at the level of problem solving (by the decision procedure).

2.3.2. The decision procedure

The decision procedure is executed when the elaboration phase reaches quiescence. It determines which slot in the context stack should have its content replaced, and by which object. This is accomplished by processing the context stack from the oldest context to the newest (ie., from the highest goal to the lowest one). Within each context, the roles are considered in turn, starting with the problem space and continuing through the state and operator in order. The process terminates when a slot is found for which action is required. Making a change to a higher slot results in the lower slots being reinitialized to undecided, thus making the processing of lower slots irrelevant.

This ordering on the set of slots in the context stack defines a fixed desirability ordering between changes for different slots: it is always more desirable to make a change higher up. The processing for each slot is driven by the knowledge symbolized in the preferences in working memory at the end of the elaboration phase. Each preference is a statement about the selection of an object for a slot (or set of slots). Three primitive concepts are available to make preference statements:[5]

acceptability: A choice is to be considered.

rejection: A choice is not to be made.

desirability: A choice is better than (worse than, indifferent to) a reference choice.

[5]There is an additional preference type that allows the statement that two choices for an operator slot can be explored in parallel. This is a special option to explore parallel processing where multiple slots are created for parallel operators. For more details, see the Soar manual [30].

Together, the acceptability and rejection preferences determine the objects from which a selection will be made, and the desirability preferences partially order these objects. The result of processing the slot, if successful, is a single object that is: new (not currently selected for that slot); acceptable; not rejected; and more desirable than any other choice that is likewise new, acceptable and not rejected.

A preference encodes a statement about the selection of an object for a slot into a set of attributes and values, as shown in Figure 2-7. The object is specified by the value of the object attribute. The slot is specified by the combination of a role and a context. The role is either the problem space, the state or the operator; a goal cannot be specified as a role in a preference because goals are determined by the architecture and not by deliberate decisions. The context is specified by the contents of its four roles: goal, problem space, state and operator. A class of contexts can be specified by leaving unspecified the contents of one or more of the roles. For example, if only the problem space and state roles are specified, the preference will be relevant for all goals with the given problem space and state.

The desirability of the object for the slot is specified by the *value* attribute of a preference, which takes one of seven alternatives. Acceptable and reject cover their corresponding concepts; the others — best, better, indifferent, worse, and worst — cover the ordering by desirability. All assertions about ordering locate the given object relative to a *reference* object for the same slot. Since the reference object always concerns the same slot, it is only necessary to specify the object. For better, worse, and some indifferent preferences, the reference object is another object that is being considered for the slot, and it is given by the *reference* attribute of the preference. For best, worst, and the remaining indifferent preferences, the reference object is an abstract anchor point, hence is implicit and need not be given. Consider an example where there are two eight-puzzle operators, named up and left, being considered for state S1 in goal G1. If the identifier for the eight-puzzle problem space is P1, and the identifiers for up and left are O1 and O2, then the following preference says that up is better than left:

```
(preference ↑object O1 ↑role operator ↑value better ↑reference O2
   ↑goal G1 ↑problem-space P1 ↑state S1)
```

The decision procedure computes the best choice for a slot based on the preferences in working memory and the semantics of the preference concepts, as given in Figure 2-8. The preference scheme of Figure 2-8 is a modification of the straightforward application of the concepts of acceptability, rejection and desirability. The modifications arise from two sources. The first is *independence.* The elaboration phase consists of the contributions of independently firing individual productions, each expressing an independent source of knowledge. There is no joint constraint on what each asserts. These separate expressions must be combined, and the only way to do so is to conjoin them. Independence implies that one choice can be (and often is) both acceptable and rejected. For a decision to be possible with such preferences, rejection can not be

```
Attribute

Object   The object that is to occupy the slot

Role     The role the object is to occupy                       |
         (problem space, state, or operator)                    |
                                                                |
Goal           |                                                |
               |                                                |   Slot
Problem space  |                                                |
               | Context in which the preference applies        |
State          | (A set of contexts can be specified)           |
               |                                                |
Operator       |                                                |

Value    acceptable  The object is a candidate for the given role

         reject      The object is not to be selected

         best        The object is as good as any object can be

         better      The object is better than the reference object

         indifferent The object is indifferent to the reference object
                     if there is one, otherwise the object is indifferent
                     to all other indifferent objects

         worse       The object is worse than the reference object
                     (the inverse of better)

         worst       The object is as bad as any object can be
                     (the inverse of best)

Reference  The reference object for order comparison
```

Figure 2-7: The encoding of preferences.

¬acceptable, which would lead to a logical contradiction. Instead, rejection overrides acceptable by eliminating the choice from consideration. Independence also implies that one choice can be both better and worse than another. This requires admitting *conflicts* of desirability between choices. Thus, the desirability order is quite weak, being transitive, but not irreflexive or antisymmetric, and *dominates* must be distinguished from simply *better* — namely, domination implies better without conflict. The possibility of conflicts modifies the notion of the *maximal* subset of a set to be those elements that no other element dominates. For example, in the set of {x, y} if (x > y) and (y > x) then the maximal subset contains both x and y.

The second source of modifications to the decision procedure is *incompleteness*. The elaboration phase will deliver some collection of preferences. These can be silent on any particular fact, e.g., they may assert that x is better than y, and that y is rejected, but say nothing about whether x is acceptable or not, or rejected or not. Indeed, an unmentioned object could be better than any that are mentioned. No constraint on completeness

Primitive predicates and functions on objects, x, y, z, ...

current	The object that currently occupies the slot
acceptable(x)	x is acceptable
reject(x)	x is rejected
$(x > y)$	x is better than y
$(x < y)$	x is worse than y (same as $y > x$)
$(x \sim y)$	x is indifferent to y
$(x >> y)$	x dominates y = $(x > y)$ and $\neg(y > x)$

Reference anchors

$\text{indifferent}(x) \Rightarrow \forall y\, [\text{indifferent}(y) \Rightarrow (x \sim y)]$

$\text{best}(x) \Rightarrow \forall y\, [\text{best}(y) \Rightarrow (x \sim y)] \wedge [\neg\text{best}(y) \wedge \neg(y > x) \Rightarrow (x > y)]$

$\text{worst}(x) \Rightarrow \forall y\, [\text{worst}(y) \Rightarrow (x \sim y)] \wedge [\neg\text{worst}(y) \wedge \neg(y < x) \Rightarrow (x < y)]$

Basic properties

Desirability $(x > y)$ is transitive, but <u>not</u> complete or antisymmetric

Indifference is an equivalence relationship and substitutes over >

$(x > y)$ and $(y \sim z)$ implies $(x > z)$

Indifference does <u>not</u> substitute in acceptable, reject, best, and worst.

acceptable(x) and $(x \sim y)$ does <u>not</u> imply acceptable(y),

reject(x) and $(x \sim y)$ does <u>not</u> imply reject(y), etc.

Default assumption

All preference statements that are not explicitly mentioned and not implied by transitivity or substitution are not assumed to be true

Intermediate definitions

$\text{considered-choices} = \{x \in \text{objects} \mid \text{acceptable}(x) \wedge \neg\text{reject}(x)\}$

$\text{maximal}(X) = \{x \in X \mid \forall y\, \neg(y >> x)\}$

$\text{maximal-choices} = \text{maximal}(\text{considered-choices})$

$\text{empty}(X) = \neg\exists x \in X$

$\text{mutually-indifferent}(X) \Leftrightarrow \forall x,y \in X\, (x \sim y)$

random(X) = choose one element of X randomly

select(X) = if current$\in$X then current else random(X)

Final choice

$\text{empty}(\text{maximal-choices}) \wedge \neg\text{reject}(\text{current}) \Rightarrow \text{final-choice}(\text{current})$

$\text{mutually-indifferent}(\text{maximal-choices}) \wedge \neg\text{empty}(\text{maximal-choices})$
$\Rightarrow \text{final-choice}(\text{select}(\text{maximal-choices}))$

Impasse

$\text{empty}(\text{maximal-choices}) \wedge \text{reject}(\text{current}) \Rightarrow \text{impasse}$

$\neg\text{mutually-indifferent}(\text{maximal-choices}) \Rightarrow \text{impasse}(\text{maximal-choices})$

Figure 2-8: The semantics of preferences.

can hold, since Soar can be in any state of incomplete knowledge. Thus, for the decision procedure to get a result, assumptions must be made to close the world logically. The assumptions all flow from the principle that positive knowledge is required to state a preference — to state that an object is acceptable, rejected or has some desirability relation. Hence, no such assertion should be made by default. Thus, objects are not acceptable unless explicitly acceptable; are not rejected unless explicitly rejected; and are not ordered in a specific way unless explicitly ordered. To do otherwise without explicit support is to rob the explicit statements of assertional power.

Note, however, that this assumption does allow for the existence of preferences implied by the explicit preferences and their semantics. For example, two objects are indifferent if either there is a binary indifferent-preference containing them, there is a transitive set of binary indifferent-preferences containing both of them, they are both in unary indifferent-preferences, they are both in best-preferences, or they are both in worst-preferences.

The first step in processing the preferences for a slot is to determine the set of choices to be considered. These are objects that are acceptable (there are acceptable-preferences for them) and are not rejected (there are no reject-preferences for them). Dominance is then determined by the best, better, worst, and worse preferences. An object dominates another if it is better than the other (or the other is worse) and the latter object is not better than the former object. A best choice dominates all other non-best choices, except those that are explicitly better than it through a better-preference or worst-preference. A worst choice is dominated by all other non-worst choices, except those that are explicitly worse than it through a better or worst preference. The maximal-choices are those that are not dominated by any other objects.

Once the set of maximal-choices is computed, the decision procedure determines the final choice for the slot. The current choice acts as a default so that a given slot will change only if the current choice is displaced by another choice. Whenever there are no maximal-choices for a slot, the current choice is maintained, unless the current choice is rejected. If the set of maximal-choices are mutually indifferent — that is, all pairs of elements in the set are mutually indifferent — then the final choice is one of the elements of the set. The default is to not change the current choice, so if the current choice is an element of the set, then it is chosen; otherwise, one element is chosen at random.[6] The random selection is justified because there is positive knowledge, in the form of preferences, that explicitly states that it does not matter which of the mutually indifferent objects is selected.

If the decision procedure determines that the value of the slot should be changed — that is, there is a final

[6]In place of a random selection, there is an option in Soar to allow the user to select from the set of indifferent choices.

choice different from the current object in the slot — the change is installed, all of the lower slots are reinitialized to undecided, and the elaboration phase of the next decision cycle ensues. If the current choice is maintained, then the decision procedure considers the next slot lower in the hierarchy. If either there is no final choice, or all of the slots have been exhausted, then the decision procedure fails and an *impasse*[7] occurs. In Soar, four impasse situations are distinguished:

1. *Tie:* This impasse arises when there are multiple maximal-choices that are not mutually indifferent and do not conflict. These are competitors for the same slot for which insufficient knowledge (expressed as preferences) exists to discriminate among them.

2. *Conflict:* This impasse arises when there are conflicting choices in the set of maximal choices.

3. *No-change:* This impasse arises when the current value of every slot is maintained.

4. *Rejection:* This impasse arises when the current choice is rejected and there are no maximal choices; that is, there are no viable choices for the slot. This situation typically occurs when all of the alternatives have been tried and found wanting.

The rules at the bottom of Figure 2-8 cover all but the third of these, which involves cross-slot considerations not currently dealt with by the preference semantics. These four conditions are mutually exclusive, so at most one impasse will arise from executing the decision procedure. The response to an impasse in Soar is to set up a subgoal in which the impasse can be resolved.

2.3.3. Implementing the eight puzzle

Making use of the processing structure so far described — and postponing the discussion of impasses and subgoals until Section 2.4 — it is possible to describe the implementation of the eight puzzle in Soar. This implementation consists of both task-implementation knowledge and search-control knowledge. Such knowledge is eventually to be acquired by Soar from the external world in some representation and converted to internal forms, but until such an acquisition mechanism is developed, knowledge is simply posited of Soar, encoded into problem spaces and search control, and incorporated directly into the production memory.

Figures 2-9, 2-10, and 2-11 list the productions that encode the knowledge to implement the eight puzzle task.[8] Figure 2-9 contains the productions that set up things up so that problem solving can begin, and detect when the goal has been achieved. For this example we assume that initially the current goal is to be augmented with the name solve-eight-puzzle, a description of the initial state, and a description of the desired state. The problem space is selected based on the description of the goal. In this case, production select-eight-puzzle-problem-space is sensitive to the name of the goal and suggests eight-puzzle as the

[7] The term was first used in this sense in Repair theory [8]; we had originally used the term difficulty [29].

[8] These descriptions of the productions are an abstraction of the actual Soar productions, which are given in the Soar manual [30].

problem space. The initial state is determined by the current goal and the problem space. Production define-initial-state translates the description of the initial state in the goal to be a state in the eight-puzzle problem space. Similarly. define-final-state translates the description of the desired state to be a state in the eight-puzzle problem space. By providing different initial or desired states, different eight puzzle problems can be attempted. Production detect-eight-puzzle-success compares the current state, tile by tile and cell by cell to the desired state. If they match, the goal has been achieved.

select-eight-puzzle-space:
If the current goal is solve-eight-puzzle, then make an acceptable-preference for eight-puzzle as the current problem space.

define-initial-state:
If the current problem space is eight-puzzle, then create a state in this problem space based on the description in the goal and make an acceptable-preference for this state.

define-final-state:
If the current problem space is eight-puzzle, then augment the goal with a desired state in this problem space based on the description in the goal.

detect-eight-puzzle-success:
If the current problem space is eight-puzzle and the current state matches the desired state of the current goal in each cell, then mark the state with success.

Figure 2-9: Productions that set up the eight puzzle.

The final aspect of the task definition is the implementation of the operators. For a given problem, many different realizations of essentially the same problem space may be possible. For the eight puzzle, there could be twenty-four operators, one for each pair of adjacent cells between which a tile could be moved. In such an implementation, all operators could be made acceptable for each state, followed by the rejection of those that cannot apply (because the blank is not in the appropriate place). Alternatively, only those operators that are applicable to a state could be made acceptable. Another implementation could have four operators, one for each direction in which tiles can be moved into the blank cell: up, down, left, and right. Those operators that do not apply to a state could be rejected.

In our implementation of the eight puzzle, there is a single general operator for moving a tile adjacent to the blank cell into the blank cell. For a given state, an instance of this operator is created for each of the adjacent cells. We will refer to these instantiated operators by the direction they move their associated tile: up, down, left and right. To create the operator instantiations requires a single production, shown in Figure 2-10. Each operator is represented in working memory as an object that is augmented with the cell containing the blank and one of the cells adjacent to the blank. When an instantiated operator is created, an acceptable-preference is also created for it in the context containing the eight-puzzle problem space and the state for which the instantiated operator was created. Since operators are created only if they can apply, an additional production that rejects inapplicable operators is not required.

An operator is applied when it is selected by the decision procedure for an operator role — selecting an

instantiate-operator:
If the current problem space is eight-puzzle and the current state has a tile in a cell adjacent to the blank's cell, then create an acceptable-preference for a newly created operator that will move the tile into the blank's cell.

Figure 2-10: Production for creating eight puzzle operator instantiations.

operator produces a context in which productions associated with the operator can execute (they contain a condition that tests that the operator is selected). Whatever happens while a given operator occupies an operator role comprises the attempt to apply that operator. Operator productions are just elaboration productions, used for operator application rather than for search control. They can create a new state by linking it to the current context (as the object of an acceptable-preference), and then augmenting it. To apply an instantiated operator in the eight puzzle requires the two productions shown in Figure 2-11. When the operator is selected for an operator slot, production create-new-state will apply and create a new state with the tile and blank in their swapped cells. The production copy-unchanged-binding copies pointers to the unchanged bindings between tiles and cells.

create-new-state:
If the current problem space is eight-puzzle, then create an acceptable-preference for a newly created state, and augment the new state with bindings that have switched the tiles from the current state that are changed by the current operator.

copy-unchanged-binding:
If the current problem space is eight-puzzle and there is an acceptable-preference for a new state, then copy from the current state each binding that is unchanged by the current operator.

Figure 2-11: Productions for applying eight puzzle operator instantiations.

The seven productions so far described comprise the task-implementation knowledge for the eight puzzle. With no additional productions, Soar will start to solve the problem, though in an unfocused manner. Given enough time it will search until a solution is found.[9] To make the behavior a bit more focused, search-control knowledge can be added that guides the selection of operators. Two simple search-control productions are shown in Figure 2-12. Avoid-undo will avoid operators that move a tile back to its prior cell. Mea-operator-select is a means-ends-analysis heuristic that prefers the selection of an operator if it moves a tile into its desired cell. This is not a fool-proof heuristic rule, and will sometimes lead Soar to make an incorrect move.

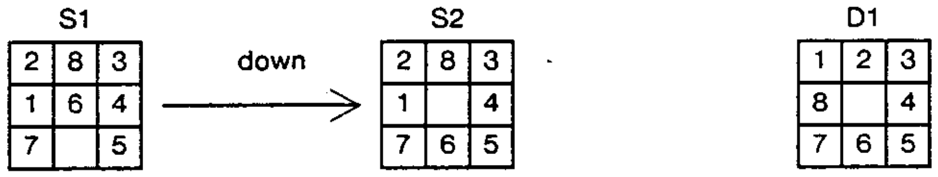
avoid-undo:
If the current problem space is eight-puzzle, then create a worst-preference for the operator that will move the tile that was moved by the operator that created the current state.

mea-operator-selection:
If the current problem space is eight-puzzle and an operator will move a tile into its cell in the desired state, then make a best-preference for that operator.

Figure 2-12: Search-control productions for the eight puzzle.

[9] The default search is depth-first where the choices between competing operators are made randomly. Infinite loops do not arise because the choices are made non-deterministically.

Figure 2-13 contains a trace of the initial behavior using these nine productions (the top of the figure shows the states and operator involved in this trace). The trace is divided up into the activity occurring during each of the first five decision cycles (plus an initialization cycle). Within each cycle, the activity is marked according to whether it took place within the elaboration phase (E), or as a result of the decision procedure procedure (D). The steps within the elaboration phase are also marked; for example, line 4.1E represents activity occurring during the first step of the elaboration phase of the fourth decision cycle. Each line that is part of the elaboration phase represents a single production firing. Included in these lines are the production's name and a description of what it does. When more than one production is marked the same, as in 4.2E, it means that they fire in parallel during the single elaboration step.



Cycle	Production	Action
D	G1 is the current goal	G1 is already augmented with solve-eight-puzzle
1E	select-eight-puzzle-space	Make an acceptable-preference for eight-puzzle
1D	Select eight-puzzle as problem space	
2E	define-final-state	Augment goal with the desired state (D1)
2E	define-initial-state	Make an acceptable-preference for S1
2D	Select S1 as state	
3.1E	instantiate-operator	Create O1 (down) and an acceptable-preference for it
3.1E	instantiate-operator	Create O2 (right) and an acceptable-preference for it
3.1E	instantiate-operator	Create O3 (left) and an acceptable-preference for it
3.2E	mea-operator-selection (O1-down)	Make a best-preference for down
3D	Select O1 (down) as operator	
4.1E	create-new-state	Make an acceptable-preference for S2, swap bindings
4.2E	copy-unchanged-binding	Copy over unmodified bindings
4.2E	copy-unchanged-binding	
4.2E	copy-unchanged-binding	
4.2E	copy-unchanged-binding	
4.2E	copy-unchanged-binding	
4.2E	copy-unchanged-binding	
4.2E	copy-unchanged-binding	
4D	Select S2 as state	
5E	instantiate-operator	Create O4 (down) and an acceptable-preference for it
5E	instantiate-operator	Create O5 (right) and an acceptable-preference for it
5E	instantiate-operator	Create O6 (left) and an acceptable-preference for it
5E	instantiate-operator	Create O7 (up) and an acceptable-preference for it
5E	Avoid-undo (O7-up)	Make a worst-preference for up
5D	Tie impasse, create subgoal	

Figure 2-13: Trace of initial eight puzzle problem solving.

The trace starts where the current goal (called G1) is the only object defined. In the first cycle, the goal is augmented with an acceptable-preference for eight-puzzle for the problem-space role. The decision procedure then selects eight-puzzle as the current problem space. In cycle 2, the initial state, S1, is created with an acceptable-preference for the state role, and the problem space is augmented with its operators. At the end of cycle 2, the decision procedure selects S1 as the current state. In cycle 3, operator instances, with corresponding acceptable-preferences, are created for all of the tiles that can move into the blank cell. Production mea-operator-selection makes operator O1 (down) a best choice, resulting in its being selected as the current operator. In cycle 4, the operator is applied. First, production create-new-state creates the preference for a new state (S2) and augments it with the swapped bindings, and then production copy-unchanged fills in the rest of the structure of the new state. Next, state S2 is selected as the current state and operator instances are created — with corresponding acceptable-preferences — for all of the tiles that can move into the cell that now contains the blank. On the next decision cycle (cycle 5), none of the operators dominate the others, and an impasse occurs.

2.4. Impasses and Subgoals

When attempting to make progress in attaining a goal, the knowledge directly available in the problem space (encoded in Soar as productions) may be inadequate to lead to a successful choice by the decision procedure. Such a situation occurred in the last decision cycle of the eight puzzle example in Figure 2-13. The knowledge directly available about the eight puzzle was incomplete — it did not specify which of the operators under consideration should be selected. In general, impasses occur because of incomplete or inconsistent information. Incomplete information may yield a rejection, tie, or no-change impasse, while inconsistent information yields a conflict impasse.

When an impasse occurs, returning to the elaboration phase cannot deliver additional knowledge that might remove the impasse, for elaboration has already run to quiescence. Instead, a subgoal and a new context is created for each impasse. By responding to an impasse with the creation of a subgoal, Soar is able to deliberately search for more information that can lead to the resolution of the impasse. All types of knowledge, task-implementation and search-control, can be encoded in the problem space for a subgoal.

If a tie impasse between objects for the same slot arises, the problem solving to select the best object will usually result in the creation of one or more desirability preferences, making the subgoal a locus of search-control knowledge for selecting among those objects. A tie impasse between two objects can be resolved in a number of ways: one object is found to lead to the goal, so a best preference is created; one object is found to be better than the other, so a better preference is created; no difference is found between the objects, so an indifferent preference is created; or one object is found to lead away from the goal, so a worst preference is created. A number of different problem solving strategies can be used to generate these outcomes, including: further elaboration of the tied objects (or the other objects in the context) so that a detailed comparison can be made; look-ahead searches to determine the effects of choosing the competing objects; and analogical mappings to other situations where the choice is clear.

If a no-change impasse arises with the operator slot filled, the problem solving in the resulting subgoal will usually involve operator implementation, terminating when an acceptable-preference is generated for a new state in the parent problem space. Similarly, subgoals can create problem spaces or initial states when the required knowledge is more easily encoded as a goal to be achieved through problem-space search, rather than as a set of elaboration productions.

When the impasse occurs during the fifth decision cycle of the eight-puzzle example in Figure 2-13, the following goal and context are added to working memory.

```
    (goal G2 ↑supergoal G1 ↑impasse tie ↑choices multiple ↑role operator
            ↑item O4 O5 O6
            ↑problem-space undecided ↑state undecided ↑operator undecided)
```

The subgoal is simply a new symbol augmented with: the supergoal (which links the new goal and context into the context stack); the type of impasse; whether the impasse arose because there were no choices or multiple choices in the maximal-choices set; the role where the impasse arose; the objects involved in conflicts or ties (the items); and the problem-space, state, and operator slots (initialized to undecided). This information provides an initial definition of the subgoal by defining the conditions that caused it to be generated and the new context. In the following elaboration phase, the subgoal can be elaborated with a suggested problem space, an initial state, a desired state or even path constraints. If the situation is not sufficiently elaborated so that a problem space and initial state can be selected, another impasse ensues and a further subgoal is created to handle it.

Impasses are resolved by the addition of preferences that change the results of the decision procedure. When an impasse is resolved, allowing problem solving to proceed in the context, the subgoal created for the impasse has completed its task and can be terminated. For example, if there is a subgoal for a tie impasse at the operator role, it will be terminated when a new preference is added to working memory that either rejects all but one of the competing operators, makes one a best choice, makes one better than all the others, etc. The subgoal will also be terminated if new preferences change the state or problem-space roles in the context, because the contents of the operator role depends on the values of these higher roles. If there is a subgoal created for a no-change impasse at the operator role — usually because of an inability to implement the operator directly by rules in the problem space — it can be resolved by establishing a preference for a new state, most likely the one generated from the application of the operator to the current state.

In general, any change to the context at the affected role or above will lead to the termination of the subgoal. Likewise, a change in any of the contexts above a subgoal will lead to the termination of the subgoal because its depends on the higher contexts for its existence. Resolution of an impasse terminates all goals below it.

When a subgoal is terminated, many working-memory elements are no longer of any use since they were created solely for internal processing in the subgoal. The *working-memory manager* removes these useless working-memory elements from terminated subgoals in essentially the same way that a garbage collector in Lisp removes inaccessible CONS cells. Only the results of the subgoal are retained — those objects and preferences in working memory that meet the criteria of linked access to the unterminated contexts, as defined in Section 2.3.1. The context of the subgoal is itself inaccessible from supergoals — its supergoal link is one-way — so it is removed.

The architecture defines the concept of goal termination, not the concept of goal success or failure. There are many reasons why a goal should disappear and many ways in which this can be reflected in the preferences. For instance, the ordinary (successful) way for a subgoal implementing an operator to terminate is to create the new result state and preferences that enable it to be selected (hence leading to the operator role becoming undecided). But sometimes it is appropriate to terminate the subgoal (with failure) by rejecting the operator or selecting a prior state, so that the operator is never successfully applied.

Automatic subgoal termination at any level of the hierarchy is a highly desirable, but generally expensive, feature of goal systems. In Soar, the implementation of this feature is not expensive. Because the architecture creates all goals, it has both the knowledge and the organization necessary to terminate them. The decision procedure iterates through all contexts from the top, and within each context, through the different roles: problem space, state and operator. Almost always, no new preferences are available to challenge the current choices. If new preferences do exist, then the standard analysis of the preferences ensues, possibly determining a new choice. If everything remains the same, the procedure continues with the next lower slot; if the value of a slot changes then all lower goals are terminated. The housekeeping costs of termination, which is the removal of irrelevant objects from the working memory, is independent of how subgoal termination occurs.

2.5. Default Knowledge for Subgoals

An architecture provides a frame within which goal-oriented action takes place. What action occurs depends on the knowledge that the system has. Soar has a basic complement of task-independent knowledge about its own operation and about the attainment of goals within it that may be taken as an adjunct to the architecture. A total of fifty-two productions embody this knowledge. With it, Soar exhibits reasonable default behavior; without it (or other task knowledge), Soar can flounder and fall into an infinitely deep series of impasses. We describe here the default knowledge and how it is represented. All of this knowledge can be over-ridden by additional knowledge that adds other preferences.

Common search-control knowledge. During the problem solving in a problem space, search-control rules are available for three common situations that require the creation of preferences.

1. **Backup from a failed state.** If there is a reject-preference for the current state, an acceptable-preference is created for the previous state. This implements an elementary form of backtracking.

2. **Make all operators acceptable.** If there are a fixed set of operators that can apply in a problem space, they should be candidates for every state. This is accomplished by creating acceptable-preferences for those operators that are directly linked to the problem space.

3. **No operator retry.** Given the deterministic nature of Soar, an operator will create the same result whenever it is applied to the same state. Therefore, once an operator has created a result for a state in some context, a preference is created to reject that operator whenever that state is the current state for a context with the same problem space and goal.

Diagnose impasses. When an impasse occurs, the architecture creates a new goal and context that provide some specific information about the nature of the impasse. From there, the situation must be diagnosed by search-control knowledge to initiate the appropriate problem-solving behavior. In general this will be task-dependent, conditional on the knowledge embedded in the entire stack of active contexts. For situations in which such task-dependent knowledge does not exist, default knowledge exists to determine what to do.

1. **Tie impasse.** Assume that additional knowledge or reasoning is required to discriminate the items that caused the tie. The selection problem space (described below) is made acceptable to work on this problem. A worst-preference is also generated for the problem space, so that any other proposed problem space will be preferred.

2. **Conflict impasse.** Assume that additional knowledge or reasoning is required to resolve the conflict and reject some of the items that caused the conflict. The selection problem space is also the appropriate space and it is made acceptable (and worst) for the problem space role.[10]

3. **No-change impasse.**

   a. **For goal, problem space and state roles.** Assume that the next higher object in the context is responsible for the impasse, and that a new path can be attempted if the higher object is rejected. Thus, the default action is to create a reject-preference for the next higher object in the context or supercontext. The default action is taken only if a problem space is not selected for the subgoal that was generated because of the impasse. This allows the default action to be overriden through problem solving in a problem space selected for the no-change impasse. If there is a no-change impasse for the top goal, problem solving is halted because there is no higher object to reject and no further progress is possible.

[10] There has been little experience with conflict subgoals so far. Thus, little confidence can be placed in the treatment of conflicts and they will not be discussed further.

b. **For operator role.** Such an impasse can occur for multiple reasons. The operator could be too complex to be performed directly by productions, thus needing a subspace to implement it, or it could be incompletely specified, thus needing to be instantiated. Both of these require task-specific problem spaces and no appropriate default action based on them is available. A third possibility is that the operator is inapplicable to the given state, but that it would apply to some other state. This does admit a domain-independent response, namely attempting to find a state in the same problem space to which the operator will apply (*operator subgoaling*). This is taken as the appropriate default response.

4. **Rejection impasse.** The assumption is the same as for (nonoperator) no-change subgoals: the higher object is responsible and progress can be made by rejecting it. If there is a rejection impasse for the top problem space, problem solving is halted because there is no higher object.

The selection problem space. This space is used to resolve ties and conflicts. The states of the selection space contain the candidate objects from the supercontext (the *items* associated with the subgoal). Figure 2-14 shows the subgoal structure that arises in the eight puzzle when there is no direct search-control knowledge to select between operators (such as the mea-operator-selection production). Initially, the problem solver is at the upper-left state and must select an operator. If search control is unable to uniquely determine the next operator to apply, a tie impasse arises and a subgoal is created to do the selection. In that subgoal, the selection problem space is used.

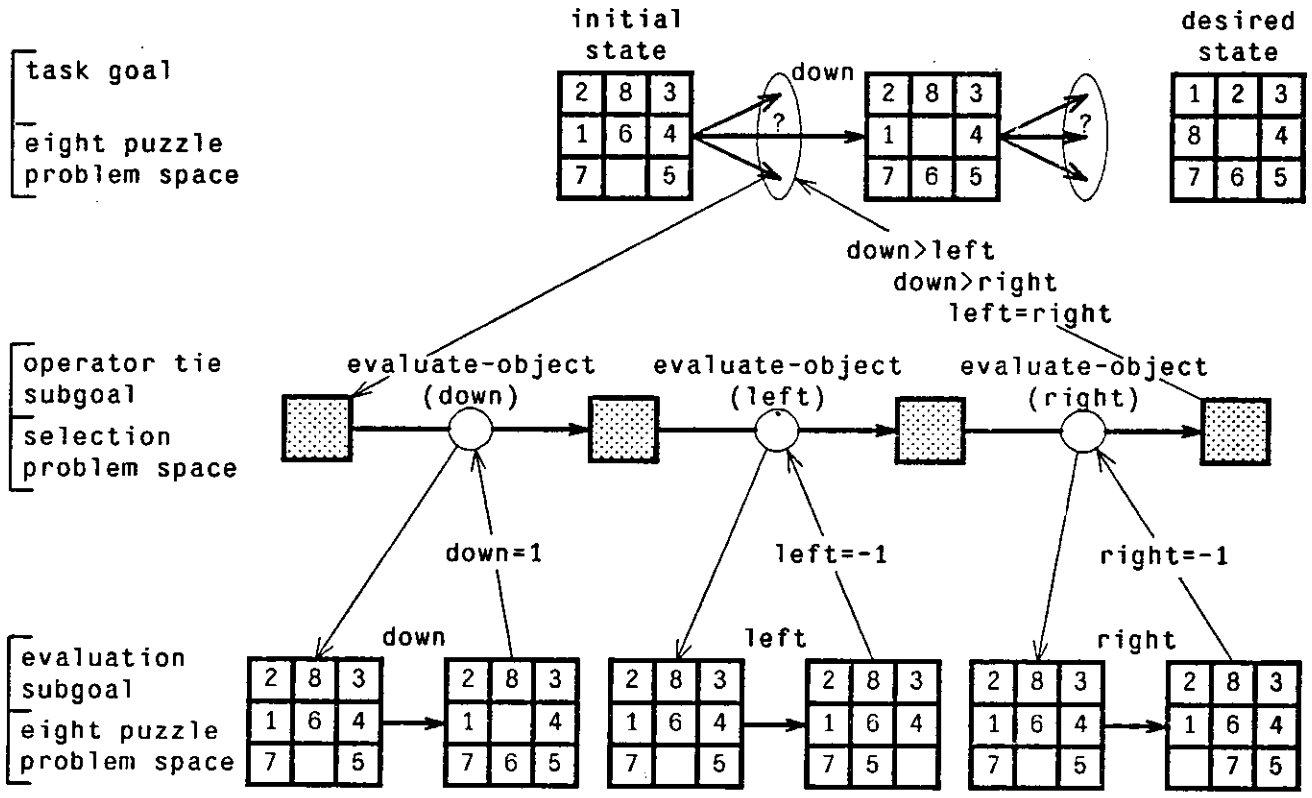


Figure 2-14: The subgoal structure for the eight puzzle.

The one operator in the selection space, evaluate-object, is a general operator that is instantiated with each tying (or conflicting) object; that is, a unique evaluate-object operator is created for each instantiation. Each state in the selection space is a set of evaluations produced by evaluate-object operators (the contents of these states is not shown in the figure). In the figure, an evaluate-object operator is created for each of the tied operators: down, left, and right. Each evaluate-object operator produces an evaluation that allows the creation of preferences involving the objects being evaluated. This requires task-specific knowledge, so either productions must exist that evaluate the contending objects, or a subgoal will be created to perform this evaluation (see below for a default strategy for such an evaluation). Indifferent-preferences are created for all of the evaluate-object operators so that a selection between them can be made without an infinite regression of tie impasses. If all of the evaluate-object operators are rejected, but still no selection can be made, problem solving in the selection problem space will have failed to achieve the goal of resolving the impasse. When this happens, default knowledge (encoded as productions) exists that makes all of the tied alternatives indifferent (or, correspondingly, rejects all of the conflicting alternatives). This allows problem solving to continue.

The evaluation subgoal. In the selection problem space, each evaluate-object operator must evaluate the item with which it is instantiated. Task-dependent knowledge may be available to do this. If not, a no-change impasse will occur, leading to a subgoal to implement the operator. One task-independent evaluation technique is *lookahead* — try out the item temporarily to gather information. This serves as the default. For this, productions reconstruct the task context (i.e., the supercontext that lead to the tie), making acceptable-preferences for the objects selected in the context and augmenting the new goal with information from the original goal. In Figure 2-14, the original task problem space and state are selected for the evaluation subgoals. Once the task context has been reconstructed, the item being evaluated — the down operator — is selected (it receives a best-preference in the evaluation subgoal). This allows the object to be tried out and possibly an evaluation to be produced based on progress made toward the goal.

When knowledge is available to evaluate the states in the task space, the new state produced in the evaluation subgoal will receive an evaluation, and that value can be backed up to serve as the evaluation for the down operator in this situation. One simple eight-puzzle evaluation is to compute the number of tiles that are changed relative to the locations in the desired state. A value of 1 is assigned if the moved tile is out of position in the original state and in position in the result state. A value of 0 is assigned if the moved tile is out of position in both states. A value of -1 is assigned if the moved tile is in position in the original state and out of position in the result state. When an evaluation has been computed for down, the evaluation subgoal terminates, and then the whole process is repeated for the other two operators (left and right). These evaluations can be used to generate preferences among the competing operators. Since down creates a state with a better evaluation than the other operators, better-preferences (signified by > in the figure) are created for down. An indifferent-preference (signified by = in the figure) is also created for left and right because they have equal evaluations. The preferences for down lead to its selection in the original task goal and problem space, terminating the tie subgoal. At this point down is reapplied to the initial state, the result is selected and the process continues.

Figure 2-15 shows, in a state-space representation, two steps of the search that occurs within the eight puzzle problem space. The distinctive pattern of moves in Figure 2-15 is that of steepest-ascent hill climbing, where the state being selected at each step is the best at that level according to the evaluation function. These states were generated in the attempt to solve many different subgoals, rather than from the adoption of a coordinated method of hill climbing in the original task space. Other types of search arise in a similar way. If no knowledge to evaluate states is available except when the goal is achieved, a depth-first search arises. If it is known that every other move is made by an opponent in a two-player game, a mini-max search emerges. The emergence of methods directly from knowledge in Soar is discussed further in Section 3.2.

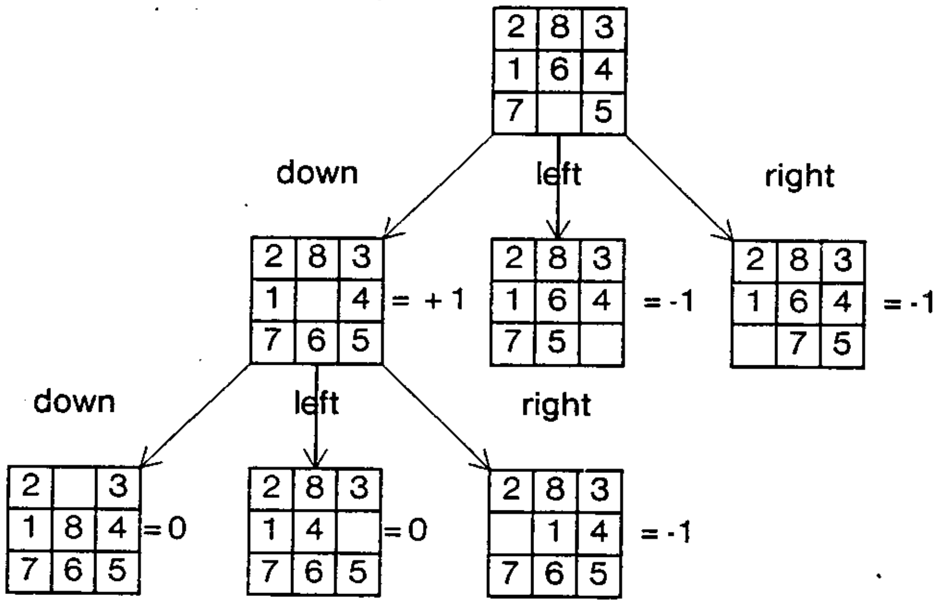


Figure 2-15: A trace of steepest ascent hill climbing.

2.6. Chunking

Chunking is a learning scheme for organizing and remembering ongoing experience automatically on a continuing basis. It has been much studied in psychology [7, 12, 43, 50] and it was developed into an explicit learning mechanism within a production-system architecture in prior work [35, 61, 63]. The current chunking scheme in Soar is directly adapted from this latter work. As defined there, it was a process that acquired chunks that generated the results of a goal, given the goal and its parameters. The parameters of a goal were defined to be those aspects of the system existing prior to the goal's creation that were examined during the processing of the goal. Each chunk was represented as a set of three productions, one that encoded the parameters of a goal, one that connected this encoding in the presence of the goal to (chunked) results, and one that decoded the results. Learning was bottom-up: chunks were built only for terminal goals — goals for which there were no subgoals that had not already been chunked. These chunks improved task performance by substituting efficient productions for complex goal processing. This mechanism was shown to work for a set of simple perceptual-motor skills based on fixed goal hierarchies [61] and it exhibited the power-law speed improvement characteristic of human practice [50]. Currently, Soar does away with one feature of this chunking scheme, the three-production chunks, and allows greater flexibility on a second, the bottom-up nature of chunking. In Soar, single-production chunks are built for either terminal subgoals or for every subgoal, depending on the user's option.

The power of chunking in Soar stems from Soar's ability to generate goals automatically for problems in any aspect of its problem-solving behavior: a goal to select among alternatives leads to the creation of a chunk-production that will later control search; a goal to apply an operator to a state leads to the creation of a chunk-production that directly implements the operator. The occasions of subgoals are exactly the conditions where Soar requires learning, since a subgoal is created if and only if the available knowledge is insufficient for the next step in problem solving. The subgoal is created to find the necessary knowledge and the chunking mechanism stores away the knowledge so that under similar circumstances in the future, the knowledge will be available. Actually, Soar learns what is necessary to avoid the impasse that led to the subgoal, so that henceforth a subgoal will be unnecessary, as opposed to learning to supply results after the subgoal has been created. As search-control knowledge is added through chunking, performance improves via a reduction in the amount of search. If enough knowledge is added, there is no search; what is left is an efficient algorithm for a task. In addition to reducing search within a single problem space, chunks can completely eliminate the search of entire subspaces whose function is to make a search-control decision or perform a task-implementation function (such as applying an operator or determining the initial state of the task).

2.6.1. The chunking mechanism

A chunk production summarizes the processing in a subgoal. The actions generate those working-memory elements that eliminated the impasse responsible for the subgoal (and thus terminated the subgoal). The conditions test those aspects of the current task that were relevant to those actions being performed. The chunk is created when the subgoal terminates — that is when all of the requisite information is available. The chunk's actions are based on the results of the subgoal — those working-memory elements created in the subgoal (or its subgoals) that are accessible from a supergoal. An augmentation is a result if its identifier either existed before the subgoal was created, or is in another result. A preference is a result if all of its specified context objects (goal, problem space, state and operator) either existed before the subgoal was created, or are in another result.

The chunk's conditions are based on a *dependency analysis* of traces of the productions that fired during the subgoal. The traces are accumulated during the processing of the subgoal, and then used for condition determination at subgoal termination time. Each trace contains the working-memory elements that the production matched (*condition elements*) and those it generated (*action elements*).[11] Only productions that actually add something to working memory have their traces saved. Productions that just monitor the state (that is, only do output) do not affect what is learned, nor do productions that attempt to add working-memory elements that already exist (recall that working memory is a set).

Once a trace is created it needs to be stored on a list associated with the goal in which the production fired. However, determining the appropriate goal is problematic in Soar because elaborations can execute in parallel for any of the goals in the stack. The solution comes from examining the contexts tested by the production. The lowest goal in the hierarchy that is matched by conditions of the production is taken to be the one affected by the production firing. The production will affect the chunks created for that goal and possibly, as we shall see shortly, the higher goals. Because the production firing is independent of the lower goals — it would have fired whether they existed or not — it will have no effect on the chunks built for those goals.

When the subgoal terminates, the results of the subgoal are factored into independent subgroups, where two results are considered dependent if they are linked together or they both have links to a third result object. Each subgroup forms the basis for the actions of one production, and the conditions of each production are determined by an independent dependency analysis. The effect of factoring the result is to produce more productions, with fewer conditions and actions in each, and thus more generality than if a single production was created that had all of the actions together. For each set of results, the dependency-analysis procedure starts by finding those traces that have one of the results as an action element. The condition elements of these traces are then divided up into those that existed prior to the creation of the subgoal and those that were created in the subgoal. Those created prior to the subgoal become conditions of the chunk. The others are then recursively analyzed as if they were results, to determine the pre-subgoal elements that were responsible for their creation.

Earlier versions of chunking in Soar [36] implicitly embodied the assumption that problem solving was perfect — if a rule fired in a subgoal, then that rule must be relevant to the generation of the subgoal's results.

[11] If there is a condition that tests for the absence of a working-memory element, a copy of that negated condition is saved in the trace with its variables instantiated from the values bound elsewhere in the production.

The conditions of a chunk were based on the working-memory elements matched by all of the productions that fired in the subgoal. When the assumption was violated, as it was when the processing involved searches down paths that led to failure, overly specific chunks were created. By working backward from the results, the dependency analysis includes only those working-memory elements that were matched by the productions that actually led to the creation of the results. Working-memory elements that are examined by productions, but that turn out to be irrelevant, are not included.

A generalization process allows the chunk to apply in a future situation in which there are objects with the same descriptions, but possibly different identifiers. Once the set of chunk-productions is determined, they are generalized by replacing the identifiers in the working-memory elements with variables. Each identifier serves to tie together the augmentations of an object, and serves as a pointer to the object, but carries no meaning of its own — in fact, a new identifier is generated each time an object is created. Constant symbols — those that are not used as the identifiers of objects — are not modified by this variablization process, only the identifiers. All instances of the same identifier are replaced by the same variable. Different identifiers are replaced by different variables which are forced to match distinct identifiers. This scheme may sometimes be in error, creating productions that will not match when two elements just happen to have the same (or different) identifiers, but it always errs by being too constraining.

The final step in the chunk creation process is to perform a pair of optimizations on the chunk productions. The first optimization simplifies productions learned for the implementation of a complex operator. As part of creating the new state, much of the substructure of the prior state may be copied over to the new state. The chunk for this subgoal will have a separate condition, with an associated action, for each of the substructures copied. The chunk thus ends up with many condition-action pairs that are identical except for the names of the variables. If such a production were used in Soar during a new situation, a huge number of instantiations would be created, one for every permutation of the objects to be copied. The optimization eliminates this problem by removing the conditions that copy substructure from the original production. For each type of substructure being copied, a new production is created which includes a single condition-action pair that will copy substructures of that type. Since all of the actions are additive, no orderering of the actions has to be maintained and the resulting set of rules will copy all of the substructure in parallel.

The second optimization is to order the production conditions in an attempt to make the matcher faster. Each condition acts like a query — returning all of the working-memory elements that match the condition — and the overall match process returns all of the production instantiations that match the conjunctive queries specified by the condition sides of the productions. The efficiency of such a match process is heavily dependent on the order of the queries [74]. By automatically ordering the conditions in Soar, the number of intermediate instantiations of a production is greatly reduced and the overall efficiency improved.[12]

2.6.2. An example of chunk creation

Figure 2-16 shows a trace of the productions that contribute to a chunk built for the evaluation subgoal in the eight-puzzle example discussed in Section 2.5. The first six decision cycles lead up to the subgoal that implements evaluate-object(down) (evaluate the eight-puzzle operator down). G1 is the initial goal, G2 is the subgoal to eliminate a tie between operators, and G3 is the subgoal to implement evaluate-object(down). Included in this trace are the names of those productions fired during subgoal G3 that provide traces used by the dependency analysis. Listed for each of these rule firings are the condition elements that existed prior to the goal, and which therefore become the basis of the chunk's conditions; and the action elements that are linked to preexisting structure, and which therefore become the basis of the actions of the chunk.

```
Cycle
0    G: G1 [Solve the eight puzzle]
1    P: P1 [Eight-Puzzle]
2    S: S1
3       G: G2 (Tie impasse, operators {O1[down] O2[left] O3[right]})
4       P: P2 [Selection]
5       S: SS1
6       O: O4 [evaluate-object[O1[down]]]
7          G: G3 (No-change impasse, operator)
  eval*select-role-operator                             ;wm elements tested to
    (goal G2 ↑operator O4)                              ;establish the context
    (operator O4 ↑name evaluate-object ↑desired D1      ;in which operator O1[down]
                ↑role operator ↑superoperator O1        ;can be evaluated
                ↑superproblem-space P1 ↑superstate S1)
    -->
8          P: P1 [Eight-Puzzle]
9          S: S1
10         O: O1 [down]
  create-new-state
    (problem-space P1 ↑name eight-puzzle)               ;wm elements tested to
    (operator O1 ↑name move-tile ↑adjacent-cell C1)     ;apply operator that moves
    (state S1 ↑binding B1 ↑binding B2)                  ;the tile in C1 into the
    (binding B1 ↑tile T1 ↑cell C2)                      ;cell with the blank (C2)
    (tile T1 ↑name blank)                               ;T1 is the blank
    (binding B2 ↑tile T2 ↑cell C1)                      ;T2 is the tile in cell C1
    -->
11         S: S2
  eval*state-plus-one
    (problem-space P1 ↑name eight-puzzle)               ;wm elements tested to
    (operator O4 ↑name evaluate-object                  ;create evaluation for
                ↑desired D1 ↑evaluation E1)             ;state based on detecting
    (desired D1 ↑binding DB1)                           ;that the operator
    (binding DB1 ↑cell C2 ↑tile T2)                     ;has moved a tile into
    (cell C2 ↑cell C1)                                  ;its desired position
    -->
    (evaluation E1 ↑value 1)                            ;the result/action

12    O: O5 [evaluate-object[O2[left]]
```

Figure 2-16: Partial production trace of an eight-puzzle evaluation subgoal.

[12] The details of the reordering algorithm are not important here, except that the most recent version (Sept. 86), by Dan Scales and John Laird, is almost as effective as ordering by hand.

Once the evaluation subgoal is generated, the production eval*select-role-operator fires and creates acceptable-preferences for the original task problem space (P1), the original task state (S1), and the operator being evaluated (O1). The production also augments goal G3 with the task goal's desired state (D1). Many of the production's conditions match working-memory elements that are a part of the definition of the evaluate-object operator, and thus existed prior to the creation of subgoal G3. These test that the subgoal is to implement the evaluate-object operator, and they access identifiers of super-objects so that the identifiers can be included in the preferences generated by the actions of the production. Following the selection of P1 and S1, a production instantiation fires to generate a best-preference for operator O1 for this specific goal, problem space, and state. This production firing is not shown because it does not add new conditions to the chunk.

The problem solving continues with the selection of O1 and the generation of a new state (S2). The unchanged bindings are copied by a rule that is not shown because it does not affect the subgoal's result. S2 is selected and then evaluated by production eval*state-plus-one, which augments object E1 with the value of the evaluation. This augmentation is a result of the subgoal because object E1 is linked to the state in the parent context. Immediately afterwards, in the same elaboration phase, a production generates a reject-preference for operator O4, the evaluate-object operator. This production has no effect on the chunk built for subgoal G3 because it looks only at higher contexts. Once the reject-preference is created, operator O4 is rejected, another operator is selected, the no-change impasse is eliminated, subgoal G3 is terminated, and a chunk is built.

Only certain of the augmentations of the objects are included in the chunk; namely, those that played an explicit role in attaining the result. For instance, only portions of the state (S1) and the desired state (D1) are included. Even in the substructure of the state, such as binding B2, its tile (T2) has only its identifier saved, and not its value (6), because the actual value was never tested. The critical aspect to be tested in the chunk is that the tile appears as a tile-augmentation of both bindings B2 and DB1 (a binding in the desired state, D1). The exact value of the tile is never tested in the subgoal, so it is not included in the chunk. The conditions created from these working-memory elements will test: that a tile (in this case T2) in the current state (S1) is in a cell adjacent to the cell containing the blank; and that the cell containing the blank is the cell in which the tile appears in the desired state. In other words, the chunk fires whenever the evaluate-object operator is selected in the selection problem space and the operator being evaluated will move a tile into place.

The action of the chunk is to create an evaluation of 1. This value is used to create preferences by comparing it to the values produced by evaluating other operators. The other evaluation values arise when a tile is neither moved into nor out of its desired cell (0), or when a tile is move out of its desired cell (-1). Symbolic values could have been used in place of the numeric ones, as long as there are additional productions to compare the values and create appropriate preferences.

Figure 2-17 contains the one-production chunk built for this example in the format used as input to Soar, which is similar to that used for Ops5 productions. Each production is a list, consisting of a name, the conditions, the symbol "-->", and the actions. Each condition is a template to be matched against working-memory elements. Symbols in a production of the form "<...>" (e.g., <G1>) are variables, all others are constants. The actions are templates for the generation of working-memory elements. In building the chunk, all identifiers from the original working-memory elements have been replaced by variables. The constants in the working-memory elements, those symbols that have no further augmentations (evaluate-object, eight-puzzle, blank), remain as constants in the conditions. Identifier variablization is also responsible for the additional negation predicates in the specification of objects <S1> and <B2>, such as { <> <B1> <B2> } in object <S1>. This is a conjunctive test that succeeds only if <B2> can be bound to a value that is not equal to the value bound to <B1>, thus forcing the objects that are bound to the two variables to be different.

```
(sp p0038
    (goal <G2> ↑operator <O4>)
    (operator <O4> ↑name evaluate-object ↑role operator
        ↑superproblem-space <P1> ↑superstate <S1>
        ↑superoperator <O1> ↑evaluation <E1> ↑desired <D1>)
    (problem-space <P1> ↑name eight-puzzle)
    (operator <O1> ↑adjacent-cell <C1>)
    (state <S1> ↑binding <B1> ↑binding { <> <B1> <B2> })
    (binding <B1> ↑tile <T1> ↑cell <C2>)
    (tile <T1> ↑name blank)
    (binding <B2> ↑cell { <> <C2> <C1> } ↑tile { <> <T1> <T2> })
    (cell <C2> ↑cell <C1>)
    (desired <D1> ↑binding <DB1>)
    (binding <DB1> ↑cell <C2> ↑tile <T2>)
  -->
    (evaluation <E1> ↑value 1))
```

Figure 2-17: Production built by chunking the evaluation subgoal.

3. Discussion

The Soar architecture has been fully described in the previous section. However, the consequences of an architecture are hardly apparent on surface examination. The collection of tasks that Soar has accomplished, exhibited in Figure 1, provides some useful information about viability and scope. However, simply that Soar can perform these tasks — that the requisite additional knowledge can be added — is not entirely surprising. The mechanisms in Soar are variants of mechanisms that have emerged as successful over the history of AI research. Soar's accomplishing these tasks does provide answers to other questions as well. We take up some of these here. This discussion also attempts to ensure that Soar's mechanisms and their operation are clear. We limit ourselves to aspects that will shed light on the architecture. The details of Soar's behavior on specific tasks can be found in the references.

The first question we take up is what Soar is like when it runs a real task consisting of multiple aspects with varying degrees of knowledge. The second question is how Soar embodies the weak methods, which form the foundation of intelligent action. The third question involves learning by chunking.

3.1. Combining knowledge and problem solving

R1 is a well-known large knowledge-intensive expert system — consisting of 3300 rules plus a data base of over 7000 component descriptions, circa 1984 — used at Digital Equipment Corporation to configure Vax and PDP-11 computers [3, 41]. R1-Soar is an implementation in Soar of a system that exhibits about 25% of the functionality of R1, using the same knowledge as obtained from R1's Ops5 rules [65, 75]. This is a big enough fraction of R1 to assure that extension to a complete version would be straightforward, if desired.[13] The part covered includes the most involved activity of the system, namely, the assignment of modules to backplanes, taking into account requirements for power, cabling, etc.

R1-Soar was created by designing a set of problem spaces for the appropriate subpart of the configuration task. The problem spaces were added to the basic Soar system (the architecture plus the default knowledge, as described in the previous section). No task-dependent search-control knowledge was included. The resulting system was capable of accomplishing the configuration subtask, although with substantial search. R1-Soar's behavior was initially explored by adding various amounts of search control and by turning chunking on and off. Later experiments were run with variations in the problem spaces and their organization. Thus, R1-Soar is a family of systems, used to explore how to combine knowledge and problem solving.

In the eight puzzle there was a single operator which was realized entirely by productions within a single problem space. However, the configuration task is considerably more complicated. In an extended version of

[13]Indeed, a revision of R1 is underway at DEC that draws on the problem structure developed for R1-Soar [76].

R1-Soar [75], which covered about 25% of R1 (compared to about 16% in the initial version [65]), there were thirty-four operators. Twenty-six of the operators could be realized directly by productions, but eight were complex enough to require implementation in additional problem spaces. Figure 3-1 shows the nine task spaces used in the extended version of R1-Soar. This structure, which looks like a typical task-subtask hierarchy, is generated by the implementation of complex operators. In operation, of course, specific instances of these problem spaces were created, along with instances of the selection problem space. Thus, Figure 3-1 represents the logical structure, not the dynamic subgoal hierarchy.

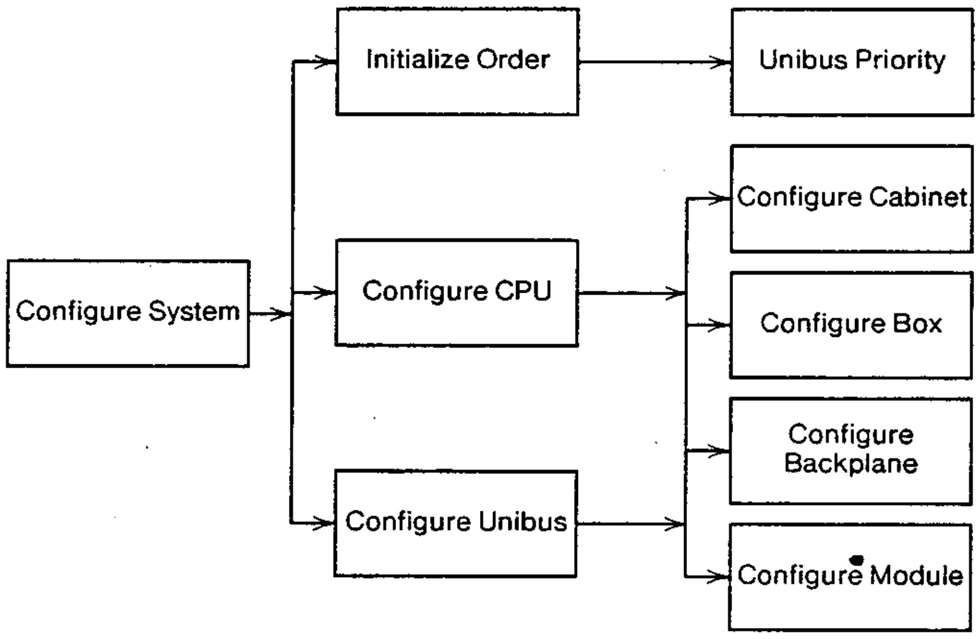


Figure 3-1: Task problem spaces for the extended version of R1-Soar [75].

The total set of task operators is given in Figure 3-2. Many operators are generic and have instantiations, a feature of the operator in the eight-puzzle task as well. However, in R1-Soar, some of the instantiations of the same operator have quite distinct character. Two problem spaces, configure-cpu and configure-unibus, make use of the same generic operators (although they instantiate them differently), such as configured-cabinet. This accounts for Figure 3-1 not being a pure hierarchy, with both configure-cpu and configure-unibus linking to the same four subspaces.

The task decomposition used by R1-Soar is very different than the one used by R1. Soar is a problem solver capable of working in lean spaces by extensive search. R1 is a knowledge-intensive shallow expert system, in which as much direct recognition and as little search as possible is done. It is built around a very large pre-established subtask hierarchy (some 321 subtasks, circa 1984) plus a database containing templates

PROBLEM-SPACE	*OPERATOR*
configure-system	initialize order
	configure CPU
	configure unibus
	instance = place modules in sequence
	instance = maximum module placement
	show output
initialize-order	get component data from database
	assign unibus-module priority numbers
unibus-priority	sequence unibus modules
configure-cpu	configure cabinet
	instance = cpu cabinet
	configure box
	instance = cpu box
	configure backplane
	instance = cpu backplane
	configure module
	instance = maximum module placement
	unused component
	go to previous slot
configure-unibus	configure cabinet
	instance = unibus cabinet
	instance = empty cabinet
	configure box
	instance = unibus box
	instance = empty box
	configure backplane
	instance = unibus backplane
	instance = empty backplane
	instance = unibus repeater
	instance = special backplane
	configure module
	instance = place modules in sequence
	instance = maximum module placement
	unused component
	remove backplane
	instance = replace backplane with repeater
	instance = put backplane in next box
configure-cabinet	configure cabinet
	add component to order
configure-box	configure box
	next cabinet
	install unibus repeater
	add component to order
configure-backplane	configure backplane
	next section
	next box
	install unibus repeater
	add component to order
configure-module	configure module in special backplane
	configure module with one board
	configure module with more than one board
	next slot

Figure 3-2: Task operators for the extended version of R1-Soar [75].

for the variety of components available. R1-Soar was given a set of basic spaces that corresponded closely to the physical manipulations used in configuring computers. The component templates are encoded as rules that implement the operator that adds components to the order. It thus has an appropriate physical model in terms of which to do basic reasoning about the task.

The use of basic spaces in the initial version of R1-Soar was deliberate, to demonstrate that a general problem solver (Soar) could operate in knowledge-intensive mode; and could also mix search-intensive and knowledge-intensive modes as appropriate, dropping back to search whenever the task demanded it (and not by predesign). To do this, Soar was given only the task-implementation knowledge — the basic spaces, desired states, and path constraints — without heuristic search control. Expertise was then to be given by adding search control. Thus, in one small configuration task the base system (no domain-dependent search control at all) took 1731 decision cycles to solve the task; a version with a small amount of search control took 243 cycles; and a version with a large amount of search control (equal to that in the original R1) took 150 cycles [65].[14] One surprise in this experiment was how little search control was involved in moving to the knowledge-intensive versions. Thus, the base system contained a total of 232 rules (for basic Soar plus the configuration task); only two productions were added for the small amount of search control; and only 8 more productions for the large amount of search control (for a total of 242). Thus, there is no correspondence at all between the number of productions of R1 and the productions of R1-Soar.

The version of R1-Soar described in Figures 3-1 and 3-2 extended the coverage of the system beyond the initial version and modified the problem spaces to allow it to run larger orders more efficiently. The previously separate rules for proposing and checking the legality of an operator (using acceptable and reject preferences) were combined into a single rule that only made the operator acceptable when it was legal. Also, additional domain-dependent search-control productions were added (a total of 27 productions for the nine spaces). These changes converted R1-Soar to a system somewhat more like the original R1. Figure 3-3 shows the performance of this system on a set of 15 typical orders. This figure gives a brief description of the size of the order (Components) and the number of decision cycles taken to complete the order (Decisions). From the performance figures we see that the times range from one to three minutes and reflect the amount of work that has to be done to process the order, rather than any search (approximately 60 decisions + 7 decisions/component). The extended version of R1-Soar pretty much knows what needs to be done. These times are somewhat slower than the current version of R1 (about a factor of 1.5, taking into account the speed differences of the Ops5 systems involved). This is encouraging for an experimental system, and more recent improvements to Soar have improved its performance by a factor of 3 [70].

3.2. Weak Methods

Viewed as behavior, *problem-solving methods* are coordinated patterns of operator applications that attempt to attain a goal. Viewed as behavioral specifications, they are typically given as bodies of code that can control behavior for the duration of the method, where a selection process determines which method to use for a

[14] These runs took about 29, 4 and 2.5 minutes respectively on a Symbolics 3600 running at approximately one decision cycle per second. Each decision cycle comprises about 8 production firings spread over two cycles of the elaboration phase (because of the parallel firing of rules).

	Tasks														
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15
Components	5	5	2	7	5	8	2	3	5	5	15	2	11	7	9
Decisions	88	78	78	196	94	100	70	74	88	90	173	78	124	123	129

Figure 3-3: Performance of the extended version of R1-Soar (without learning).

given attempt. In Soar, methods are specified as a collection of search-control productions within a set of related problem spaces — a given *task problem space* and its subspaces. Analogously to a code body, such a collection can be coordinated by making the search-control productions conditional on the method name (plus perhaps other names for relevant subparts), where method selection occurs by establishing the method name in working memory as part of a goal or state. Thus, methods in Soar can be handled according to the standard scheme of selecting among pre-established specifications.

Method behavior may also emerge as the result of problem solving being guided by the appropriate knowledge, even though that knowledge has not been fashioned into a deliberate method (however specified). Behind every useful method is knowledge about the task that justifies the method as a good (or at least possible) way to attain the goal. As bodies of code, methods are simply the result of utilizing that knowledge at some prior design time, in an act of program synthesis. The act of program synthesis brings together the relevant knowledge and packages it in such a way that it can be directly applied to produce behavior. What normally prevents going directly from knowledge to action at behavior time is the difficulty of program synthesis. However, under special conditions direct action may be possible, hence avoiding the task of program synthesis into a stored method, and avoiding the pre-choice of which knowledge is relevant for the task. Instead, whatever knowledge is relevant at the time of behavior is brought to bear to control behavior. Although no prepackaged method is being used, the behavior of the system follows the pattern of actions that characterize the method.

This is the situation with Soar in respect to the weak methods[15] — methods such as depth-first search, hill climbing, and means-ends analysis. This situation arises both because of the nature of the weak methods and because of the nature of Soar. First, the weak methods involve relatively little knowledge about the task [47]. Thus, the generation of behavior is correspondingly simple. Second, all the standard weak methods are built on heuristic search. Thus, realizing their behavior within Soar, which is based on problem spaces, is relatively straightforward. In addition, search control in Soar is realized in a production system with an additive elaboration phase and no built-in conflict resolution. Thus, new search control can be added without regard to the existing search control, with the guarantee that it will get considered. Of course, the relevant total

[15]We have called this a *universal weak method*, on the analogy that Soar behaves according to any weak method, given the appropriate knowledge about the task [31].

search-control knowledge does interact in the decision procedure, but according to a relatively clean semantics that permits clear establishment of the role of each bit of added knowledge.

Our previous example of steepest ascent hill climbing in Figure 2-15 provides an illustration of these three factors. First, the central knowledge for hill climbing is simply that newly generated states can be compared to each other. The comparison may itself be complex to compute, but its role in the method is simple. Second, the other aspects of hill climbing, such as the existence of operators, the need to select one, etc., are implicit in the problem-space structure of Soar. They do not need to be specified. Third, the knowledge to climb the hill can be incorporated simply by search-control productions that add preferences for the operators that produce better states. No other control is necessary and hence complex program synthesis is not required. In short, Soar can be induced to hill climb simply by providing it the knowledge of a specific function that permits states to be compared plus the knowledge that an operator that generates a better state is to be preferred.

Methods require two types of knowledge. The first is about aspects of objects or behavior. Examples are the position of the blank square in the eight puzzle or the number of moves taken since the blank was in the center. Such knowledge says nothing about how a system *should* behave. The second type of knowledge provides the linkage from such objective descriptions to appropriate action of the system. For the weak methods in Soar this takes the especially simple form of single productions that have objective task descriptions as conditions and produce preferences for behavior as actions. No other coordinative productions are required, such as cuing off the name of the method or explicitly asserting that one action should follow another as in a sequential program. Sometimes several control productions are involved in producing the behavior of a weak method, but each are independent, providing links between some aspect of task structure and preferences for action. For instance a depth-limited lookahead has one production that deals with the evaluation preferences and one that deals with enforcing the depth constraint. Soar would produce appropriate (though different) behavior with any combination of these productions. Another important determiner of a method may be specialized *task structure*, rather than any deliberate responses encoded in search control. As a simple instance, if a problem space has only one operator, which generates new states that are candidates for attaining the task, then generate-and-test behavior is produced, without any search control in addition to that defining the task.

The methods listed in Figure 3-4 constitute the aggregate that have been realized in the various versions of Soar, mostly in Soar 1 [31] and Soar 2 [29], where deliberate explorations of the universal weak method were conducted. The purpose of these explorations was to demonstrate that each of the weak methods could be realized in Soar. Most of the weak methods were realized in a general form so that it was clear that the method could be used for any task for which the appropriate knowledge was available. For a few weak methods, such as analogy by implicit generalization and simple abstraction planning, the method was realized for a single task, and more general forms are currenly under investigation.

The descriptions of the weak methods in Figure 3-4 are extremely abbreviated, dispensing with the operating environment, initial and terminating conditions, side constraints, and degenerate cases. All these things are part of a full specification and sometimes require additional (independent) control productions. Figure 3-5 shows graphically the structural relationships among the weak methods implemented in Soar 2 [29]. The common task structure and knowledge forms the trunk of a tree, with branches occurring when there is different task structure or knowledge available, making each leaf in the tree a different weak method. Each of the additions as one goes down the tree are independent control productions.

These simple schemes are more than just a neat way to specify some methods. The weak methods play a central role in attaining intelligence, being used whenever the situation becomes knowledge lean. This occurs in all situations of last resort, where the prior knowledge, however great, has finally been used up without attaining the task. This also occurs in all new problem spaces, which are necessarily knowledge lean. The weak methods are also the essential drivers of knowledge acquisition. Chunking necessarily implies that there exists some way to attain goals before the knowledge has been successfully assimilated (i.e., before it has been chunked). The weak methods provide this way. Finally, there is no need to *learn* the weak methods themselves as packaged specifications of behavior. The task descriptions involved must be acquired and the linkage of the task descriptions to actions. But these linkages are single isolated productions. Once this happens, behavior follows automatically. Thus, this is a particularly simple acquisition framework that avoids any independent stage of program synthesis.

3.3. Learning

The operation of the chunking mechanism was described in detail in the previous section. We present here a picture of the sort of learning that chunking provides, as it has emerged in the explorations to date. We have no indication yet about where the limits of chunking lie in terms of its being a general learning mechanism [36].

3.3.1. Caching, within-trial transfer and across-trial transfer

Figure 3-6 provides a demonstration of the basic effects of chunking, using the eight puzzle [33]. The left-hand column (no learning) show the moves made in solving the eight puzzle without learning, using the representation and heuristics described in the prior section (the evaluation function was used rather than the mea-operator-selection heuristic). As described in Figures 2-14 and 2-15, Soar repeatedly gets a tie impasse between the available moves, goes into the selection problem space, evaluates each move in an incarnation of the task space, chooses the best alternative, and moves forward. Figure 3-6 shows only the moves made in the

Heuristic search. Select and/or reject candidate operators and/or states.

Avoid Duplication. Produce only one version of a state. (Extend: an essentially identical state.)

Operator Subgoaling. If an operator does not apply to the current state, find a state where it does.

Match. Put two patterns containing variables into correspondence and bind variables to their correspondents.

Hypothesize and Match. Generate possible hypothesis forms and match them to the exemplars.

And-Or heuristic search. Makes all moves at and-states and selects moves at or-states until goal is attained.

Waltz Constraint Propagation. Repeatedly propagate the restrictions in range produced by applying constraints in variables with finite ranges.

Means-Ends Analysis. Make a move that reduces the difference between the current state and the desired state.

Generate and Test. Generate candidate solutions and test each for success; terminate when found.

Breadth-First Search. Make a move from a state with untried operators at the least depth.

Depth-First Search. Make a move from a state with untried operators at the greatest depth.

Lookahead. Consider all terminal states to max-depth.

Simple Hill Climbing. Make a move that increases a given value.

Steepest Ascent Hill Climbing. Make a move that increases a given value most from the state.

Progressive Deepening. Repeatedly move depth-first until new information is obtained, then return to initial state for repeat.

Modified Progressive Deepening. Progressive Deepening with consideration of all moves at each state before extension.

B* (Progressive Deepening). Progressive Deepening with optimistic and pessimistic values at each state (not a proof procedure).

Mini-Max. Make moves of each player until can select the best move for each player.

Depth-Bounded Mini-Max. Mini-Max with max-depth bound.

Alpha-Beta. Depth-Bounded Mini-Max, without lines of play that cannot be better than already examined moves.

Ordered Alpha-Beta. Alpha-Beta with the moves tried in a heuristic order.

Iterative Deepening. Repeat ordered Alpha-Beta with increasing depth bound (from 1 to max-depth), with each ordering improved.

B* (Mini-Max). Analogous to Alpha-Beta, with each state having optimistic and pessimistic values [5].

Branch and Bound. Heuristic search, without lines of search that cannot be better than already examined moves.

Best-First Search. Move from the state produced so far that has the highest value.

Modified Best-First Search. Best-First Search with one-step lookahead for each move.

A*. Best-First Search on the minimum depth (or weighted depth).

Exhaustive Maximization. Generate all candidate solutions and pick the best one.

Exhaustive Maximization with Cutoffs. Exhaustive Maximization without going down paths to candidate solutions that cannot be better than the current best candidate.

Macro-Operators for Serially-Decomposable Goals [28]. Learn and use macro-operators that span regions where satisfied goals are violated and reinstated.

Analogy by Implicit Generalization. Find a related problem, solve the related problem, and transfer the generalized solution path to the original problem.

Simple Abstraction Planning. Analogy by Implicit Generalization in which the related problem is an abstract version of the original problem.

Figure 3-4: Weak methods, as patterns of behavior.

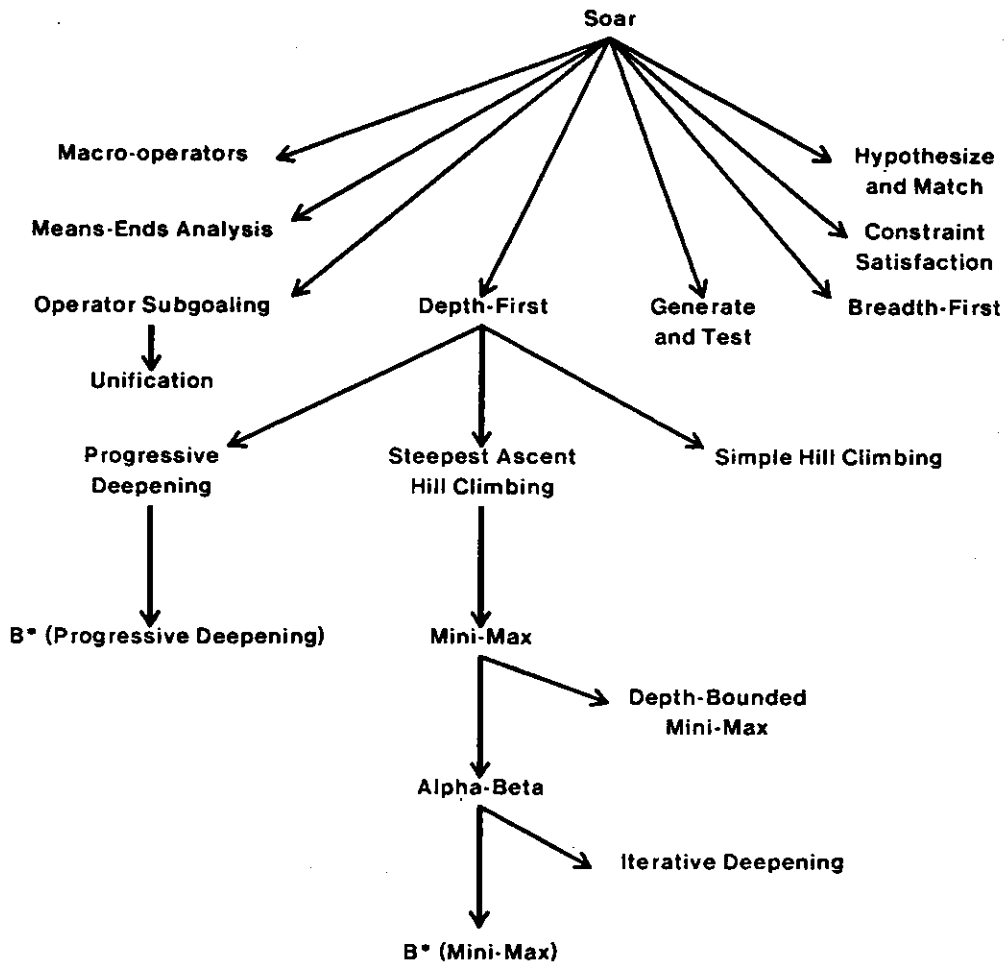


Figure 3-5: Structure of weak methods realized in Soar [29].

task space, coalescing the various incarnations of it. Each state, except for the initial and desired states, is shown as a black square. The move made to reach the state is shown as a single letter (either Left, Right, Up, or Down). Soar explores 20 states in all to solve this problem.

The second column (with learning) has chunking turned on. Although Soar starts out examining the same states as in the run without learning (L, U and R in each of the first two levels), it soon deviates. The chunking that occurs in the early part of the task already becomes effective in the later part. This is *within-trial transfer*. It answers one basic question about chunking — whether it will provide any transfer at all to new situations, or only simple practice effects. Not only is there transfer, but it occurs on the initial performance — a total of 15 states is examined, compared to 20 without learning. Thus, with Soar, no rigid behavioral separation is possible between performance and learning — learning becomes integral to every performance.

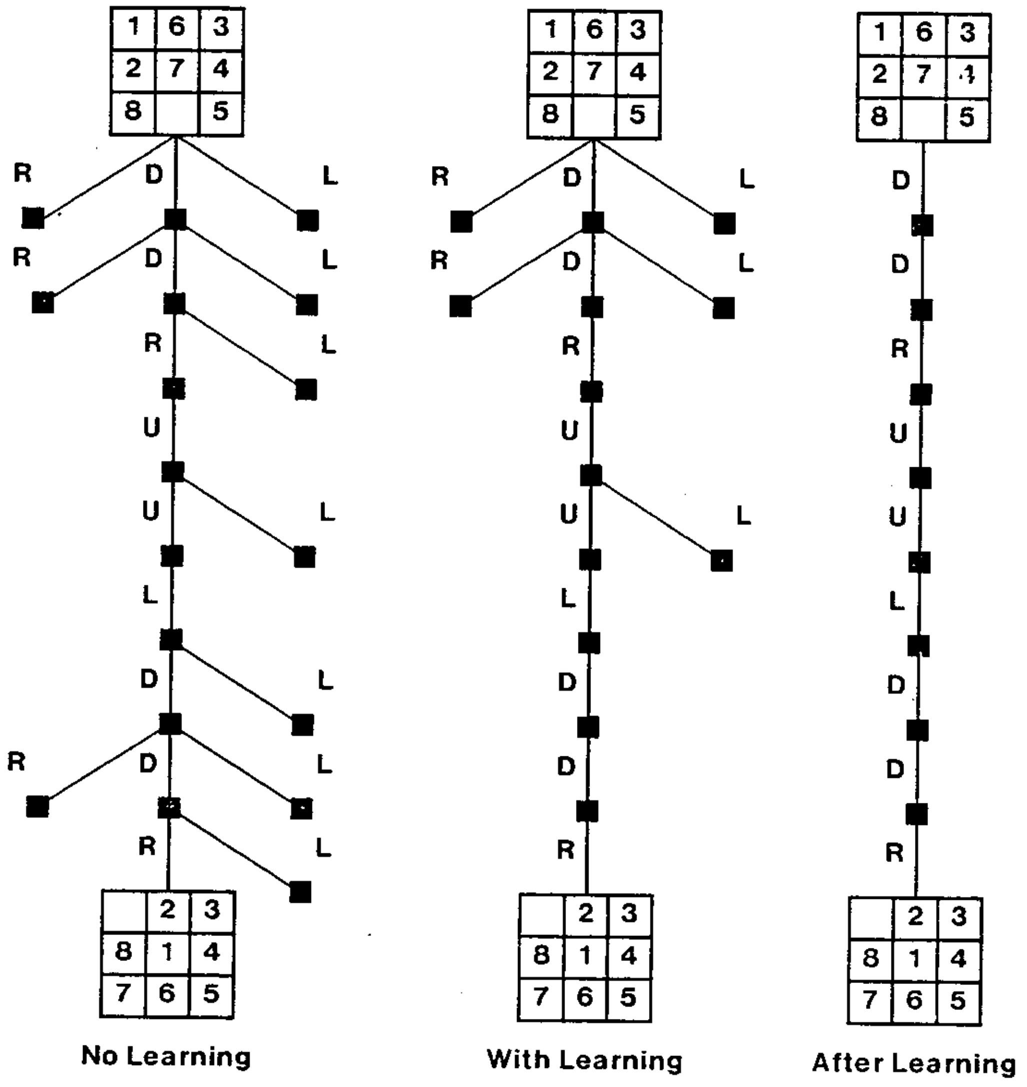


Figure 3-6: Learning in the eight puzzle [33].

If Soar is run again after it has completed its with-learning trial, column 3 (after learning) results. All of the chunks to be learned in this task have been learned during the one with-learning trial, so Soar always knows which move to make. This is the direct effect of practice — the use of results *cached* during earlier trials. The number of states examined (10) now reflects the demands of the task, not the demands of finding the solution. This improvement depends on the original evaluation function being an accurate measure of progress to the goal. Chunking eliminates the necessity for the look-ahead search, but the path Soar takes to the goal will still be determined by the evaluation function cached in the chunks.

Figure 3-7 shows *across-task transfer* in the Eight Puzzle. The first column (task 1, no learning) is the same trace as the first column in Figure 3-6. In the second column (task 2, during learning) Soar has been started over from scratch and run on an entirely different eight-puzzle task — the initial and final positions are different from those of task 1, as are all the intermediate positions. This is preparation for the third column (task 1, after learning about task 2 but without any learning during task 1), where Soar shows across-task transfer. If the learning on task 2 had no effect, then this column would have been identical to the original one on task 1 (first column), whereas it takes only 16 states rather than 20.

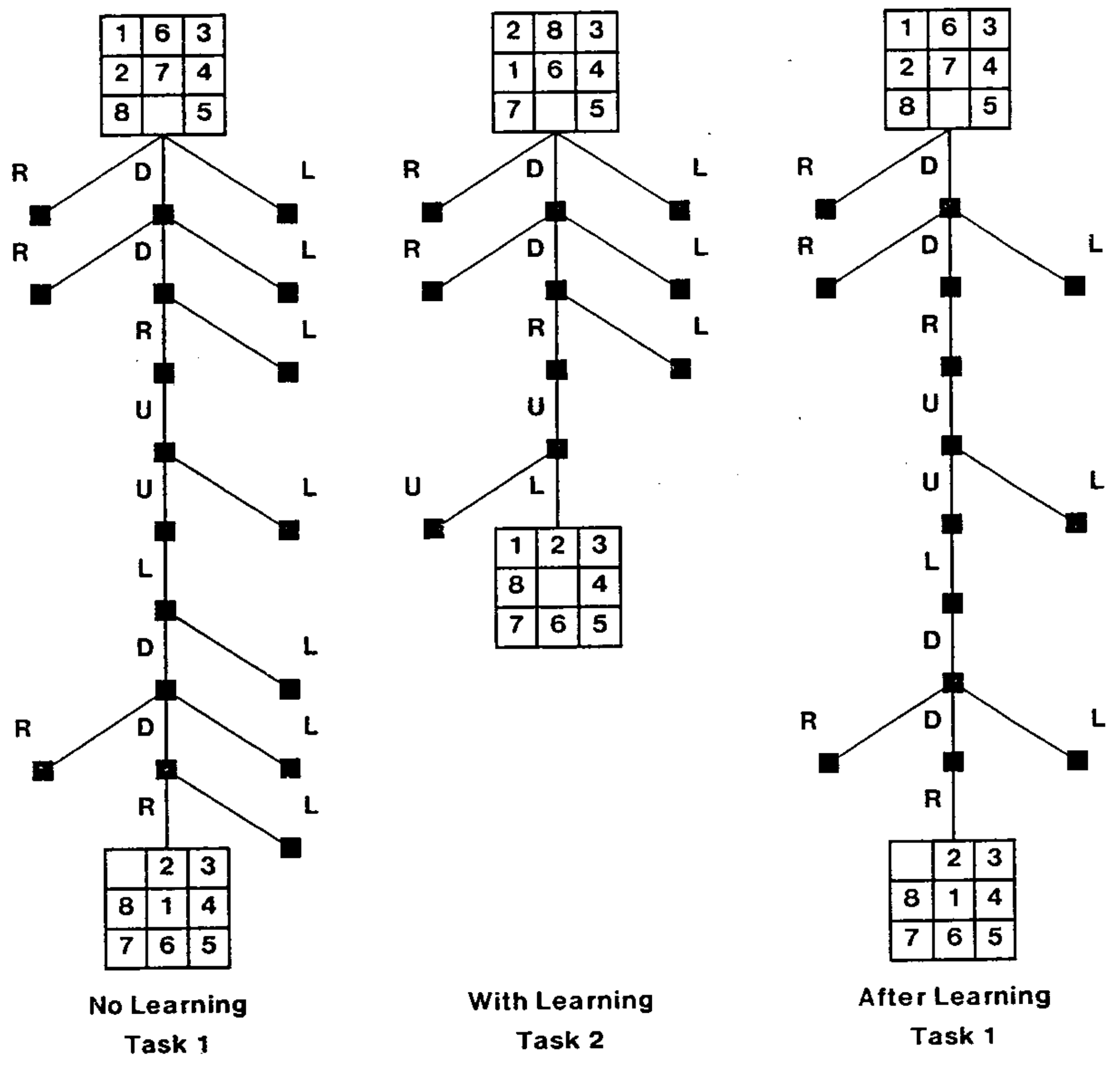


Figure 3-7: Across-task transfer in the eight puzzle [33].

What Soar has learned in these runs is search control to choose moves, and rules which implement the evaluate-object operators. The comparison based on the evaluation function is cached into productions that create preferences based on direct comparisons between the current and desired states. In this example, chunking does not improve the evaluation function. If the evaluation function is imperfect, as it is in this case, the imperfections are included in the chunks. Also in this example, no eight-puzzle operators have been learned because the operator was already realized directly by productions in the task space. But if the operator had required subspaces for implementation (as the evaluate-object operator in the selection problem space did), it would have been learned as well.

3.3.2. Learning in an expert-system task

A striking feature of chunking is that it applies automatically to every task Soar performs, without modification of Soar or any special additions. For example, the investigations that used R1-Soar to show that general problem-solving capability can be combined with domain expertise (by adding domain-dependent search control to a basic task representation) became immediately a demonstration that the domain expertise can be acquired automatically. Figure 3-8 shows that on the task mentioned above that took 1731 decision cycles with no domain-dependent search control, a pattern of results emerged that followed exactly the pattern on the eight puzzle [65].

R1-Soar Version	Decisions with no learning	Decisions with learning	Decisions after learning
Base	1731	485 72% [+59]	7
Partial [+2]	243	111 54% [+14]	7
Full [+8]	150	90 40% [+12]	7

Figure 3-8: Learning in R1-Soar.

The first column of Figure 3-8 shows the effects of the manual addition of search control from none for the basic version, to 2 productions for the partial version, to 8 more productions (for a total of 10 search control productions) for the full version. This was the basic investigation, and no learning was involved. The second column shows the effect on performance of running with chunking turned on — the number of decision cycles, the percent improvement over the trial without learning, and the number of chunks learned. There is within-task transfer, just as in the eight puzzle. As the system starts with more initial knowledge, the effect diminishes (from 72% to 54% to 40%) but the effect is appreciable in all cases. Finally, the result of rerunning the task after learning is complete is to reduce the task to its necessary processing steps (namely, 7). The automatic acquisition of knowledge does involve the addition of many more productions than was involved in the manual acquisition (shown in brackets in the second column), because the chunks are more specific than the manually encoded rules.

The extensive test on the extended version of R1-Soar yielded additional data on learning, as shown in the four right-hand columns of Figure 3-9. In these runs, chunking occurred from the bottom up, that is, chunks were built for a goal only if no subgoals occurred. Enough runs with bottom-up chunking will yield the same results as all-at-once chunking (which was used in both the eight puzzle and initial R1-Soar cases). Bottom-up chunking has the advantage of tending to create only the chunks that have a greater chance of being repeatedly used. The higher up in the subgoal hierarchy (measured from the bottom, not the top), the more specific a chunk becomes — it performs a larger proportion of the task — and the less chance it has to be used [50]. Thus, in R1-Soar all-at-once chunking will create many productions that will never be evoked again in any but identical reruns of the same task. Figure 3-9 shows two passes of bottom-up chunking (Pass 2 and Pass 4), embedded in three passes with chunking turned off to assess the effects (Pass 1, Pass 3, and Pass 5), giving a total of 30 trials with chunking. The test mimics what would be expected in the real situation with an expert system, namely that the chunk-productions accumulate throughout the entire series of 30 chunking runs (and remain fixed during the learning-off passes).[16]

	Pass 1	Pass 2	Pass 3	Pass 4	Pass 5
	Before	During	After	During	After
TASK	(learn off)	(learn on)	(learn off)	(learn on)	(learn off)
T1	88	88 [6]	44	44 [3]	9
T2	78	68 [4]	40	40 [3]	9
T3	78	78 [6]	38	38 [3]	9
T4	196	174 [14]	113	113 [6]	58
T5	94	84 [6]	48	48 [3]	9
T6	100	85 [3]	48	48 [3]	9
T7	70	48 [3]	38	38 [3]	9
T8	74	59 [3]	40	40 [3]	9
T9	88	73 [3]	42	42 [3]	9
T10	90	75 [3]	48	48 [3]	9
T11	173	158 [10]	86	86 [2]	48
T12	78	52 [3]	38	38 [3]	9
T13	124	102 [7]	58	58 [3]	9
T14	123	108 [7]	67	67 [4]	28
T15	129	109 [5]	64	64 [2]	28
Productions		----		----	
Total:	314	397 [83]	397	444 [47]	444

Figure 3-9: Performance of the extended version of R1-Soar (with bottom-up learning) [75].

The figure reveals several interesting features. First, there is a 14% average improvement during the first learning pass. This is primarily due to within-trial transfer in each of the 15 tasks. There is only a small effect due to across-task transfer, both positive and negative. Negative transfer comes about from overly-general search-control chunks that guide the problem solving down the incorrect path. Recovery from the misguided

[16]Thus, the table is not to be read as if it were 15 independent little learning experiments.

search occurs, but it takes time. On Pass 3, the assessment pass after the first learning pass, there is a substantial improvement, reflecting the full force of the cached chunks: an additional drop of 35% from the original times, for a total savings of 49% of the original times. The second learning pass (Pass 4) leads to no further within-task or across-task transfer — the times on this pass are identical to the times on the prior assessment pass. But after this second learning pass is completed, the final assessment pass (Pass 5) shows another large drop of 35% from the original times, yielding a total drop of 84% from the original times. All but four large tasks have reached their minimum (all at 9 steps). Thus the contribution of this second pass has been entirely to cache results that then do not have to be performed on a rerun.

The details of this version of R1-Soar and the test must be taken with caution, yet it confirms some expectations. This extended version has substantial domain-dependent knowledge, so we would not expect as much improvement as in the earlier version, even beyond the effect of using bottom-up chunking. Investigation of the given productions in the light of the transfer results reveals that many of them test numerical constants where they could have tested for inequality of two values, and the constant tests restricted their cross-situational applicability. But even so, we see clearly that the transfer action comes from the lowest level chunks (the first pass), which confirms theoretical expectations that they have the most generality. And, more globally, learning and performance always go together in Soar in accomplishing any task.

3.3.3. Chunking, generality, and representation

Chunking is a learning scheme that integrates learning and performance. Fundamentally, it simply records problem-solving experience. Viewed as knowledge acquisition, it combines the existing knowledge available for problem solving with knowledge of results in a given problem space, and converts it into new knowledge available for future problem solving. Thus it is strongly shaped by the knowledge available. This integration is especially significant with respect to generalization — to the transfer of chunks to new situations (e.g., as documented above). Generalization occurs in two ways in Soar chunking. One is *variablization* (replacing identifiers with variables), which makes Soar respond identically to any objects with the same description (attribute-value augmentations). This generalization mechanism is the minimum necessary to get learning at all from chunking, for most identifiers will never occur again outside of the particular context in which they were created (e.g., goals, states, operator instantiations).

The second way in which generalization occurs is *implicit generalization.* The conditions that enter into a new chunk-production are based only on those working-memory elements that both existed prior to the creation of the goal and affected the goal's results. This is simple abstraction — ignoring everything about a situation except what has been determined at chunk-creation time to be relevant. It is enabled by the natural abstraction of productions — that the conditions only respond to selected aspects of the objects available in the working memory. If the conditions of a chunk do not test for a given aspect of a situation, then the chunk will ignore whatever that aspect might be in some new situation.

A good example is provided by the implementation in Soar of Korf's technique for learning and using *macro-operators* [28]. Korf showed that any problem that is serially decomposable — that is, when some ordering of the subgoals exists in which each subgoal is dependent only on the preceding subgoals, and not on the succeeding ones — can have a *macro table* defined for it. Each entry in the table is a macro-operator — a sequence of operators that can be treated as a single operator [19]. For the eight puzzle, a macro table can be created if the goals are, in order: (1) place the space in its correct position; (2) place the space and the first tile in their correct positions; (3) place the space, the first tile, and the second tile in their correct positions; etc. Each goal depends only on the locations of the tiles already in position and on the location of the one new tile. The macro table is a simple two dimensional structure in which each row represents a goal, and each column represents the position of the new tile. Each macro-operator specifies a sequence of moves that can be made to satisfy the goal, given the current position of the new tile (the positions of the previously placed tiles are fixed). The macro table enables efficient solutions from any initial state of the problem to a particular goal state.

Implementing this in Soar requires two problem spaces, one containing the normal eight-puzzle operators (up, down, left, right), and one containing operators corresponding to the serially-decomposable goals, such as place the space and the first tile in their correct positions [36]. Problem solving starts in this latter problem space with the attempt to apply a series of the high-level operators. However, because these operators are too complex to encode directly in productions, they are implemented by problem solving in the normal eight-puzzle problem space.

Based on this problem solving, macro-operators are learned. Each of these macro-operators specifies the sequence of eight-puzzle operators that need to be applied to solve a particular higher-level goal for a particular position of the new tile. These macro-operators then lead to efficient solutions for a large class of eight-puzzle problems, demonstrating how choosing the right problem solving decomposition can allow a simple caching scheme to achieve a large degree of generality. The generality, which comes from using a single goal in many different situations, is possible only because of the implicit generalization that allows the macro-operators to ignore the positions of all tiles not yet in place. If the identities of the not-yet-placed tiles are not examined during problem solving, as they need not be, then the chunks will also not examine them. The subgoal structure by itself does not tap all of the possible sources of generality in the eight puzzle. One additional source of generality comes from transfer between macro-operators. Rather than a macro-operator being encoded as a monolithic data structure that specifies each of the moves, it is represented in Soar as a set of search-control rules that select the appropriate eight-puzzle operator at each state. These rules are general enough to transfer across different macro-operators. Because of this transfer, only 112 productions are required to encode all 35 of the macro-operators, rather than the 170 that would otherwise be required.

One of the most important sources of generality is the *representation* used for the task states. Stated generally, if the representation is organized so that aspects that are relevant are factored cleanly from the parts that are not (i.e., are noise) then chunking can learn highly general concepts. Factoring implies both that the aspects are encoded as distinct attributes and that the operators are sensitive only to the relevant attributes and not to the irrelevant attributes. One representational possibility for the eight-puzzle state is a two-dimensional array, where each array cell would contain the number of the tile that is located at the position on the board specified by the array indices. Though this representation is logically adequate, it provides poor support for learning general rules in Soar. For example, it is impossible to find out which tiles are next to the blank cell without looking at the numbers on the tiles and the absolute positions of the tiles. It is thus impossible, using just implicit generalization, to abstract away these irrelevant details. Though this is not a good representation for the eight puzzle, the results presented in the previous paragraphs, which were based on this representation, show that even it provides significant transfer.

By adopting a better representation that explicitly represents the relative orientation of the tiles and the relationship between where the tile is and where it should be — the representation presented in Section 2.2 — and adding an incremental goal test, the amount of sharing is increased to the point where only 61 productions are required to represent the entire macro table. Because the important relationships are represented directly, and the absolute tile position and name are represented independently of this information, the chunks are invariant over tile identity as well as translation, rotation, and reflection of groups of tiles. The chunks also transfer to different desired states and between macro-operators for different starting positions, neither of which were possible in Korf's original implementation.

Figure 3-10 shows the most complex case of transfer. The top two boards are intermediate subgoals to be achieved on the path to getting all eight tiles in place. Below them are possible initial states that the relevant tiles might be in (all others are X's). A series of moves must be made to transform the initial state to the corresponding desired intermediate subgoal. The arrow shows the path that the blank takes to move the next tile into position. The paths for both problems are the same, except for a rotation. In Soar, the chunks learned for the first subgoal transfer to the second subgoal, allowing it to be solved directly, without any additional search.

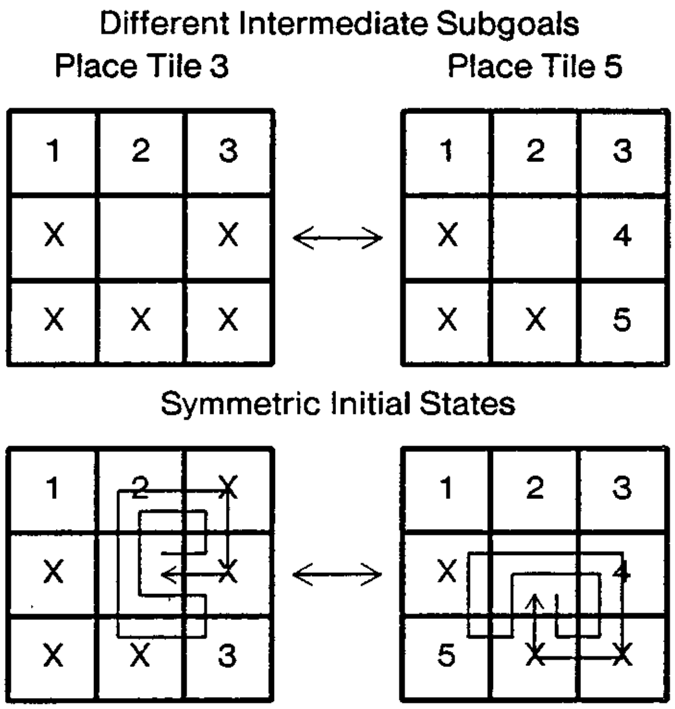


Figure 3-10: Transfer possible with macro-operators in the eight puzzle.

4. Conclusion

Soar embodies eleven basic hypotheses about the structure of an architecture for general intelligence:

1. **Physical symbol-system hypothesis:** A general intelligence must be realized with a symbolic system [52].

2. **Goal-structure hypothesis:** Control in a general intelligence is maintained by a symbolic goal system.

3. **Uniform elementary-representation hypothesis:** There is a single elementary representation for declarative knowledge.

4. **Problem-space hypothesis:** Problem spaces are the fundamental organizational unit of all goal-directed behavior [49].

5. **Production-system hypothesis:** Production systems are the appropriate organization for encoding all long-term knowledge.

6. **Universal-subgoaling hypothesis:** Any decision can be an object of goal-oriented attention.

7. **Automatic-subgoaling hypothesis:** All goals arise dynamically in response to impasses and are generated automatically by the architecture.

8. **Control-knowledge hypothesis:** Any decision can be controlled by indefinite amounts of knowledge, both domain dependent and independent.

9. **Weak-method hypothesis:** The weak methods form the basic methods of intelligence [47].

10. **Weak-method emergence hypothesis:** The weak methods arise directly from the system responding based on its knowledge of the task.

11. **Uniform-learning hypothesis:** Goal-based chunking is the general learning mechanism.

These hypotheses have varying standing in current research in artificial intelligence. The first two, about symbols and goals, are almost universally accepted for current AI systems of any scope. At the opposite end, the weak-method emergence hypothesis is unique to Soar. The remaining hypotheses are familiar in AI, or at least components of them are, but are rarely, if ever, taken to the limit as they are in Soar. Soar uses a problem-space representation for *all* tasks, a goal-based chunking mechanism for *all* learning, and a production system for *all* long-term memory. Many systems use production systems exclusively, but they are all pure performance systems without learning, which does not test the use of productions for declarative memory.

Many aspects of the Soar architecture are not reflected in these eleven hypotheses. Some examples are: automatic goal termination anywhere in the goal hierarchy; the structure of the decision cycle, with its parallel elaboration phase; the language of preferences; the limitation of production actions to addition of working-memory elements; the removal of working-memory elements by the architecture; the restriction of production conditions to test only memory elements accessible through the context stack. There are also details of the mechanisms mentioned in the hypotheses — attribute-value triples, the form of conditions of productions, etc. Some of these are quite important, but we do not yet know in AI how to describe architectures completely in functional terms or which features should be stipulated independently.

Much is still missing in the current version of Soar. Figure 1 pointed out several aspects that are under active investigation. But others are not recorded there — the acquisition of declarative knowledge from the external environment and the use of complex analogies to name a couple. Until Soar has acquired the capabilities to do all of these aspects, there will be no assurance that the Soar architecture is complete or stable.

References

1. Amarel, S. On the representation of problems of reasoning about actions. In *Machine Intelligence 3*, Michie, D., Ed., American Elsevier, New York, 1968, ch. 10, pp. 131-171.

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

3. Bachant, J. & McDermott, J. "R1 revisited: Four years in the trenches". *AI Magazine 5* (1984).

4. Balzer, R., Erman, L. D., London, R. & Williams, C. HEARSAY-III: A domain-independent framework for expert systems. Proceedings of AAAI1, Los Altos, CA, 1980.

5. Berliner, H. J. "The B* tree search algorithm: A best-first proof procedure". *Artificial Intelligence 12* (1979), 201-214.

6. Boggs, M. & Carbonell, J. A Tutorial Introduction to DYPAR-1. Computer Science Department, Carnegie-Mellon University.

7. Bower, G. H. & Winzenz, D. "Group structrure, coding and memory for digit series". *Journal of Experimental Psychology Monograph 80* (1969), 1-17. (May, Pt. 2).

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

9. Buchanan, B. G. & Shortliffe, E. H.. *Rule-Based Expert Systems: The Mycin experiments of the Stanford Heuristic Programming Project.* Addison-Wesley, Reading, MA, 1984.

10. Carbonell, J. G. Learning by analogy: Formulating and generalizing plans from past experience. In *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell, & T. M. Mitchell, Eds., Tioga, Palo Alto, CA, 1983.

11. Card, S. K., Moran, T. P. & Newell, A. "Computer text editing: An information-processing analysis of a routine cognitive skill". *Cognitive Psychology 12*, 1 (1980), 32-74.

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

13. Clancey, W. J. "The epistemology of a rule-based expert system: A framework for explanation". *Artificial Intelligence 20* (1983), 215-251.

14. Davis, R. "Meta-rules: Reasoning about control". *Artificial Intelligence 15* (1980), 179-222.

15. DeJong, G., & Mooney, R. "Explanation-based learning: An alternative view". *Machine Learning 1*, 2 (1986), 145-176. In press.

16. Erman, L., Hayes-Roth, F., Lesser, V., & Reddy, D. R. "The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty". *Computing Surveys 12* (June 1980), 213-253.

17. Ernst, G. W. & Newell, A.. *GPS: A Case Study in Generality and Problem Solving.* Academic Press, New York, 1969.

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

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

20. Forgy, C. L. OPS5 User's Manual. Computer Science Department, Carnegie-Mellon University, July, 1981.

21. Forgy, C. L. & McDermott, J. OPS, a domain-independent production system language. Proceedings Fifth International Joint Computer Conference, MIT AI Laboratory, Cambridge MA, 1977.

22. Genesereth, M. An overview of meta-level architecture. Proceedings of the Third Annual National Conference on Artificial Intelligence, Los Altos, CA, 1983.

23. Hayes, J. R. & Simon, H. A. Understanding written problem instructions. Knowledge and Cognition, Potomac, MD, 1974.

24. Hayes-Roth, B. "A blackboard architecture for control". *Artificial Intelligence 26* (1985), 251-321.

25. Hayes-Roth, F., Waterman, D. A. & Lenat, D. B. (Eds.). *Building Expert Systems.* Addison-Wesley, Reading, MA, 1983.

26. Kant, E. & Newell, A. An automatic algorithm designer: An initial implementation. Proceedings of AAAI83, Menlo Park, CA, 1983.

27. Korf, R. E. "Towards a model of representation changes". *Artificial Intelligence 14* (1980), 41-78.

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

29. Laird, J. E. *Universal Subgoaling.* Ph.D. Th., Carnegie-Mellon University, 1984.

30. Laird, J. E. *Soar User's Manual: Version 4.0.* Xerox Palo Alto Research Center, 1986.

31. Laird, J. & Newell, A. A Universal Weak Method. Computer Science Department, Carnegie-Mellon University, June, 1983.

32. Laird, J. & Newell, A. A universal weak method: Summary of results. Proceedings of IJCAI-83, Los Altos, CA, 1983.

33. Laird, J. E., Rosenbloom, P. S. & Newell, A. Towards chunking as a general learning mechanism. Proceedings of AAAI-84, National Conference on Artificial Intelligence, American Association for Artificial Intelligence, 1984.

34. Laird, J. E., Rosenbloom, P. S. & Newell, A. Overgeneralization during knowledge compilation in Soar. Proceedings of the Workshop on Knowledge Compilation, Otter Crest, OR, 1986.

35. Laird, J. E., Rosenbloom, P. S. & Newell, A.. *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies.* Kluwer Academic Publishers, Hingham, MA, 1986.

36. Laird, J. E., Rosenbloom, P. S. & Newell, A. "Chunking in Soar: The anatomy of a general learning mechanism". *Machine Learning 1* (1986), 11-46.

37. Langley, P. "Learning to Search: From weak methods to domain-specific heuristics". *Cognitive Science 9* (1985), 217-260.

38. Lenat, D. B. "EURISKO: A program that learns new heuristics and domain concepts. The nature of heuristics III: program design and results". *Artificial Intelligence 20* (1983), 61-98.

39. Lenat, D. B. & Brown, J. S. "Why AM and Eurisko appear to work". *Artificial Intelligence 23* (1984), 269-294.

40. McDermott, D. "Planning and acting". *Cognitive Science 2* (1978), 71-109.

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

42. McDermott, J. & Forgy, C. L. Production system conflict resolution strategies. In *Pattern-directed Inference Systems*, Waterman, D. A. & Hayes-Roth, F., Eds., Academic Press, New York, 1978.

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

44. Mitchell, T. M. *Version Spaces: An approach to concept learning.* Ph.D. Th., Stanford University, 1978.

45. Mitchell, T. M., Utgoff, P. E., & Banerji, R. Learning by experimentation: Acquiring and refining problem-solving heuristics. In *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell, T. M. Mitchell, Eds., Tioga Publishing Co., Palo Alto, CA, 1983.

46. Mostow, D. J. Machine transformation of advice inta a heuristic search procedure. In *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell, & T. M. Mitchell, Eds., Tioga Publishing Company, Palo Alto, CA, 1983, ch. 12.

47. Newell, A. Heuristic programming: Ill-structured problems. In *Progress in Operations Research, III*, Aronofsky, J., Ed., Wiley, New York, 1969, pp. 360-414.

48. Newell, A. Production systems: Models of control structures. In *Visual Information Processing*, Chase, W. C., Ed., Academic Press, New York, 1973, pp. 463-526.

49. Newell, A. Reasoning, problem solving and decision processes: The problem space as a fundamental category. In *Attention and Performance VIII*, R. Nickerson, Ed., Erlbaum, Hillsdale, NJ, 1980.

50. Newell, A. & Rosenbloom, P. Mechanisms of skill acquisition and the law of practice. In *Learning and Cognition*, Anderson, J. A., Ed., Erlbaum, Hillsdale, NJ, 1981.

51. Newell. A. & Simon, H. A.. *Human Problem Solving.* Prentice-Hall, Englewood Cliffs, 1972.

52. Newell, A. & Simon, H. A. "Computer science as empirical inquiry: Symbols and search". *Communications of the ACM 19*, 3 (1976), 113-126.

53. Newell, A., Shaw, J. C. & Simon, H. A. Empirical explorations of the Logic Theory Machine: A case study in heuristics. Proceedings of the 1957 Western Joint Computer Conference, Western Joint Computer Conference, 1957, pp. 218-230. (Reprinted in Feigenbaum, E. & Feldman, J. (Eds.) *Computers and Thought*, New York: McGraw-Hill, 1963).

54. Newell, A., Shaw, J. C., & Simon, H. A. Report on a general problem-solving program for a computer. In *Information Processing: Proceedings of the International Conference on Information Processing*, UNESCO, Paris, 1960, pp. 256-264.

55. Newell, A., Tonge, F. M., Feigenbaum, E. A., Green, B., & Mealy, G.. *Information Processing Language V Manual.* Prentice-Hall, Englewood Cliffs, 1964. 2nd Edition.

56. Nii, H. P. & Aiello, N. AGE (Attempt to Generalize): A knowledge-based program for building knowledge-based programs. Proceedings of the Sixth International Joint Conference on Artificial Intelligence, IJCAI, 1979.

57. Nilsson, N.. *Problem-solving Methods in Artificial Intelligence.* McGraw-Hill, New York, 1971.

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

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

60. Robinson, J. A. "A machine-oriented logic based on the resolution principle". *Journal of the ACM 12* (1965), 23-41.

61. Rosenbloom, P. S. *The Chunking of Goal Hierarchies: A model of practice and stimulus-response compatibility.* Ph.D. Th., Carnegie-Mellon University, 1983. (available as Tech Rep #83-148, Computer Science Department).

62. Rosenbloom, P. S. & Laird, J. E. Mapping explanation-based generalization onto Soar. Proceedings of AAAI-86, National Conference on Artificial Intelligence, American Association for Artificial Intelligence, Philadelphia, 1986.

63. Rosenbloom, P. S., & Newell, A. The chunking of goal hierarchies: A generalized model of practice. In *Machine Learning: An Artificial Intelligence Approach, Volume II,* R. S. Michalski, J. G. Carbonell, & T. M. Mitchell, Eds., Morgan Kaufmann Publishers, Inc., Los Altos, CA, 1986.

64. Rosenbloom, P. S., Laird, J. E. & Newell, A. Meta-levels in Soar. Preprints of the Workshop on Meta-level Architectures and Reflection, Sardinia, 1986.

65. Rosenbloom, P. S., Laird, J. E., McDermott, J., Newell, A., & Orciuch, E. "R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture". *IEEE Transactions on Pattern Analysis and Machine Intelligence 7,* 5 (1985), 561-569.

66. Rychener, M. D. Production systems as a programming langauge for artificial intelligence applications. Computer Science Department, Carnegie-Mellon University, 1976.

67. Rychener, M. D. The instructable production system: A retrospective analysis. In *Machine Learning: An artificial intelligence approach,* Michalski, R. S., Carbonell, J. G. & Mitchell, T. M., Eds., Tioga, Palo Alto, CA, 1983.

68. Rychener, M. D. & Newell, A. An instructable production system: Basic design issues. In *Pattern-Directed Inference Systems,* Waterman, D. A. & Hayes-Roth, F., Eds., Academic Press, New York, 1978, pp. 135-153.

69. Sacerdoti, E. D.. *A Structure for Plans and Behavior.* Elsevier, New York, 1977.

70. Scales, D. Efficient Matching Algorithms for the Soar/Ops5 Production System. Computer Science Department, Stanford University, 1986.

71. Shortliffe, E. H.. *Computer-based Medical Consultations: MYCIN.* American Elsevier, New York, 1976.

72. Simon, H. A. "Search and reasoning in problem solving". *Artificial Intelligence 21* (1983), 7-30.

73. Smith, B. C. Reflection and Semantics in a Procedural Langauge. MIT/LCS/TR-272, Laboratory for Computer Science, MIT, 1982.

74. Smith, D. E. & Genesereth, M. R. "Ordering Conjunctive Queries". *Artificial Intelligence 26* (1985), 171-216.

75. van de Brug, A., Rosenbloom, P. S., & Newell, A. Some Experiments with R1-Soar. Computer Science Department, Carnegie-Mellon University, 1986. (in preparation).

76. van de Brug, A., Bachant, J., & McDermott, J. "The taming of R1". *IEEE Expert 1* (1986), 33-39.

77. VanLehn, K. Felicity Conditions for Human Skill Acquisition: Validating an AI-Based Theory. Xerox Palo Alto Reserch Center, November, 1983.

78. Waterman, D. A. & Hayes-Roth, F., (Eds.). *Pattern Directed Inference Systems.* Academic Press, New York, 1978.

79. Wilensky, R.. *Planning and Understanding: A computational approach to human reasoning.* Addison-Wesley, Reading, MA, 1983.