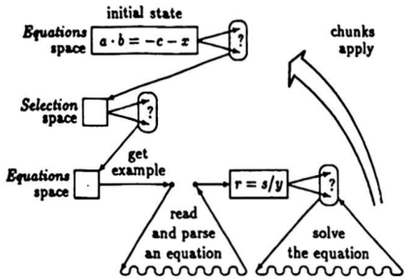

Figure 2: Subgoal structure for learning from an illustrative problem.

IV. Discussion

The system described here represents a first step toward the construction of an agent that is able to improve its behavior merely by taking in advice from the outside. Soar appears ideally suited as a research vehicle to this end, as it provides general capabilities for problem solving and learning that were not available in previous research efforts [McCarthy, 1968, Rychener, 1983]. The relatively straightforward implementation of direct advice and illustrative problems shows that the advice-taking paradigm fits naturally into Soar. Below, these methods of taking advice are compared with related work, and extensions to the straightforward implementations are proposed.

A. Direct Advice

The system is similar to a learning apprentice [Mitchell *et al.*, 1985] in its treatment of direct advice; instead of accepting it blindly, it first explains to itself why it is correct. Nevertheless, the system cannot accurately be called a learning apprentice, as it *actively* seeks advice, as opposed to passively monitoring the user's behavior. In fact, the learning-apprentice style of interaction could be considered a special case of advice-taking in which the guidance consists of a protocol of the user's problem-solving.

The limitation of direct advice is that it forces the advisor to name a particular operator; it would be desirable to allow higher-level specifications of what to do. To take the canonical example of the game of Hearts, the advisor might want to tell the system to play a card that avoids taking points, instead of spelling out exactly which card to play. To accept such indirect advice, the system would have to reduce it to a directly usable form [Mostow, 1983].

B. Illustrative Problems

The system processes an illustrative problem by applying the chunks it learns from that problem to the original one. Since it is solving the two problems in serial order, it may seem that this approach amounts to just working through a graded sequence of exercises. There are two reasons that it does not, however. First, the teacher can observe how the student fails, and take this into account in choosing a suitable illustrative problem. Second, the system is solving the illustrative problem *in service of* the original one; thus it can abandon the illustrative problem as soon as it learns enough to resolve the original impasse.

A more apt way to view the system's processing of illustrative problems is as a type of analogical transfer from the illustrative to the original problem. The trouble with this type of analogy, though, is that the generalizations are based solely on the source problem, without regard for how they will apply to the target. A more effective approach would be to establish a mapping between the two problems explicitly. This forces the system to attend to commonalities between the problems, which would then be captured in its generalizations. This is in fact just the way generalizations are constructed in GRAPES [Anderson, 1986].

V. Conclusion

The system presented here shows how Soar can acquire general search-control knowledge from outside guidance. The guidance can be either direct advice about what the system should do, or a problem that illustrates a relevant idea. The system's strategy of verifying direct advice before accepting it illustrates how Soar can extract general lessons, while protecting itself from erroneous advice. This strategy could be extended by permitting the advice to be indirect; the system would then have to operationalize it. In applying the lessons learned from solving an illustrative problem to its original task, the system demonstrates an elementary form of analogical reasoning. This reasoning capability could be greatly improved, however, if the system were to take into consideration the target problem of the analogy as well as the source problem.

References

John R. Anderson. Knowledge compilation: the general learning mechanism. In *Machine Learning: An Artificial Intelligence Approach*, pages 289–310, Morgan Kaufmann, Los Altos, CA, 1986.

John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: an architecture for general intelligence. *Artificial Intelligence*, 1987. In press.

John E. Laird, Paul S. Rosenbloom,. and Allen Newell. Chunking in Soar: the anatomy of a general learning mechanism. *Machine Learning*, 1, 1986.

John McCarthy. Programs with common sense. In Marvin Minsky, editor, *Semantic Information Processing*, pages 403–418, MIT Press, Cambridge, MA, 1968.

T. Mitchell, S. Mahadevan, and L. Steinberg. LEAP: a learning apprentice for VLSI design. In *Proceedings of IJCAI-85*, Los Angeles, 1985.

David Jack Mostow. Machine transformation of advice into a heuristic search procedure. In *Machine Learning: An Artificial Intelligence Approach*, pages 367–404, Tioga, Palo Alto, CA, 1983.

Michael D. Rychener. The Instructible Production System: a retrospective analysis. In *Machine Learning: An Artificial Intelligence Approach*, pages 429–460, Tioga, Palo Alto, CA, 1983.

CHAPTER 15

Soar: An Architecture for General Intelligence

J. E. Laird, Xerox Palo Alto Research Center, A. Newell, Carnegie Mellon University and P. S. Rosenbloom, Stanford University*

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.

* Present address: Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109, U.S.A.

Introduction

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 OPS and LISP), initially as SOAR1 [31, 32], then as SOAR2 [29, 35], and currently as SOAR4 [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 played 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 (Fig. 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 knowledge. The main effort has been R1-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's 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.

FIG. 1. Summary of SOAR performance scope.

[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.

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 the 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, 57–59, 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 2 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

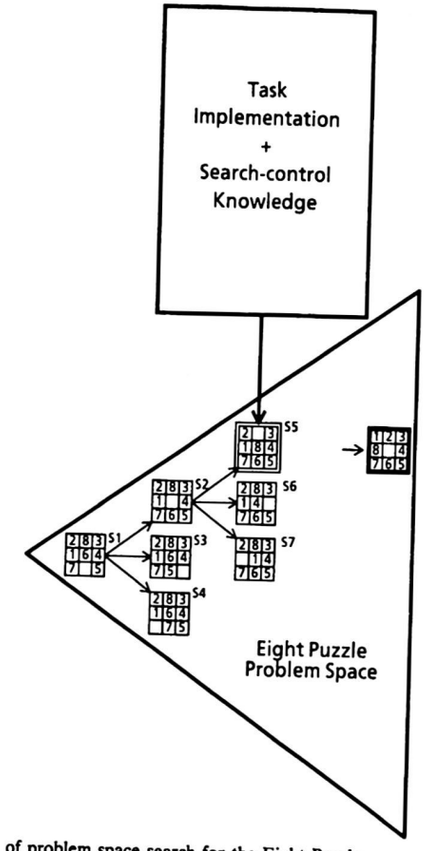

FIG. 2. The structure of problem space search for the Eight Puzzle.

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 Fig. 2, 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. 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 Fig. 2 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 (Fig. 3). 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 Figs. 2 and 3 are filled in with a uniform production system. Productions deliver control knowledge, 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.

SOAR employs a specialized production system (a modified version of OPS5 [20]). All satisfied productions are fired in parallel, without conflict resolution.

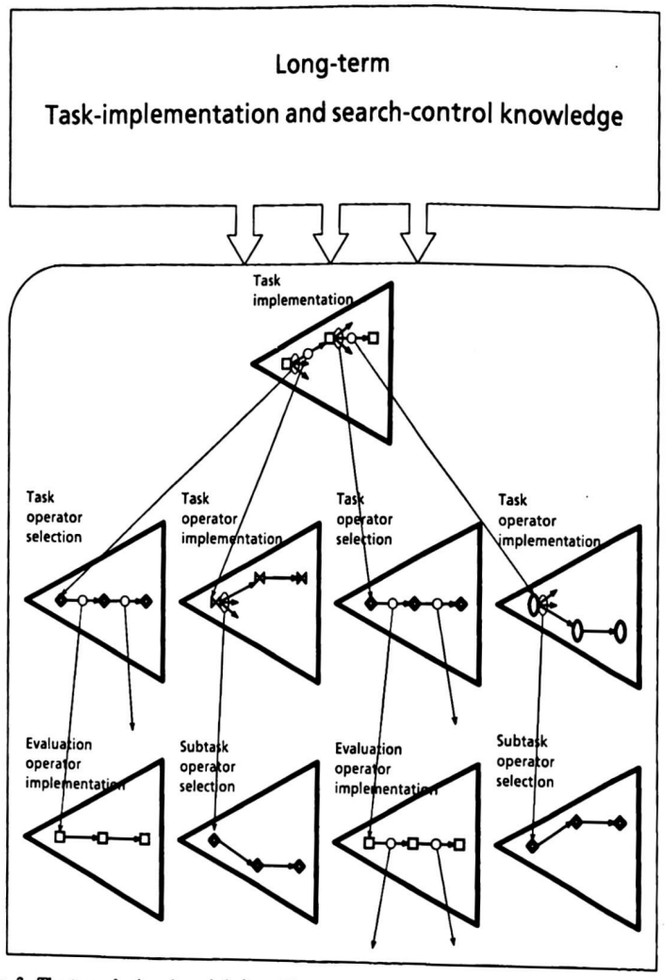

FIG. 3. The tree of subgoals and their problem spaces.

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 Fig. 3 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 Fig. 3, 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 Fig. 4 shows, the behavior of SOAR on the Eight Puzzle can be described as a sequence of such acts. Other important functions

```
[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
...
```

FIG. 4. Problem space trace in the Eight Puzzle. (Task-implementation steps are bracketed.)

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 Fig. 2, 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 5 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

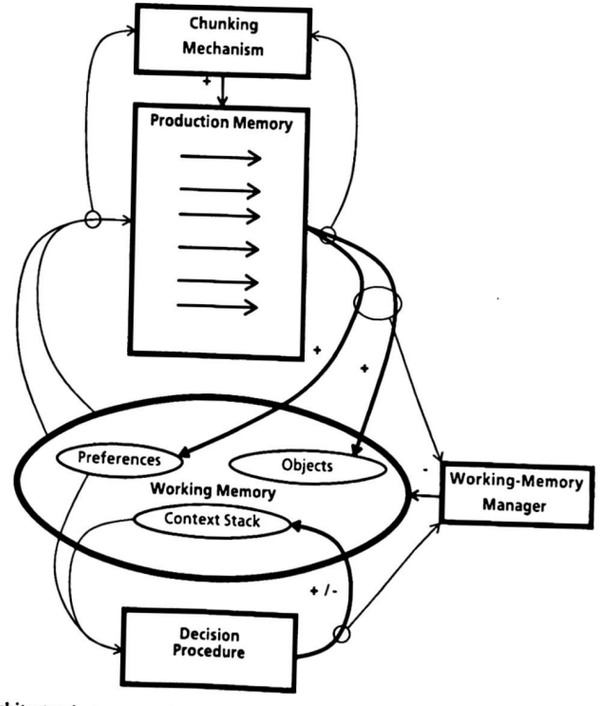

FIG. 5. Architectural structure of SOAR.

context stack that specifies the hierarchy of active goals, problem spaces, states and 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 of 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 COMMON-LISP, 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).

2.2. The working memory

Working memory consists of a *context stack*, a set of *objects* linked to the context stack, and *preferences*. Figure 6 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.

The basic representation is object-centered. An object, such as a goal or a

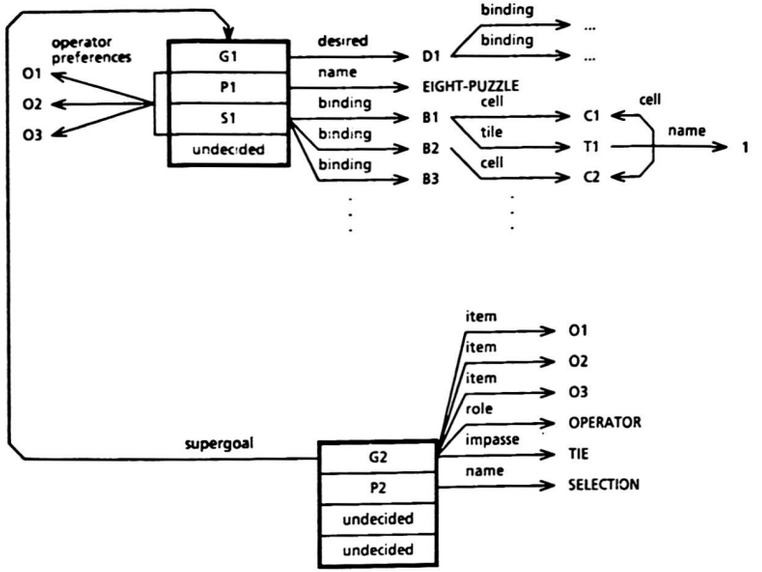

FIG. 6. Snapshot of fragment of working memory.

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.[1]

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 Fig. 10), only the

[1] 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.

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 Fig. 7.[2] 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 Fig. 7 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 Fig. 7. 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 Fig. 7, yielding a simple representation of sets.

```
(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)
```

FIG. 7. Working-memory representation of the structure in Fig. 6.

[2] Some basic notation and structure is inherited from OPS5.

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 8 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 lowest left-hand corner. The board in the Eight Puzzle is represented by nine *cells* (the 3×3 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 horizontal sequence of objects above the

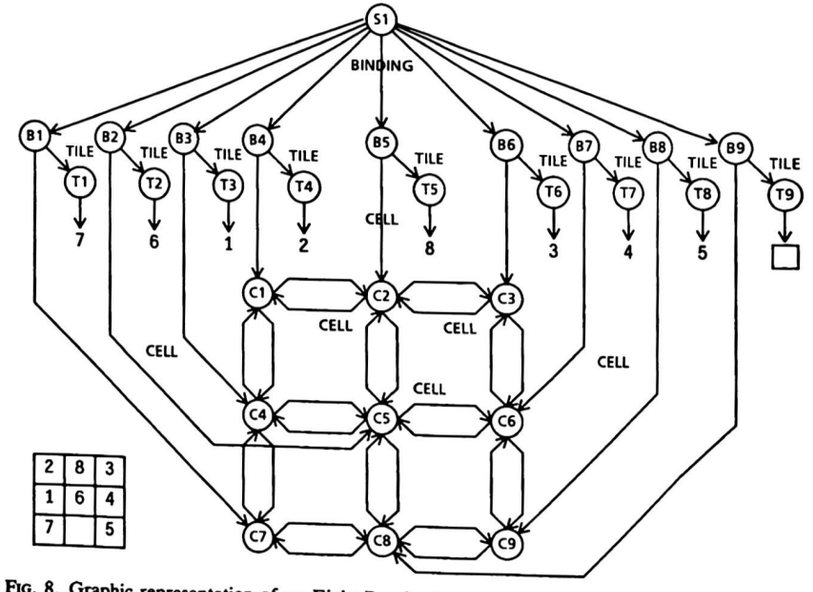

FIG. 8. Graphic representation of an Eight Puzzle state.

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.

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 9 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.

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.
- $(x \gg y)$: x dominates y: $(x > y)$ and $\neg(y > x)$.

Reference anchors:
- indifferent(x) $\Rightarrow$ $\forall y$ [indifferent(y) $\Rightarrow$ $(x \sim y)$].
- best(x) $\Rightarrow$ $\forall y$ [best(y) $\Rightarrow$ $(x \sim y)] \wedge [\neg$best(y) $\wedge \neg(y > x) \Rightarrow (x > y)]$.
- worst(x) $\Rightarrow$ $\forall y$ [worst(y) $\Rightarrow$ $(x \sim y)] \wedge [\neg$worst(y) $\wedge \neg(y < x) \Rightarrow (x < y)]$.

Basic properties:
- Desirability $(x > y)$ is transitive, but *not* complete or antisymmetric.
- Indifference is an equivalence relationship and substitutes over $>$
  - $(x > y)$ and $(y \sim z)$ implies $(x > z)$.
- Indifference does *not* substitute in acceptable, reject, best, and worst.
  - acceptable(x) *and* $(x \sim y)$ does *not* imply acceptable(y),
  - reject(x) and $(x \sim y)$ does *not* 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:
- considered-choices = $\{x \in$ objects $\mid$ acceptable(x) $\wedge \neg$reject(x)$\}$.
- maximal(X) = $\{x \in X \mid \forall y \neg(y \gg x)\}$.
- maximal-choices = maximal(considered-choices).
- empty(X) = $\neg\exists x \in X$.
- 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:
- empty(maximal-choices) $\wedge \neg$reject(current) $\Rightarrow$ final-choice(current).
- mutually-indifferent(maximal-choices) $\wedge \neg$empty(maximal-choices) $\Rightarrow$ final-choice(select(maximal-choices)).

Impasse:
- empty(maximal-choices) $\wedge$ reject(current) $\Rightarrow$ impasse.
- $\neg$mutually-indifferent(maximal-choices) $\Rightarrow$ impasse(maximal-choices).

FIG. 11. The semantics of preferences.

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 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.[5] 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 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

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

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*[6] 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 Fig. 11 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 12–14 list the productions that encode the knowledge to implement the Eight Puzzle task.[7] Figure 12 contains the productions that set 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-space is

[6] Term was first used in this sense in repair theory [8]: we had originally used the term difficulty [29].

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

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.

FIG. 12. Productions that set up the Eight Puzzle.

sensitive to the name of the goal and suggests eight-puzzle as the 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.

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 Fig. 13. 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

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 25 provides a demonstration of the basic effects of chunking, using the Eight Puzzle [33]. The left-hand column (no learning) shows 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 Figs. 17 and 18, 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 25 shows only

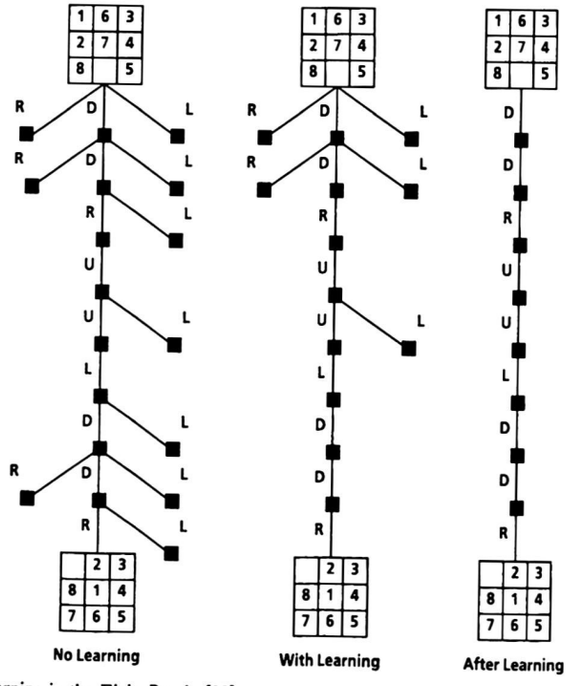

FIG. 25. Learning in the Eight Puzzle [33].

the moves made in the 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 state as in the run without learning (R, D and L 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.

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 26 shows *across-task transfer* in the Eight Puzzle. The first column (task 1, no learning) is the same trace as the first column in Fig. 25. 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 effects, 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.

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

Task	Pass 1 Before (learn off)	Pass 2 During (learn on)	Pass 3 After (learn off)	Pass 4 During (learn on)	Pass 5 After (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

FIG. 28. Performance of the extended version of R1-SOAR (with bottom-up learning) [75].

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 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 *variabilization* (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 29 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 crosses). 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

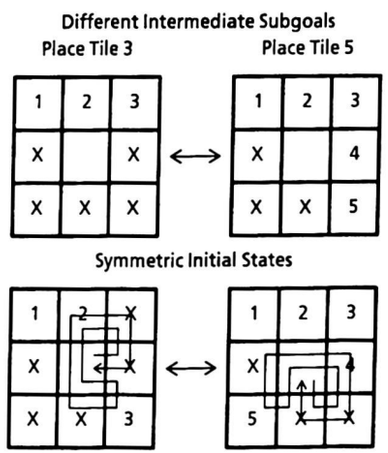

FIG. 29. Transfer possible with macro-operators in the Eight Puzzle.

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.

4. Conclusion

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

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

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

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

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

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

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

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

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

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

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

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.

Appendix A. Weak Methods as Patterns of Behavior

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: Make all moves at and-state and select 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.

Look-ahead: 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 look-ahead 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.

ACKNOWLEDGMENT

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.

REFERENCES

1. Amarel, S., On the representation of problems of reasoning about actions, in: D. Michie (ed.), *Machine Intelligence* 3 (American Elsevier, New York, 1968) 131–171.
2. Anderson, J.R., *The Architecture of Cognition* (Harvard University Press, Cambridge, MA, 1983).
3. Bachant, J. and McDermott, J., R1 revisited: Four years in the trenches, *AI Magazine* 5 (1984).
4. Balzer, R., Erman, L.D., London, R. and Williams, C., HEARSAY-III: A domain-independent framework for expert systems, in: *Proceedings AAAI-80*, Stanford, CA, 1980.
5. Berliner, H.J., The B* tree search algorithm: A best-first proof procedure, *Artificial Intelligence* 12 (1979) 23–40.
6. Boggs, M. and Carbonell, J., A tutorial introduction to DYPAR-1, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA.
7. Bower, G.H. and Winzenz, D., Group structure, coding and memory for digit series, *J. Experimental Psychol. Monograph* 80 (1969) 1–17.
8. Brown, J.S. and VanLehn, K., Repair theory: A generative theory of bugs in procedural skills, *Cognitive Sci.* 4 (1980) 379–426.
9. Buchanan, B.G. and 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: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga, Palo Alto, CA, 1983).
11. Card, S.K., Moran, T.P. and Newell, A., Computer text editing: An information-processing analysis of a routine cognitive skill, *Cognitive Psychol.* 12 (1) (1980) 32–74.
12. Chase, W.G. and Simon, H.A., Perception in chess, *Cognitive Psychol.* 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. and Mooney, R., Explanation-based learning: An alternative view, *Machine Learning* 1 (2) (1986) 145–176.
16. Erman, L., Hayes-Roth, F., Lesser, V. and Reddy, D.R., The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty, *Comput. Surveys* 12 (1980) 213–253.
17. Ernst, G.W. and Newell, A., *GPS: A Case Study in Generality and Problem Solving* (Academic Press, New York, 1969).
18. Feigenbaum, E.A. and Feldman, J. (Eds.), *Computers and Thought* (McGraw-Hill, New York, 1963).
19. Fikes, R.E., Hart, P.E. and 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, Pittsburgh, PA, 1981.
21. Forgy, C.L. and McDermott, J., OPS, a domain-independent production system language, in: *Proceedings IJCAI-77*, Cambridge MA, 1977.
22. Genesereth, M., An overview of meta-level architecture, in: *Proceedings AAAI-83*, Washington, DC, 1983.
23. Hayes, J.R. and 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. and Lenat, D.B. (Eds.), *Building Expert Systems* (Addison-Wesley, Reading, MA, 1983).
26. Kant, E. and Newell, A., An automatic algorithm designer: An initial implementation, in: *Proceedings AAAI-83*, Washington, DC, 1983.
27. Korf, R.E., Toward 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. Thesis, Carnegie-Mellon University, Pittsburgh, PA, 1984.
30. Laird, J.E., **Soar user's manual**: Version 4.0, Xerox Palo Alto Research Center, Palo Alto, CA, 1986.
31. Laird, J. and Newell, A., A universal weak method, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1983.
32. Laird, J. and Newell, A., A universal weak method: Summary of results, in: *Proceedings IJCAI-83*, Karlsruhe, F.R.G., 1983.
33. Laird, J.E., Rosenbloom, P.S. and Newell, A., Towards chunking as a general learning mechanism, in: *Proceedings AAAI-84*, Austin, TX, 1984.
34. Laird, J.E., Rosenbloom, P.S. and Newell, A., Overgeneralization during knowledge compilation in Soar, in: *Proceedings Workshop on Knowledge Compilation*, Otter Crest, OR, 1986.
35. Laird, J.E., Rosenbloom, P.S. and 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. and 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 Sci.* **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* **21** (1983) 61–98.
39. Lenat, D.B. and Brown, J.S., Why AM and EURISKO appear to work, *Artificial Intelligence* **23** (1984) 269–294.
40. McDermott, D., Planning and acting, *Cognitive Sci.* **2** (1978) 71–109.
41. McDermott, J., R1: A rule-based configurer of computer systems, *Artificial Intelligence* **19** (1982) 39–88.
42. McDermott, J. and Forgy, C.L., Production system conflict resolution strategies, in: D.A. Waterman and F. Hayes-Roth (Eds.), *Pattern Directed Inference Systems* (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, *Psychol. Rev.* **63** (1956) 81–97.
44. Mitchell, T.M., Version spaces: An approach to concept learning, Ph.D. Thesis, Stanford University, Stanford, CA, 1978.
45. Mitchell, T.M., Utgoff, P.E. and Banerji, R., Learning by experimentation: Acquiring and refining problem-solving heuristics, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga, Palo Alto, CA, 1983).
46. Moston, D.J., Machine transformation of advice into a heuristic search procedure, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga, Palo Alto, CA, 1983).
47. Newell, A., Heuristic programming: Ill-structured problems, in: J. Aronofsky (Ed.), *Progress in Operations Research III* (Wiley, New York, 1969) 360–414.
48. Newell, A., Production systems: Models of control structures, in: W.C. Chase (Ed.), *Visual Information Processing* (Academic Press, New York, 1973) 463–526.

49. Newell, A. Reasoning, problem solving and decision processes: The problem space as a fundamental category, in: R. Nickerson (Ed.), *Attention and Performance* **VIII** (Erlbaum, Hillsdale, NJ, 1980).
50. Newell, A. and Rosenbloom, P., Mechanisms of skill acquisition and the law of practice, in: J.A. Anderson (Ed.), *Learning and Cognition* (Erlbaum, Hillsdale, NJ, 1981).
51. Newell, A. and Simon, H.A., *Human Problem Solving* (Prentice-Hall, Englewood Cliffs, NJ, 1972).
52. Newell, A. and Simon, H.A., Computer science as empirical inquiry: Symbols and search, *Commun. ACM* **19** (3) (1976) 113–126.
53. Newell, A., Shaw, J.C. and Simon, H.A., Empirical explorations of the logic theory machine: A case study in heuristics, in: *Proceedings (1957 Western Joint Computer Conference* (1957) 218–230; also in: E.A. Feigenbaum and J. Feldman (Eds.), *Computers and Thought* (McGraw-Hill, New York, 1963).
54. Newell, A., Shaw, J. C. and 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) 256–264.
55. Newell, A., Tonge, F.M., Feigenbaum, E.A., Green, B. and Mealy, G., *Information Processing Language V Manual* (Prentice-Hall, Englewood Cliffs, NJ, 2nd ed., 1964).
56. Nii, H.P. and Aiello, N., AGE (Attempt to Generalize): A knowledge-based program for building knowledge-based programs, in: *Proceedings IJCAI-79*, Tokyo, Japan, 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, *J ACM* **12** (1965) 23–41.
61. Rosenbloom, P.S., The chunking of goal hierarchies: A model of practice and stimulus-response compatibility, Ph.D. Thesis, Tech. Rept. #83-148, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1983.
62. Rosenbloom, P.S. and Laird, J.E., Mapping explanation-based generalization onto Soar, in: *Proceedings AAAI-86*, Philadelphia, PA, 1986.
63. Rosenbloom, P.S. and Newell, A., The chunking of goal hierarchies: A generalized model of practice, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Morgan-Kaufmann, Los Altos, CA, 1986).
64. Rosenbloom, P.S., Laird, J.E. and Newell, A., Meta-levels in Soar, in: *Preprints of the Workshop on Meta-level Architectures and Reflection*, Sardinia, 1986.
65. Rosenbloom, P.S., Laird, J.E., McDermott, J., Newell, A. and Orciuch, E., R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture, *IEEE Trans. Patt. Anal. Mach. Intell.* **7** (5) (1985) 561–569.
66. Rychener, M.D., Production systems as a programming language for artificial intelligence applications, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1976.
67. Rychener, M.D., The instructable production system: A retrospective analysis, in: R.S. Michalski, J.G. Carbonell and T.M. Mitchell (Eds.), *Machine Learning: An Artificial Intelligence Approach* (Tioga, Palo Alto, CA, 1983).
68. Rychener, M.D. and Newell, A., An instructable production system: Basic design issues, in: D.A. Waterman and F. Hayes-Roth (Eds.), *Pattern Directed Inference Systems* (Academic Press, New York, 1978) 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, Stanford, CA, 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) 7–29.
73. Smith, B.C., Reflection and semantics in a procedural language, MIT/LCS/TR-272, Laboratory for Computer Science, MIT, Cambridge, MA, 1982.
74. Smith, D.E. and Genesereth, M.R., Ordering conjunctive queries, *Artificial Intelligence* 26 (1985) 171–215.
75. van de Brug, A., Rosenbloom, P.S. and Newell, A., Some experiments with R1-Soar, Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA, 1986.
76. van de Brug, A., Bachant, J. and 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 Research Center, Palo Alto, CA, 1983.
78. Waterman, D.A. and 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).

Received September 1983 with the title "A Universal Weak Method"; revised version received November 1986

CHAPTER 16

Knowledge Level Learning in Soar

P. S. Rosenbloom, Stanford University, J. E. Laird, University of Michigan, and A. Newell, Carnegie Mellon University

Abstract

In this article we demonstrate how knowledge level learning can be performed within the Soar architecture. That is, we demonstrate how Soar can acquire new knowledge that is not deductively implied by its existing knowledge. This demonstration employs Soar's chunking mechanism – a mechanism which acquires new productions from goal-based experience – as its only learning mechanism. Chunking has previously been demonstrated to be a useful symbol level learning mechanism, able to speed up the performance of existing systems, but this is the first demonstration of its ability to perform knowledge level learning. Two simple declarative-memory tasks are employed for this demonstration: recognition and recall.

I. Introduction

Dietterich has recently divided learning systems into two classes: *symbol level learners* and *knowledge level learners* [3]. The distinction is based on whether or not the knowledge in the system, as measured by a knowledge level analysis [10], increases with learning. A system performs symbol level learning if it improves its computational performance but does not increase the amount of knowledge it contains. According to a knowledge level analysis, knowledge only increases if a fact is added that is not implied by the existing knowledge; that is, if the fact is not in the deductive closure of the existing knowledge. Explanation-based generalization (EBG) [2; 9] is a prime example of a learning technique that has proven quite successful as a mechanism for enabling a system to perform symbol level learning. EBG allows tasks that a system can already perform to be reformulated in such a way that they can be performed more efficiently. Because EBG only generates knowledge that is already within the deductive closure of its current knowledge base, it does no knowledge level learning (at least when used in any obvious ways).

Symbol level learning can be quite useful for an intelligent system. By speeding up the system's performance, it allows the system to perform more tasks while using the same amount of resources, and enables the system to complete tasks that capacity limitations previously prevented it from completing. However, an intelligent system cannot live by symbol level learning alone. If a system were incapable of performing knowledge level learning – that is, of adding facts not already implied by its existing knowledge – a host of critical capabilities would be beyond its grasp. These range from relatively simple declarative memory capabilities, such as learning to recognize previously seen objects and to store and retrieve representations of new objects, to more complex capabilities of learning to perform novel tasks.

Soar is an attempt to build an architecture that can support general intelligent behavior [6]. It contains a single learning mechanism, *chunking* [7]. Chunking creates new productions, or chunks, based on the results of goal-based problem solving. The actions of a chunk contain the results of the goal. The conditions of the chunk test those aspects of the pre-goal situation that were relevant to the generation of the results. We have been successful in demonstrating Soar's ability to acquire a variety of types of knowledge, including search-control productions, macro-operators, and operator-implementation productions [14]. We have also demonstrated Soar's ability to perform the basic tasks of EBG and, in the process, provided a mapping between EBG and Soar [12]. This mapping suggests that chunking is a symbol level learning mechanism; an identification that is supported by the fact that all of the demonstrations listed above involve only symbol level learning.

However, chunking originated in psychological theories of the structure of declarative memory [8]. One classical chunking result is that if a list of items is to be memorized for later recall, the memory structure is organized as a hierarchical structure of chunks [1]. Chunking, at least of this classical variety, is thus strongly implicated in the acquisition of new knowledge. This has encouraged us to believe that Soar's chunking mechanism should be able to perform knowledge level learning. It has also placed a responsibility on us to demonstrate these classical chunking phenomena in Soar in order to justify that its learning mechanism is entitled to its name. To distinguish between the form of chunking currently performed by Soar and the more declarative classical chunking capability, we refer to the classical form of chunking as *data chunking*.

The purpose of this article is to report on recent work with Soar that demonstrates data chunking, and thus knowledge level learning, using chunking as the only learning mechanism. In Section II we give a brief overview of Soar. In Section III we describe the fundamental concepts underlying the implementation of data chunking in Soar. In Sections IV and V we then describe in detail how Soar performs two simple data chunking tasks: learning to recognize new objects and learning to recall new objects. In Section VI we conclude

[1]This research was sponsored by the Defense Advanced Research Projects Agency (DOD) under contract N00039-86-C-0133 and by the Sloan Foundation. Computer facilities were partially provided by NIH grant RR-00785 to Sumex-Aim. 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 US Government, the Sloan Foundation, or the National Institutes of Health.