Two *Soar* Studies

Towards Chunking as a General Learning Mechanism

John E. Laird, Paul S. Rosenbloom and Allen Newell

R1-Soar: An Experiment in Knowledge-Intensive Programming in a Problem-Solving Architecture

Paul S. Rosenbloom, John E. Laird, John McDermott,
Allen Newell & Edmund Orciuch

January 1985

Two *Soar* Studies

The ***Soar*** project is attempting to build a system capable of general intelligent behavior. We seek to understand what mechanisms are necessary for intelligent behavior and how they work together to form a general cognitive architecture. Our knowledge about this enterprise is reflected in the evolving assumptions embedded in the ***Soar*** architecture (Laird, 1985). The body of this report consists of a pair of papers (lightly edited from their original published form) reporting on investigations with ***Soar*** into two components of general intelligence: learning, and performance in knowledge-intensive tasks.

The first paper is titled *Towards Chunking as a General Learning Mechanism* (Laird, Rosenbloom, & Newell, 1984). Chunks have long been proposed as a basic organizational unit for human memory. More recently chunks have been used to model human learning on simple perceptual-motor skills. In this paper we describe recent progress in extending chunking to be a general learning mechanism by implementing it within ***Soar***. By implementing chunking within a general-problem solving architecture we take significant steps toward a general problem solver that can learn about all aspects of its behavior. We demonstrate chunking in ***Soar*** on three tasks: the Eight Puzzle, Tic-Tac-Toe, and a part of the ***R1*** computer-configuration task. Not only is there improvement with practice, but chunking also produces significant transfer of learned behavior, and strategy acquisition.

The second paper, titled *R1-Soar: An Experiment in Knowledge-Intensive Programming in a Problem-Solving Architecture* (Rosenbloom, Laird, McDermott, Newell, & Orciuch, 1984), presents an experiment in knowledge-intensive programming in ***Soar***. In ***Soar***, knowledge is encoded within a set of problem spaces, yielding a system capable of reasoning from first principles. Expertise consists of additional rules that guide complex problem-space searches and substitute for expensive problem-space operators. The resulting system uses both knowledge and search when relevant. Expertise knowledge is acquired either by having it programmed, or by a chunking mechanism that automatically learns new rules reflecting the results implicit in the knowledge of the problem spaces. The approach is demonstrated on the computer-system configuration task, the task performed by the expert system, ***R1***.

References

Laird, J. E. *Soar User's Manual* (Tech. Rep.). , 1985. In preparation.

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

Rosenbloom, P. S., Laird, J. E., McDermott, J., Newell, A., & Orciuch, E. R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture. In *Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems*. IEEE Computer Society, 1984.

Towards Chunking as a General Learning Mechanism[1]

John E. Laird, Paul S. Rosenbloom and Allen Newell[2]

Chunking was first proposed as a model of human memory by Miller (Miller, 1956), and has since become a major component of theories of cognition. More recently it has been proposed that a theory of human learning based on chunking could model the ubiquitous power law of practice (Newell and Rosenbloom, 1981). In demonstrating that a practice mechanism based on chunking is capable of speeding up task performance, it was speculated that chunking, when combined with a general problem solver, might be capable of more interesting forms of learning than just simple speed ups (Rosenbloom & Newell, 1983). In this paper we describe an initial investigation into chunking as a general learning mechanism.

Our approach to developing a general learning mechanism is based on the hypothesis that all complex behavior — which includes behavior concerned with learning — occurs as search in problem spaces (Newell, 1980). One image of a system meeting this requirement consists of the combination of a performance system based on search in problem spaces, and a complex, analytical, learning system also based on search in problem spaces (Mitchell, 1983). An alternative, and the one we adopt here, is to propose that all complex behavior occurs in the problem-space-based performance system. The learning component is simply a recorder of experience. It is the experience that determines the form of what is learned.

Chunking is well suited to be such a learning mechanism because it is a recorder of goal-based experience (Rosenbloom, 1983; Rosenbloom & Newell, 1983). It caches the processing of a subgoal in such a way that a chunk can substitute for the normal (possibly complex) processing of the subgoal the next time the same subgoal (or a suitably similar one) is generated. It is a task-independent mechanism that can be applied to all subgoals of any task in a system. Chunks are created during performance, through experience with the goals processed. No extensive analysis is required either during or after performance.

---

[1]This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-k-1539. 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 or the US Government.

[2]Allen Newell is at the Carnegie-Mellon University Computer Science Department. Most of this work was done while Paul Rosenbloom and John Laird were also at CMU CSD. Paul Rosenbloom is now at the Stanford University Departments of Computer Science and Psychology. John Laird is now at the Xerox Palo Alto Research Center.

The essential step in turning chunking into a general learning mechanism is to combine it with a general problem-space problem solver. One candidate is ***Soar***, a reflective problem-solving architecture that has a uniform representation and can create goals to reason about any aspect of its problem-solving behavior (Laird, 1983). Implementing chunking within ***Soar*** yields four contributions towards chunking as a general learning mechanism.

1. Chunking can be applied to a general problem solver to speed up its performance.

2. Chunking can improve *all* aspects of a problem solver's behavior.

3. Significant transfer of chunked knowledge is possible via the implicit generalization of chunks.

4. Chunking can perform strategy acquisition, leading to qualitatively new behavior.

Other systems have tackled individual points, but this is the first attempt to do all of them. Other work on strategy acquisition deals with the learning of qualitatively new behavior (Langley, 1983; Mitchell, 1983), but it is limited to learning only one type of knowledge. These systems end up with the *wandering bottle-neck* problem — removal of a performance bottleneck from one part of a system means that some other locale becomes the bottleneck (Mitchell, 1983). Anderson (Anderson, 1983) has recently proposed a scheme of knowledge compilation to be a general learning mechanism to be applied to all of cognition, although it has not yet been used on complex problem solving or reasoning tasks that require learning about all aspects of behavior.

1. Soar — A General Problem-Solving Architecture

Soar is a problem solving system that is based on formulating all activity (both problems and routine tasks) as heuristic search in problem spaces. A problem space consists of a set of *states* and a set of *operators* that transform one state into another. Starting from an initial state the problem solver applies a sequence of operators in an attempt to reach a desired state. ***Soar*** uses a production system[3] to implement elementary operators, tests for goal satisfaction and failure, and *search control* — information relevant to the selection of goals, problem spaces, states, and operators. It is possible to use a problem space that has no search control, only operators and goal recognizers. Such a space will work correctly, but will be slow because of the amount of search required.

In many cases, the directly available knowledge may be insufficient for making a search-control

[3] A modified versions of ***Ops5*** (Forgy, 1981), which admits parallel execution of all satisfied productions.

decision or applying an operator to a state. When this happens, a *difficulty* occurs that results in the automatic creation of a subgoal to perform the necessary function. In the subgoal, ***Soar*** treats the difficulty as just another problem to solve; it selects a problem space for the subgoal in which goal attainment is interpreted as finding a state that resolves the difficulty. Thus, ***Soar*** generates a hierarchy of goals and problem spaces. The diversity of task domains is reflected in a diversity of problem spaces. Major tasks, such as configuring a computer will have a corresponding problem space, but so also will each of the various subtasks. In addition, problem spaces will exist in the hierarchy for performing tasks generated by problems in the system's own behavior, such as the selection of an operator to apply, the application of an operator to a state, and testing for goal attainment. With such an organization, all aspects of the system's behavior are open to problem solving when necessary. We call this property *universal subgoaling* (Laird, 1983).

Figure 1-1 shows a small example of how these subgoals are used in ***Soar***. This is the subgoal/problem-space structure that gets generated while trying to take steps in a task problem space. Initially (A), the problem solver is at State1 and must select an operator. If search control is unable to uniquely determine the next operator to apply, a subgoal is created to do the selection. In that subgoal (B), a *selection* problem space is used that reasons about the selection of objects from a set. In order to break the tie between objects, the selection problem space has operators to evaluate each candidate object.

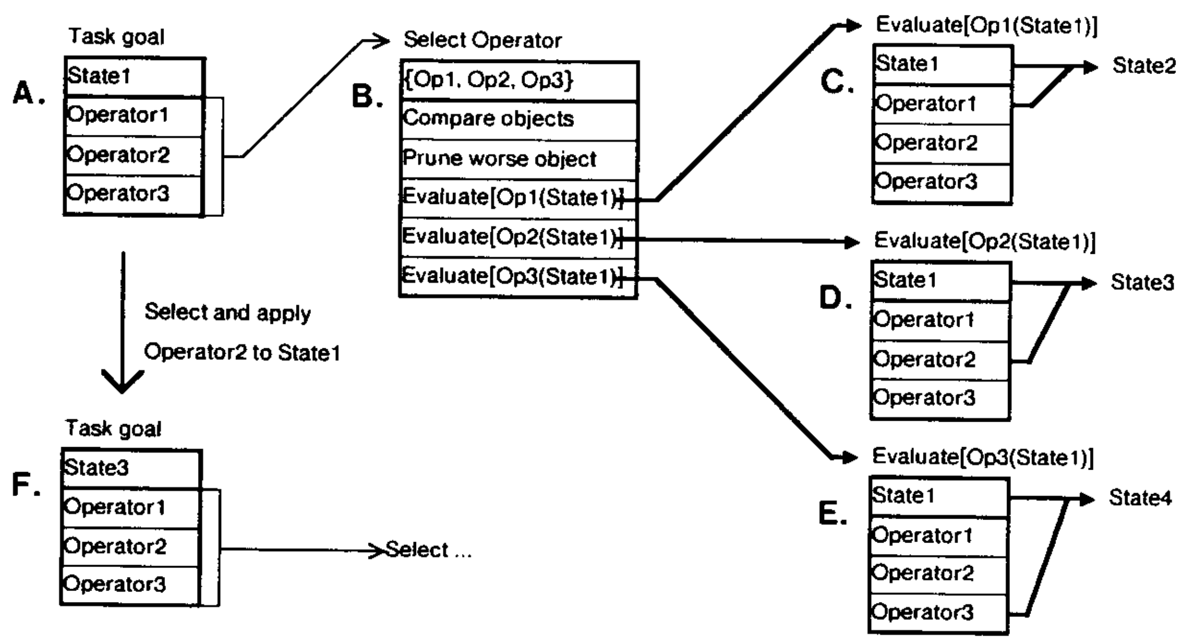

Figure 1-1: Eight Puzzle subgoal/problem space structure.

Evaluating an operator, such as Operator1 in the task space, is a complex problem requiring a new subgoal. In this subgoal (C), the original task problem space and state (State1) are selected. Operator1 is applied, creating a new state (State2). The evaluation for State2 is used to compare Operator1 to the other operators. When Operator1 has been evaluated, the subgoal terminates, and then the whole process is repeated for the other two operators (Operator2 and Operator3 in D and E). If, for example, Operator2 creates a state with a better evaluation than the other operators, it will be designated as better than them. The selection subgoal will terminate and the designation of Operator2 will lead to its selection in the original task goal and problem space. At this point Operator2 is reapplied to State1 and the process continues (F).

2. Chunking in Soar

Chunking was previously defined (Rosenbloom & Newell, 1983) as a process that acquired chunks that generate the results of a goal, given the goal and its parameters. The parameters of a goal were defined to be those aspects of the system existing prior to the goal's creation that were examined during the processing of the goal. Each chunk was represented as a set of three productions, one that encoded the parameters of a goal, one that connected this encoding in the presence of the goal to (chunked) results, and a third production that decoded the results. These chunks were learned bottom-up in the goal hierarchy; only terminal goals — goals for which there were no subgoals that had not already been chunked — were chunked. These chunks improved task performance by substituting efficient productions for complex goal processing. This mechanism was shown to work for a set of simple perceptual-motor skills based on fixed goal hierarchies (Rosenbloom, 1983).

At the moment, ***Soar*** does away with two of the features of chunking that existed for psychological modeling purposes: the three production chunks, and the the bottom-up nature of chunking. In ***Soar***, single-production chunks are built for every subgoal that terminates. The power of chunking in ***Soar*** stems from ***Soar***'s ability to automatically generate goals for problems in any aspect of its problem-solving behavior: a goal to select among alternatives leads to the creation of a production that will later control search; a goal to apply an operator to a state leads to the creation of a production that directly implements the operator; and a goal to test goal-satisfaction leads to a goal-recognition production. As search-control knowledge is added, performance improves via a reduction in the amount of search. If enough knowledge is added, there is no search; what is left is a *method* — an efficient algorithm for a task. In addition to reducing search within a single problem space, chunks can completely eliminate the search of entire subspaces whose function is to make a search-control decision, apply an operator, or recognize goal-satisfaction.

The conditions of a chunked production need to test everything that was used in creating the results of the subgoal and that existed before the subgoal was invoked. In standard problem solvers this would consist of the name of the goal and its parameters. However, in ***Soar*** there are no fixed goal names, nor is there a fixed set of parameters. Once a subgoal is selected, all of the information from the prior goal is still available. The problem solver makes use of the information about why the subgoal was created and any of the other information that it needs to solve the problem.

For each goal generated, the architecture maintains a *condition-list* of all data that existed before the goal was created and which was *accessed* in the goal. A datum is considered accessed if a production that matched it fires. Whenever a production is fired, all of the data it accessed that existed prior to the current goal are added to the goal's condition-list. When a goal terminates (for whatever reason), the condition-list for that goal is used to build the conditions of a chunk. Before being turned into conditions, the data is selectively variablized so that the conditions become tests for object descriptions instead of tests for the specific objects experienced. These variables are restricted so that two distinct variables can not match the same object.

The actions of the chunk should be the results of the goal. In traditional architectures, a goal produces a specific predefined type of result. However, in ***Soar***, anything produced in a subgoal can potentially be of use in the parent goal. Although the potential exists for all objects to be relevant, the reality is that only a few of them will actually be useful. In figuring out the actions of the chunk, ***Soar*** starts with everything created in the goal, but then prunes away the information that does not relate directly to objects in any supergoal.[4] What is left is turned into production actions after being variablized in accordance with the conditions.

At first glance, chunking appears to be simply a caching mechanism with little hope of producing results that can be used on other than exact duplicates of tasks it has already attempted. However, if a given task shares subgoals with another task, a chunk learned for one task can apply to the other, yielding *across-task* transfer of learning. *Within-trial* transfer of learning can occur when a subgoal arises more than once during a single attempt on a task. Generality is possible because a chunk only contains conditions for the aspects that were accessed in the subgoal. This is an *implicit generalization*, by which many aspects of the context — the irrelevant ones — are automatically ignored by the chunk.

---

[4] Those that are pruned are also removed from memory because they are intermediate results that will never be used again.

3. Demonstration

In this section we describe the results of experiments on three tasks: the Eight Puzzle, Tic-Tac-Toe, and computer configuration (a part of the ***R1*** expert-system implemented in ***Soar*** (Rosenbloom, Laird, McDermott, Newell, & Orciuch, 1984)). These tasks exhibit: (1) speed ups with practice; (2) within-trial transfer of learning; (3) across-task transfer of learning; (4) strategy acquisition (the learning of paths through search spaces); (5) knowledge acquisition in a knowledge-intensive system; and (6) learning of qualitatively different aspects of behavior. We conclude this section with a discussion of how chunking sometimes builds over-general productions.

3.1. Eight Puzzle

The states for the Eight Puzzle, as implemented in ***Soar***, consist of different configurations of eight numbered tiles in a three by three grid; the operators move the blank space up (U), down (D), left (L) and right (R) (Laird, 1983). Search-control knowledge was built that computed an evaluation of a state based on the number of tiles that were moved in and out of the desired positions from the previous state.[5] At each state in the problem solving, an operator must be selected, but there is insufficient search-control knowledge to intelligently distinguish between the alternatives. This leads to the selection being made using the set of selection and evaluation goals described in Section 1. The first column of Figure 3-1 shows the behavior of ***Soar*** without chunking in the Eight Puzzle problem space. All of the nodes off the main path were expanded in evaluate-operator subgoals (nodes on the main path were expanded once in a subgoal, and once after being selected in the top goal).[6]

When ***Soar*** with chunking is applied to the task, both the selection and evaluation subgoals are chunked. During this run (second column of Figure 3-1), some of the newly created chunks apply to subsequent subgoals in the search. This within-trial transfer of learning speeds up performance by dramatically reducing the amount of search. The third column in the figure shows that after one run with learning, the chunked productions completely eliminate search.

To investigate across-task learning, another experiment was conducted in which ***Soar*** started with a learning trial for a different task — the initial and final states are different, and none of the

[5] To avoid tight loops, search-control was also added that avoided applying the inverse of the operator that created a given state.

[6] At two points in the search the correct operator had to be selected manually because the evaluation function was insufficient to pick out the best operator. Our purpose is not to evaluate the evaluation function, but to investigate how chunking can be used in conjunction with search-control knowledge.

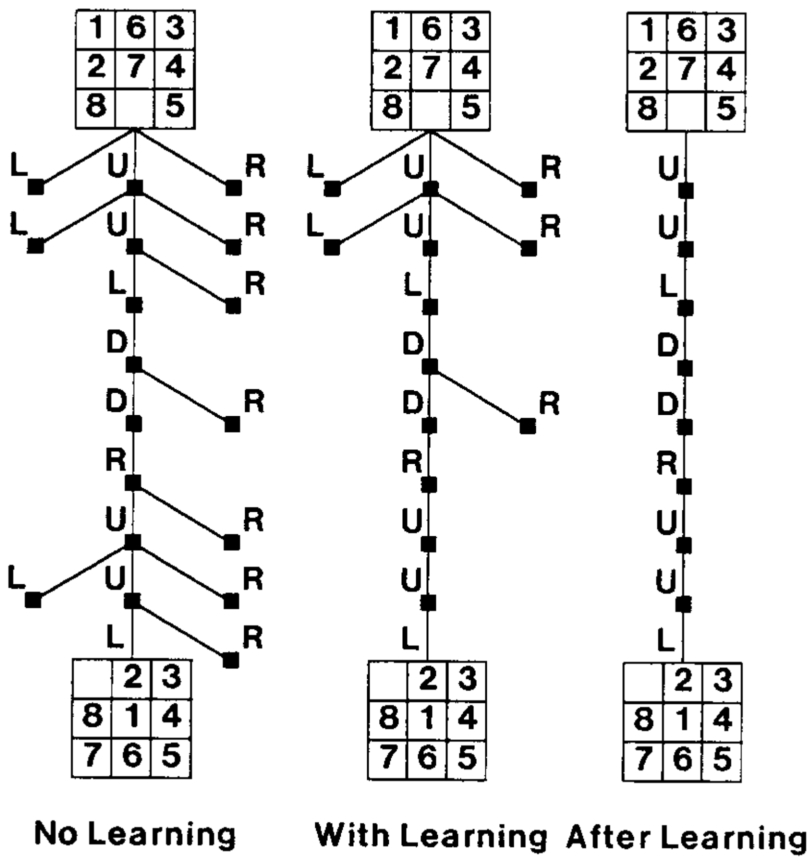

Figure 3-1: Within-trial transfer and speed-up with practice in the Eight Puzzle.

intermediate states were the same (the second column in Figure 3-2). The first task was then attempted with the productions learned from the second task, but with chunking turned off so that there would be no additional learning (the third column). The reduced search is caused by across-task transfer of learning — some subgoals in the second trial were identical in all of the relevant ways to subgoals in the first trial. This happens because of the interaction between the problem solving only accessing information relevant to the result, and the implicit generalization of chunking only recording the information accessed.

3.2. Tic-Tac-Toe

The implementation of Tic-Tac-Toe includes only the basic problem space — the state includes the board and who is on move, the operators make a mark on the board for the appropriate player and change who is on move — and the ability to detect a win, loss or draw (Laird, 1983). With just this knowledge, ***Soar*** searches depth-first through the problem space by the sequence of: (1) encountering a difficulty in selecting an operator; (2) evaluating the operators in a selection subgoal;

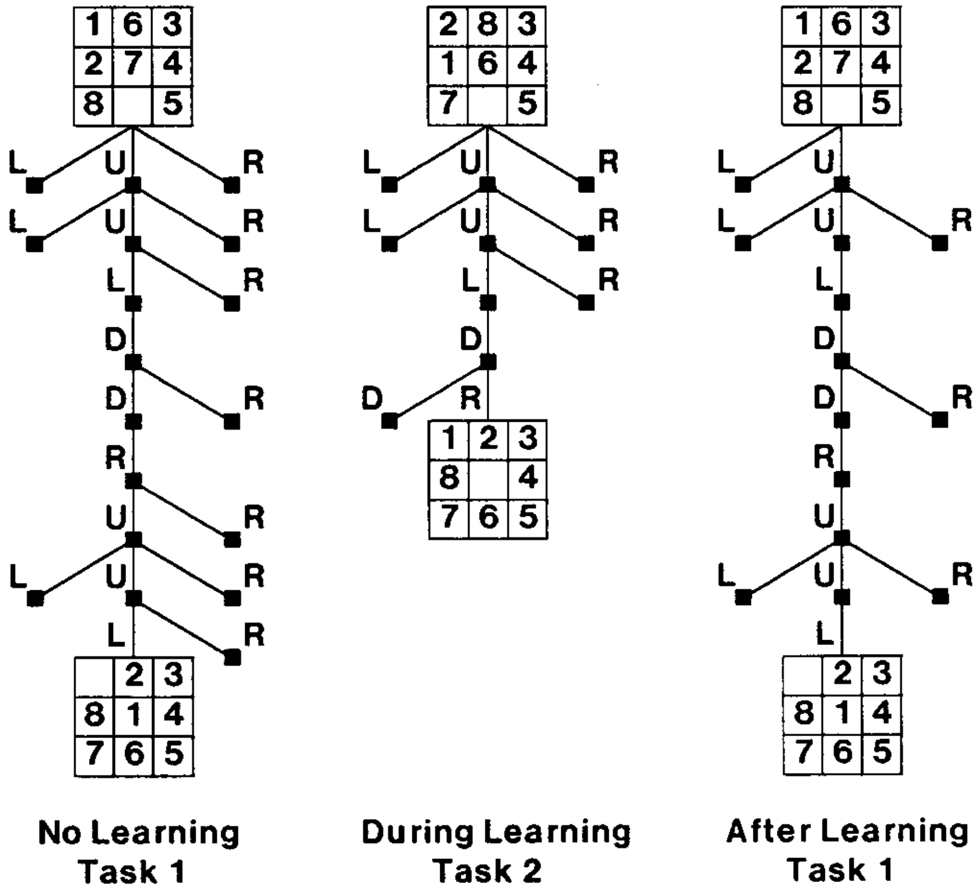

Figure 3-2: Across-task transfer in the Eight Puzzle.

(3) applying one of the operators in an evaluation subgoal; (4) encountering a difficulty in selecting an operator to apply to the resulting state; and (5) so on, until a terminal state is reached and evaluated.

Chunking in Tic-Tac-Toe yields two interesting results: (1) the chunks detect board symmetries, allowing a drastic reduction in search through within-trial transfer, (2) the chunks encode search-control knowledge so that the correct moves through the space are remembered. The first result is interesting because there is no knowledge in the system about the existence of symmetries, and without chunking the search bogs down terribly by re-exploring symmetric positions. The chunks make use of symmetries by ignoring orientation information that was not used during problem solving. The second point seems obvious given our presentation of chunking, however, it demonstrates the *strategy acquisition* (Langley, 1983; Mitchell, 1983) abilities of chunking. Chunking acquires strategic information on the fly, using only its direct experience, and without complex post-processing of the complete solution path or knowledge learned from other trials. The quality of this path depends on the quality of the problem solving, not on the learning.

3.3. R1

Part of the ***R1*** expert system (McDermott, 1982) was implemented in ***Soar*** to investigate whether ***Soar*** can support knowledge-intensive expert systems (Rosenbloom, Laird, McDermott, Newell, & Orciuch, 1984). Figure 3-3 shows the subgoal structure that can be built up through universal subgoaling, including both subgoals that implement complex operators (heavy lines) and subgoals that select operators (thin lines to Selection subgoals). Each box shows the problem-space operators used in the subgoal. The actual subgoal structure extends much further wherever there is an ellipsis (...). This subgoal structure does not pre-exist in ***Soar***, but is built up as difficulties arise in selecting and applying operators.

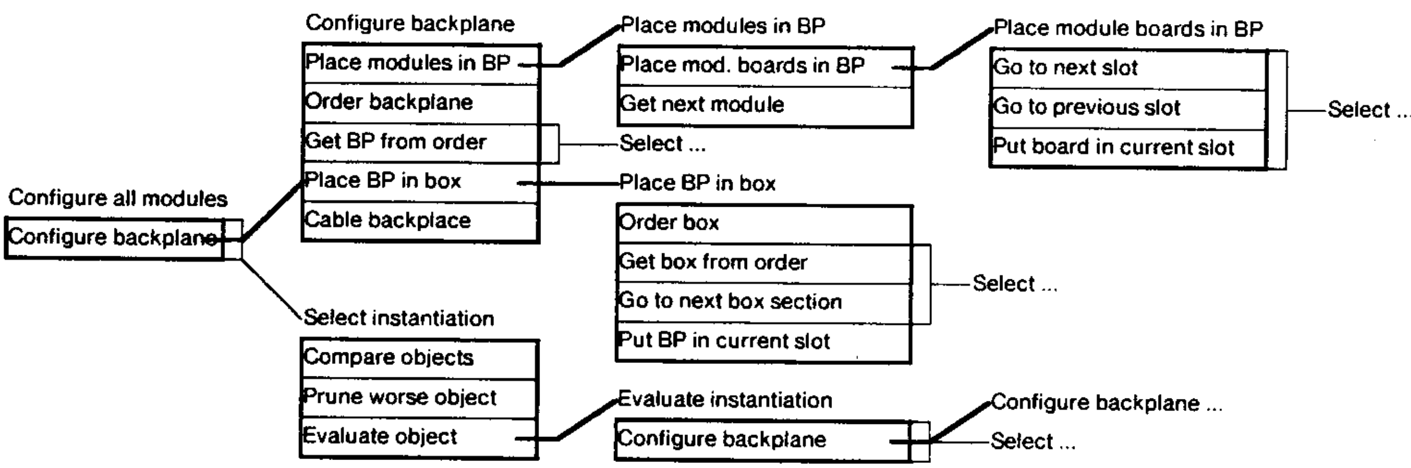

Figure 3-3: Subgoal Structure in ***R1-Soar***.

Table 3-1 presents statistics from the application of ***R1-Soar*** to a small configuration task. The first three runs (Min. S-C) are with a minimal system that has only the problem spaces and goal detection defined. This base system consists of 232 productions (95 productions come with ***Soar***, 137 define ***R1-Soar***). The final three runs (Added S-C) have 10 additional search-control productions that remove much of the search. In the table, the number of search-control decisions is used as the time metric because decisions are the basic unit of problem-solving.[7]

The first run shows that with minimal search control, 1731 decisions are needed to do the task. If chunking is used, 59 productions are built during the 485 decisions it took to do this task. No prior chunking had occurred, so this shows strong within-trial transfer. After chunking, rerunning the same

[7] On a Symbolics 3600, ***Soar*** usually runs at 1 second per decision. Chunking adds an overhead of approximately 15%, mostly to compile new productions. The increased number of productions has no affect on the overall rate if the chunked productions are fully integrated into the existing production-match network.

Run Type	Initial Prod.	Final Prod.	Decisions
Min. S-C	232	232	1731
Min. S-C with chunking	232	291	485
Min. S-C after chunking	291	291	7
Added S-C	242	242	150
Added S-C with chunking	242	254	90
Added S-C after chunking	254	254	7

Table 3-1: Run Statistics for ***R1-Soar***.

task takes only 7 decisions.

When ***Soar*** is run with 10 hand-crafted search-control rules, it only takes 150 decisions. This is only little more than three times faster than ***Soar*** without those rules took when chunking was used. When chunking is applied to this situation — where the additional search control already exists — it still helps by decreasing to 90 the number of decisions for the first trial. A second trial on this task once again takes only 7 decisions.

3.4. Over-generalization

The within-trial and across-task transfer in the tasks we have examined was possible because of implicit generalization. Unfortunately, implicit generalization leads to over-generalization when there is special-case knowledge that was *almost* used in solving a subgoal. In ***Soar*** this would be a production for which most but not all of the conditions were satisfied during a problem-solving episode. Those conditions that were not satisfied, either tested for the absence of something that is available in the subgoal (using a negated condition) or for the presence of something missing in the subgoal (using a positive condition) . The chunk that is built for the subgoal is over-general because it does not include the *inverses* of these conditions — negated conditions for positive conditions, and positive conditions for negated conditions. During a later episode, when all of the conditions of a special-case production would be satisfied in a subgoal, the chunk learned in the first trial bypasses the subgoal. If the special-case production would lead to a different result for the goal, the chunk is over-general and produces an incorrect result.

Figure 3-4 contains an example of how the problem solving and chunking in ***Soar*** can lead to over-generalization. Consider the situation where **O** is to move in state 1. It already has the center (E), while **X** is on a side (B). A tie arises between all the remaining moves (A, C, D, F, G, H, I) leading to the creation of a subgoal. The Selection problem space is chosen in which each of the tieing moves are candidates to be evaluated. If position I is evaluated first, it leads to a line of play resulting in state 2, which is a win for O because of a fork. On return to the Selection problem space, move I is immediately chosen as the best move, the original tie-subgoal terminates, move I is made, and O goes on to win. When returning from the tie-subgoal, a chunk is created, with conditions sensitive to all aspects of the original state that were tested in productions that fired in the subgoals. All positions that have marks were tested (A, B, C, E, I) as well as those positions that had to be clear for O to have a fork (G, F). However, positions D and H were not tested. To see how this production is over-general consider state 3, where O is to move. The newly chunked production, being insensitive to the X at position D, will fire and suggest position I, which leads to a loss for O.

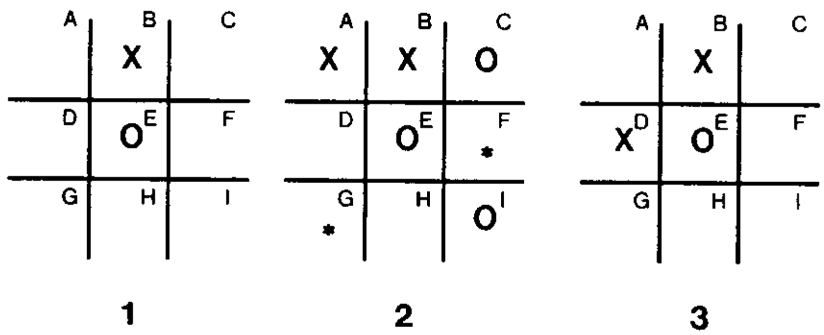

Figure 3-4: Over-generalization in Tic-Tac-Toe.

Over-generalization is a serious problem for ***Soar*** if we want to encode real tasks that are able to improve with experience. However, over-generalization is a problem for any learning system that works in many different environments and it leads to what is called *negative-transfer* in humans. We believe that the next step in handling over-generalization is to investigate how a problem solver can recover from over-general knowledge, and then carry out problem-solving activities so that new chunks can be learned that will override the over-general chunks. This would be similar to Anderson's work on discrimination learning using knowledge compilation (Anderson, 1983).

4. Conclusion

In this paper we have taken several steps towards the establishment of chunking as a general learning mechanism. We have demonstrated that it is possible to extend chunking to complex tasks that require extensive problem solving. In experiments with the Eight Puzzle, Tic-Tac-Toe, and a part of the ***R1*** computer-configuration task, it was demonstrated that chunking leads to performance improvements with practice. We have also contributed to showing how chunking can be used to improve many aspects of behavior. Though this is only partial, as not all of the different types of problem solving arose in the tasks we demonstrated, we did see that chunking can be used for subgoals that involve selection of operators and application of operators. Chunking has this generality because of the ubiquity of goals in ***Soar***. Since all aspects of behavior are open to problem solving in subgoals, all aspects are open to learning. Not only is ***Soar*** able to learn about the task (chunking the main goal), it is able to learn about how to solve the task (chunking the subgoals). Because all aspects of behavior are open to problem solving, and hence to learning, ***Soar*** avoids the wandering bottle-neck problem.

In addition to leading to performance speed ups, we have shown that the implicit generalization of chunks leads to significant within-trial and across-task transfer of learning. This was demonstrated most strikingly by the ability of chunks to use symmetries in Tic-Tac-Toe positions that are not evident to the problem solving system. And finally, we have demonstrated that chunking, which on first glance is a limited caching function, is capable of strategy acquisition. It can acquire the search control required to turn search-based problem solving into an efficient method.

Though significant progress has been made, there is still a long way to go. One of the original goals of the work on chunking was to model human learning, but several of the assumptions of the original model have been abandoned on this attempt, and a better understanding is needed of just why they are necessary. We also need to understand better the characteristics of problem spaces that allow interesting forms of generalization, such as use of symmetry to take place. We have demonstrated several forms of learning, but others, such as concept formation (Mitchell, 1978), problem space creation (Hayes and Simon, 1976), and learning by analogy (Carbonell, 1983) still need to be covered before the proposal of chunking as a general learning mechanism can be firmly established.

References

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

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

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

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

Laird, J. E. *Universal Subgoaling*. Doctoral dissertation, Computer Science Department, Carnegie-Mellon University, 1983.

Langley, P. Learning Effective Search Heuristics. In *Proceedings of IJCAI-83*. IJCAI, 1983.

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

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

Mitchell, T. M. *Version Spaces: An approach to concept learning*. Doctoral dissertation, Stanford University, 1978.

Mitchell, T. M. Learning and Problem Solving. In *Proceedings of IJCAI-83*. IJCAI, 1983.

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

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

Rosenbloom, P. S. *The Chunking of Goal Hierarchies: A Model of Practice and Stimulus-Response Compatibility*. Doctoral dissertation, Carnegie-Mellon University, 1983.

Rosenbloom, P. S., and Newell, A. The chunking of goal hierarchies: A generalized model of practice. In R. S. Michalski, J. G. Carbonell, & T. M. Mitchell (Eds.), *Proceedings of the 1983 Machine Learning Workshop.* , 1983.

Rosenbloom, P. S., Laird, J. E., McDermott, J., Newell, A., & Orciuch, E. R1-Soar: An experiment in knowledge-intensive programming in a problem-solving architecture. In *Proceedings of the IEEE Workshop on Principles of Knowledge-Based Systems*. IEEE Computer Society, 1984. (Also appears as the second half of this technical report).

R1-Soar
An Experiment in Knowledge-Intensive Programming in a Problem-Solving Architecture[1]

Paul S. Rosenbloom, John E. Laird, John McDermott, Allen Newell & Edmund Orciuch[2]

Repeatedly in the work on expert systems, domain-dependent *knowledge-intensive* methods are contrasted with domain-independent *general problem-solving methods* (Hayes-Roth, Waterman, & Lenat, 1983). Expert systems such as ***Mycin*** (Shortliffe, 1976) and ***R1*** (McDermott, 1982) attain their power to deal with applications by being knowledge intensive. However, this knowledge characteristically relates aspects of the task directly to action consequences, bypassing more basic scientific or causal knowledge of the domain. We will call this direct task-to-action knowledge *expertise knowledge* (it has also referred to as *surface knowledge* (Chandrasekaran & Mittal, 1983; Hart, 1982)), acknowledging that no existing term is very precise. Systems that primarily use weak methods (Laird & Newell, 1983a; Newell, 1969), such as depth-first search and means-ends analysis, are characterized by their wide scope of applicability. However, they achieve this at the expense of efficiency, being seemingly unable to bring to bear the vast quantities of diverse task knowledge that allow an expert system to quickly arrive at problem solutions.

This article describes ***R1-Soar***, an attempt to overcome the limitations of both expert systems and general problem-solvers by doing knowledge-intensive programming in a general weak-method problem-solving architecture. We wish to show three things: (1) a general problem-solving architecture can work at the knowledge-intensive (expert system) end of the problem-solving spectrum; (2) such a system can integrate basic reasoning and expertise; and (3) such a system can perform knowledge acquisition by automatically transforming computationally-intensive problem solving into efficient expertise-level rules.

Our strategy is to show how ***Soar***, a problem-solving production-system architecture (Laird, 1983), can deal with a portion of ***R1*** — a large rule-based expert system that configures Digital Equipment

---

[1]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 Digital Equipment Corporation. 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, or Digital Equipment Corporation.

[2]John McDermott and Allen Newell are at the Carnegie-Mellon University Computer Science Department. Most of this work was done while Paul Rosenbloom and John Laird were also at CMU CSD. Paul Rosenbloom is now at the Stanford University Departments of Computer Science and Psychology. John Laird is now at the Xerox Palo Alto Research Center. Edmund Orciuch is at Digital Equipment Corporation.

Corporation VAX-11 and PDP-11 computer systems. A *base* representation in ***Soar*** consists of knowledge about the goal to be achieved and knowledge of the operators that carry out the search for the goal state. For the configuration task this amounts to knowledge that detects when a configuration has been done and basic knowledge of the physical operations of configuring a computer. A system with a base representation is robust, being able to search for knowledge that it does not immediately know, but the search can be expensive.

Efficiency can be achieved by adding knowledge to the system that aids in the application of difficult operators and guides the system through combinatorially explosive searches. Expertise knowledge corresponds to this non-base knowledge. With little expertise knowledge, ***Soar*** is a domain-independent problem solver; with much expertise knowledge, ***Soar*** is a knowledge-intensive system. The efficient processing due to expertise knowledge replaces costly problem-solving with base knowledge when possible. Conversely, incompleteness in the expertise leads back smoothly into search in the base system.

In ***Soar***, expertise can be added to a base system either by hand crafting a set of expertise-level rules, or by automatic acquisition of the knowledge implicit in the base representation. Automatic acquisition of new rules is accomplished by *chunking*, a mechanism that has been shown to provide a model of human practice (Newell & Rosenbloom, 1981; Rosenbloom, 1983), but is extended here to much broader types of learning.

In the remainder of this article, we describe ***R1*** and ***Soar***, present the structure of the configuration task as implemented in ***Soar***, look at the system's behavior to evaluate the claims of this work and draw some conclusions.

1. R1 and the Task for R1-Soar

R1 is an expert system for configuring computers (McDermott, 1982). It provides a suitable expert system for this experiment because: (1) it contains a very large amount of knowledge; (2) its knowledge is largely pure expertise in that it simply recognizes what to do at almost every juncture; and (3) it is a highly successful application of expert systems, having been in continuous use by Digital Equipment Corporation for over four years (Bachant & McDermott, 1984). Currently written in ***Ops5*** (Forgy, 1981), ***R1*** consists of a database of over 7000 component descriptions, and a set of about 3300 production rules partitioned into 321 subtasks. The primary problem-solving technique in ***R1*** is match — recognizing in a specific situation precisely what to do next. Where match is insufficient, ***R1*** employs specialized forms of generate and test, multi-step look-ahead, planning in an abstract space, hill climbing, and backtracking.

Given a customer's purchase order, ***R1*** determines what, if any, modifications have to be made to the order for reasons of system functionality and produces a number of diagrams showing how the various components on the order are to be associated. In producing a complete configuration, ***R1*** performs a number of relatively independent subtasks; of these, the task of configuring unibus modules is by far the most involved. Given a partially ordered set of modules to be put onto one or more buses and a number of containers (backplanes, boxes, etc), the unibus configuration task involves repeatedly selecting a backplane and placing modules in it until all of the modules have been configured. The task is knowledge-intensive because of the large number of situation-dependent constraints that rule out various module placements. ***R1-Soar*** can currently perform more than half of this task. Since ***R1*** uses about a third of its knowledge (1100 of its 3300 rules) in performing the unibus configuration task, ***R1-Soar*** has approximately a sixth of the knowledge that it would require to perform the entire configuration task.

R1 approaches the unibus configuration task by laying out an abstract description of the backplane demands imposed by the next several modules and then recognizing which of the candidate backplanes is most likely to satisfy those demands. Once a backplane is selected on the basis of the abstract description, ***R1*** determines specific module placements on the basis of a number of considerations that it had previously ignored or not considered in detail. ***R1-Soar*** approaches the task somewhat differently, but for the most part makes the same judgements since it takes into account all but one of the six factors that ***R1*** takes into account. The parts of the unibus configuration task that ***R1-Soar*** does not yet know how to perform are mostly peripheral subtasks such as configuring empty backplanes after all of the modules have been placed and distributing boxes appropriately among cabinets. ***R1*** typically fires about 1000 rules in configuring a computer system; the part of the task that ***R1-Soar*** performs typically takes ***R1*** 80 - 90 rule firings, a twelfth of the total number.[3]. Since an order usually contains several backplanes, to configure a single backplane might take ***R1*** 20 - 30 rule firings, or about 3 - 4 seconds on a Symbolics 3600 Lisp Machine.

2. Soar

Soar is a problem-solving system that is based on formulating all problem-solving activity as attempts to satisfy goals via heuristic search in problem spaces. A problem space consists of a set of *states* and a set of *operators* that transform one state into another. Starting from an initial state the problem solver applies a sequence of operators in an attempt to reach a state that satisfies the goal

[3]This task requires a disproportionate share of knowledge — a sixth of the knowledge for a twelfth of the rule firings — because the unibus configuration task is more knowledge-intensive than most of the other tasks ***R1*** performs.

(called a *desired* state). Each goal has associated with it a problem space within which goal satisfaction is being attempted, a current state in that problem space, and an operator which is to be applied to the current state to yield a new state. The search proceeds via *decisions* that change the current problem space, state, or operator. If the current state is replaced by a different state in the problem space — most often it is the state generated by the current operator, but it can also be the previous state, or others — normal within-problem-space search results.

The knowledge used to make these decisions is called *search control*. Because all problem solving in ***Soar*** must take place in a problem space, search control must be computationally limited in that it can not involve problem solving. As long as the computation required to make a decision is within the limits of search control, and the knowledge required to make the decision exists, problem solving proceeds smoothly. However, ***Soar*** often works in domains where its search-control knowledge is either inconsistent or incomplete. When this happens, ***Soar***'s *universal subgoaling* mechanism (Laird, 1983) automatically creates a subgoal whose purpose is to obtain the knowledge which will allow the decision to be made. For example, if more than one operator can be applied to a state, and the available knowledge does not prefer one over the others, a subgoal will be created to find information leading to the selection of the appropriate one. Another example is when an operator is selected and its implementation requires problem solving. A subgoal is created to build the state that is the result of the operator.

A subgoal is attempted by selecting a problem space for it, with goal attainment interpreted as finding a desired state in that problem space. Should a decision be problematic in this new problem space, a new subgoal would be created to deal with it. The overall structure thus takes the form of a goal-subgoal hierarchy. Moreover, because each new subgoal will have an associated problem space, ***Soar*** generates a hierarchy of problem spaces, as well as a hierarchy of goals. The diversity of task domains is reflected in a diversity of problem spaces. Major tasks, such as configuring a computer, have a corresponding problem space, but so also do each of the various subtasks, such as placing a module into a backplane or placing a backplane into a box. In addition, problem spaces exist in the hierarchy for many types of tasks that often don't appear in a typical task-subtask decomposition, such as the selection of an operator to apply, the implementation of a given operator in some problem space, and a test of goal attainment.

Figure 2-1 gives a small example of how these subgoals are used in ***Soar***. This is a subgoal/problem-space structure that gets generated while trying to take steps in many task problem spaces. Initially (A), the problem solver is at State1 and must select an operator. If search control is unable to uniquely determine the next operator to apply, a subgoal is created to do the selection. In that subgoal (B), a *selection* problem space is used that reasons about the selection of objects from a set. In order to break the tie between objects, the selection problem space has operators to evaluate each candidate object.

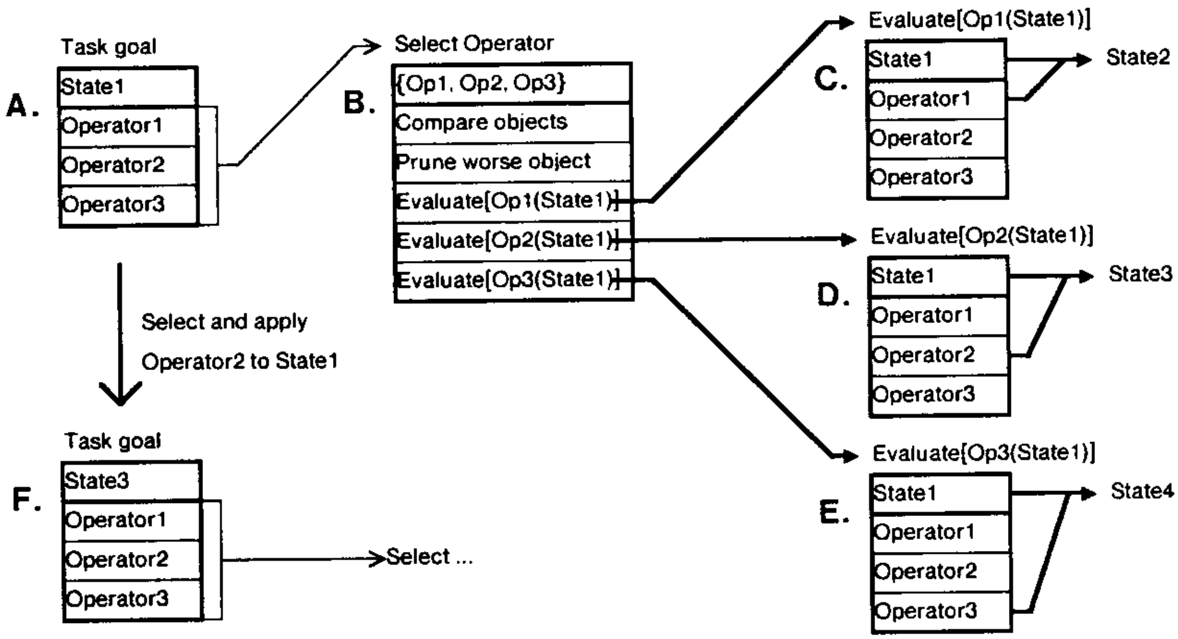

Figure 2-1: A ***Soar*** subgoal/problem-space structure.

Evaluating an operator, such as Operator1 in the task space, is a complex problem requiring a new subgoal. In this subgoal (C), the original task problem space and state (State1) are selected. Operator1 is applied, creating a new state (State2). If an evaluation function exists for State2, it is used to compare Operator1 to the other operators. When Operator1 has been evaluated, the subgoal terminates, and then the whole process is repeated for the other two operators (Operator2 and Operator3 in D and E). If, for example, Operator2 creates a state with a better evaluation than the other operators, it will be designated as better than them. The selection subgoal will terminate and the designation of Operator2 will lead to its selection in the original task goal and problem space. At this point Operator2 is reapplied to State1 and the process continues (F).

Soar uses a production system architecture — a modified version of ***Ops5*** (Forgy, 1981) that admits parallel execution of all satisfied productions — to realize its search-control knowledge and to implement its simple operators (more complex operators are encoded as separate problem spaces that are chosen for the subgoals that arise when the operator they implement has been selected to apply). Each production rule *elaborates* the current objects under consideration for a decision (e.g., candidate operators or states) with knowledge about the objects, including *preferences* relative to other candidate objects. There is a fixed decision process that integrates these preferences and makes a selection. Each decision corresponds to an elementary step in the problem solving, so a count of the number of decisions is a good measure of the amount of problem solving performed.

To have a task formulated in ***Soar*** is to have a problem space and the ability to recognize when a state satisfies the goal of the task; that is, is a desired state. The default behavior for ***Soar*** — when it has no search-control knowledge at all — is to search in this problem space until it reaches a desired state. The various weak methods arise, not by explicit representation and selection, but instead by the addition of small amounts of search control (in the form of one or two productions) to ***Soar***, which acts as a *universal weak method* (Laird & Newell, 1983a; Laird & Newell, 1983b; Laird, 1983). These production rules are responsive to the small amounts of knowledge that are involved in the weak methods, e.g., the evaluation function in hill-climbing or the difference between the current and desired states in means-ends analysis. In this fashion, ***Soar*** is able to make use of the entire repertoire of weak methods in a simple and elegant way, making it a good exemplar of a general problem-solving system.

The structure in Figure 2-1 shows how one such weak method, steepest-ascent hill climbing — at each point in the search, evaluate the possible next steps, and take the best one — can come about if the available knowledge is sufficient to allow evaluation of all of the states in the problem space. If slightly different knowledge is available, such as how to evaluate only *terminal* states (those states beyond which the search cannot extend), the search would be quite different, reflecting a different weak method. For example, if State2 in subgoal (C) cannot be evaluated, then subgoal (C) will not be satisfied, and the search will continue under that subgoal. An operator must be selected for State2, leading to a selection subgoal. The search will continue to deepen in this fashion until a terminal state is reached, leading to an exhaustive depth-first search for the best terminal state. If no evaluation information is available, that is, desired states can be recognized but not evaluated, a third weak method results: depth-first search for the first desired state to be found.

In addition to the kinds of knowledge that lead to the well-known weak methods, additional search-control knowledge can be added to any problem space. The knowledge can be in the form of new object preferences, or additional information that leads to new preferences. As more knowledge is added, the problem solving becomes more and more constrained until finally search is totally eliminated. This is the basic device in ***Soar*** to move towards a knowledge-intensive system. Each addition occurs simply by adding rules in the form of productions. Theoretically, ***Soar*** is able to move continuously from a knowledge-free solver (the default), through the weak methods to a knowledge-intensive system. It is possible to eliminate entire subspaces if their function can be realized by search-control knowledge in their superspace. For instance, if a subspace is to gather information for selecting an operator, then that information might be encodable as search control in the higher space. Similarly, if a subspace is to apply an operator, then specific instances of that operator might be carried out directly by rules in the higher space.

Knowledge acquisition in ***Soar*** consists of the creation of additional rules, by hand coding or by a mechanism that automatically *chunks* the results of successful goals (Laird, Rosenbloom, & Newell, 1984). The chunking mechanism creates new production rules that allow the system to directly perform actions that originally required problem solving in subgoals. The conditions of a chunked rule test those aspects of the task that were relevant to satisfying the goal, while its actions generate the information that actually satisfied the goal. New rules form part of search control when they deal with the selection among objects (chunks for goals that use the selection problem space), or they form part of operator implementation when they are chunks for goals dealing with problematic operators. Because ***Soar*** is driven by the goals automatically created to deal with difficulties in its performance, and chunking works for all goals, the chunking mechanism is applicable to all aspects of ***Soar's*** problem-solving behavior.

3. The Structure of R1-Soar

The first step in building a knowledge-based system in ***Soar*** is to design and implement the base representation as a set of problem spaces within which the problem can be solved. As displayed in Figure 3-1, ***R1-Soar*** currently consists of a hierarchy of ten task problem spaces (plus the selection problem space). These spaces represent a decomposition of the task in which the top space is given the goal to do the entire unibus configuration task; that is, to configure a sequence of modules to be put on a unibus. The other nine task spaces deal with subcomponents of this task. Each subspace implements one of the complex operators of its parent's problem space.

Each configuration task begins with a goal that uses the Unassigned Backplane problem space. This space has one operator for configuring a backplane that is instantiated with a parameter that determines which type of backplane is to be configured. The initial decision, of selecting which backplane to use next, appears as a choice between instances of this operator. Unless there is special search-control knowledge that knows which backplane should be used, no decision can be made. This difficulty (of indecision) leads to a subgoal that uses the selection problem space to evaluate the operators (by applying them to the original state and evaluating the resulting states). To do this, the evaluation operator makes recursive use of the Unassigned Backplane problem space.

The initial configuration of a backplane is accomplished in the five problem spaces rooted at the

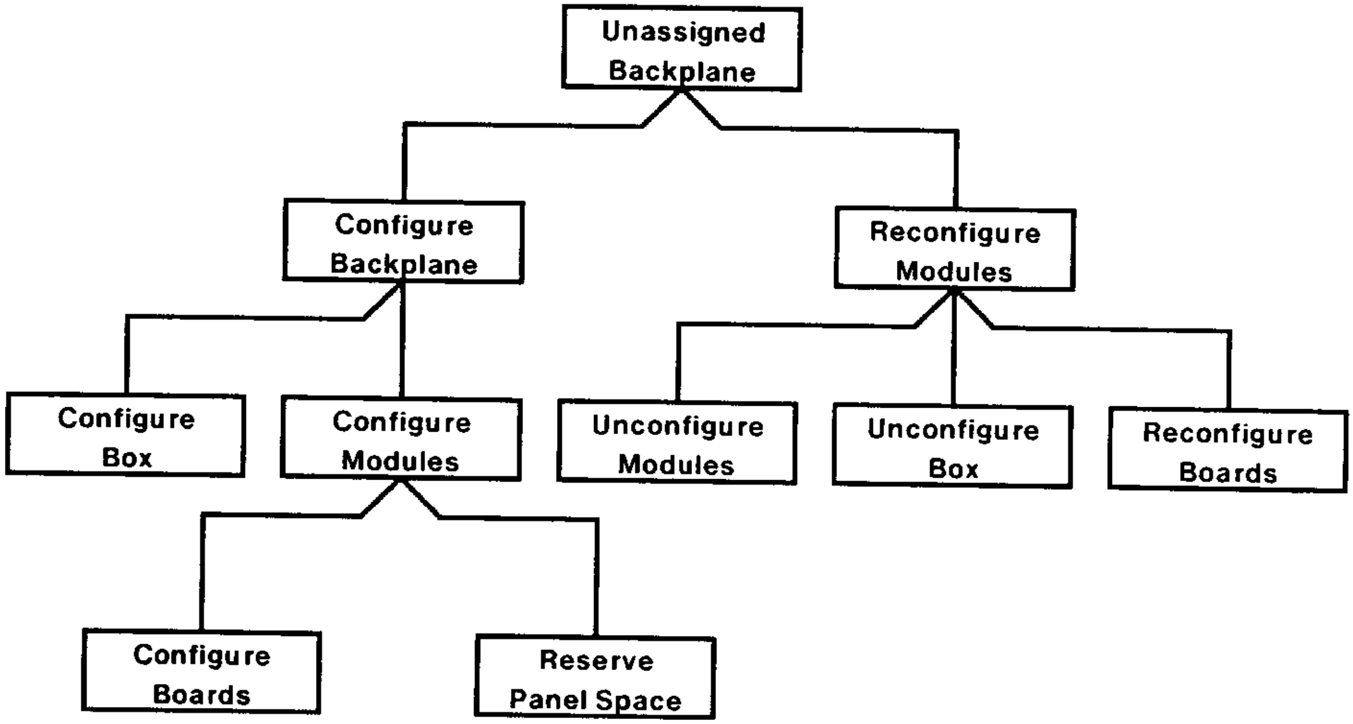

Figure 3-1: The task problem-space hierarchy for *R1-Soar*.

Configure Backplane space by: putting the backplane in a box (the Configure Box space); putting into the backplane as many modules as will fit (the Configure Modules space); reserving panel space in the cabinet for the module (the Reserve Panel Space space); putting the modules' boards into slots in the backplane (the Configure Boards space); and cabling the backplane to the previous backplane (done by an operator in the Configure Backplane space that is simple enough to be done directly by a rule, rather than requiring a new problem space). Each of these problem spaces contains between one and five operators. Some of the operators are simple enough to be implemented directly by rules, such as the cable-backplane operator in the Configure Backplane space, or the put-board-in-slot, go-to-next-slot, and go-to-previous-slot operators in the Configure Boards space. Others are complex enough to require problem solving in new problem spaces, yielding the problem-space hierarchy seen in Figure 3-1.

In addition to containing operators, each problem space contains the knowledge allowing it to recognize the satisfaction of the goal for that problem space. Several kinds of goal detection can occur: (1) recognition of a desired state; (2) satisfaction of path constraints (avoiding illegal sequences of operators); and (3) optimization over some criterion (such as maximizing the value of the result or minimizing its cost). All these different forms of goals are realized by appropriate production rules. For example, the Configure Backplane space simply has the following goal detection rule: if the modules have been placed in the backplane, and the backplane has been placed in a box, and the backplane has been cabled to the previous backplane (if there is one), then the goal is accomplished. In a more complicated case, the task of putting the boards from a module into slots in a backplane (the Configure Boards space) could be considered complete whenever all of the module's boards are associated with slots in the backplane. However, a two-board module can be configured by putting one board in the first slot and one in the last slot, or by putting the two boards into the first two slots, or by any one of the other combinatorial possibilities. For most modules it is desirable to put the boards as close to the front as possible to leave room for later modules (though there is one type of module that must go at the end of the backplane), so completed configurations are evaluated according to how much backplane space (first to last slot) they use. The goal is satisfied when the best completed configuration has been found.

In addition to the constraints handled by evaluation functions (such as using minimum backplane space), many other constraints exist in the configuration task that complicate the task of a problem-solving system. These include possible incompatibilities between boards and slots, the limited amounts of power that the boxes provide for use by modules (a new box may be required if more power is needed), components that are needed but not ordered, restrictions on the location of a module in a backplane (at the front or back), and limits on the electrical length of the unibus (for which a unibus repeater is required). ***R1-Soar*** pursues this complex configuration problem by searching for the best configuration that meets all of the constraints, and then trying to optimize the configuration some more by relaxing one of the constraints — the ordering relationship among the modules. This relaxation (occurring in the four spaces rooted at the Reconfigure Modules space) may allow the removal of backplanes that were added over and above those on the initial order. When possible, the modules configured in these backplanes are removed (the Unconfigure Modules space), placed into unused locations in other backplanes (the Reconfigure Boards space), and the extra backplanes are removed from their boxes (the Unconfigure Box space).

As described so far, ***R1-Soar*** forms a base-reasoning system, because its representation and processing is in terms of the fundamental relationships between objects in the domain. The main mode of reasoning consists of search in a set of problem spaces until the goals are achieved. One part of this search can be seen clearly in the Configure Boards space. Given a module with two boards of width 6, and a nine-slot backplane with slot-widths of 4-6-6-6-6-6-6-6-4, a search proceeds through the problem space using the go-to-next-slot and put-board-in-slot operators. The search begins begins by making the easy decision of what to do with the first slot: it must be skipped because it is too narrow for either board. Then either one board can be placed in the second slot, or the slot can be skipped. If it is skipped, one board can be placed in the third slot, or it can be skipped, and so on. If instead a board is placed in the second slot, then it must go on to the third slot and decide whether to place the other board there or to skip the slot, and so on. All complete configurations are evaluated, and the path to the best one is selected. This is clearly not the most efficient way to solve this problem but it is sufficient.

R1-Soar becomes a more knowledge-intensive system as rules are added to guide the search through the problem space and to implement special cases of operators — even though the complete operator is too complex for direct rule implementation, special cases may be amenable. Most of the hand-crafted knowledge in ***R1-Soar*** is used to control the search. In the Configure Boards space all search is eliminated (at least for modules that go in the front of the backplane) by adding three search-control rules: (1) operators that configure boards of the same width and type are equal; (2) prefer the operator that configures the widest board that will fit in the current backplane slot; and (3) prefer an operator that puts a board into a slot over one that goes to the next slot. These rules convert the search in this problem space from being depth-first to algorithmic — at each step the system knows exactly what to do next. For the example above, the correct sequence is: go-to-next-slot, put-board-in-slot, go-to-next-slot, put-board-in-slot.

4. Results and Discussion

In this section we evaluate how well ***R1-Soar*** supports the three objectives given in the introduction by examining its performance on four configuration tasks.

1. There is one two-board module to be put on the unibus.

2. There are three modules to be put on the unibus. One of the already configured backplanes must be undone in order to configure a unibus repeater.

3. There are six modules to be put on the unibus. Three of the modules require panel space in the cabinet.

4. There are four modules to be put on the unibus. Three of the modules will go into a backplane already ordered, and one will go into a backplane that must be added to the order. Later this module is reconfigured into an open location in the first backplane, allowing removal of the extra backplane from the configuration.

Most of the results to be discussed here are for tasks 1 and 2, which were done in earlier versions of both ***Soar*** and ***R1-Soar*** (containing only the Unassigned Backplane, Configure Backplane, Configure Box, Configure Modules, and Configure Boards spaces, for a total of 242 rules). Tasks 3 and 4 were run in the current versions of ***Soar*** and ***R1-Soar*** (containing all of the problem spaces, for a total of 266 rules[4]). Table 4-1 gives all of the results for these four tasks that will be used to evaluate the three objectives of this paper. The first line in the table shows that a system using a base representation can work, solving the rather simple task 1 after making 1731 decisions.

Task	Version	Before Learning	During Learning	After Learning
1	Base	1731	485 *[59]*	7
	Partial	243	111 *[14]*	7
	Full	150	90 *[12]*	7
2	Partial	1064	692 *[109]*	16
	Full	479	344 *[53]*	16
3	Full	288	143 *[20]*	10
4	Full	628		

Table 4-1: Number of decisions to completion for the four unibus configuration tasks. The base version (task 1) contains 232 rules, the partial version (tasks 1 and 2) contains 234 rules, and the full version contains 242 rules (tasks 1 and 2) or 266 rules (tasks 3 and 4). The number of rules learned for each task is shown in brackets in the during-learning column.

The first objective of this paper is to show that a general problem-solving system can work effectively at the knowledge-intensive end of the problem-solving spectrum. We examine three qualitatively different knowledge-intensive versions of ***R1-Soar***: (1) where it has enough hand-crafted rules so that its knowledge is comparable to the level of knowledge in ***R1*** (before learning on the full version); (2) where there are rules that have been acquired by chunking (after learning on the base version); and (3) where both kinds of rules exist (after learning on the full version). The hand-crafted expertise consists solely of search control (operator selection) rules. The chunked expertise consists of both search-control and operator-application rules. In either case, this is expertise knowledge, directly relating knowledge of the task domain to action in the task domain.

[4]The difference in number of task rules between these two versions is actually higher because a number of the default (non-task) rules needed by earlier versions of ***Soar*** are no longer necessary.

Table 4-1 shows the number of decisions required to complete each of the four configuration tasks when these three versions of ***R1-Soar*** are used. With hand-crafted search control, all four tasks were successfully completed, taking between 150 and 628 decisions. In the table, this is before learning on the full (search control) version. With just chunked search control, task 1 was accomplished in 7 decisions (after learning on the base version). A total of 3 of the 7 decisions deal with aspects outside the scope of the unibus configuration task (setting up of the initial goal, problem space, and state). ***Soar*** takes about 1.4 seconds per decision, so this yields about 6 seconds for the configuration task — within a factor of 2 of the time taken by ***R1***. It was not feasible to run the more complex task 2 without search control because the time required would have been enormous due to the combinatorial explosion — the first module alone could be configured in over 300 different ways. Tasks 3 and 4 were also more complicated than task 1, and were not attempted with the base version. With both hand-crafted and chunked search control, tasks 1-3 required between 7 and 16 decisions (after learning on the full version). Task 4 learning had problems of overgeneralization. It should have learned that one module could not go in a particular backplane, but instead learned that the module could not go in any backplane. More discussion on overgeneralization in chunking can be found in Laird, Rosenbloom, and Newell (1984).

In summary for the first objective, ***R1-Soar*** is able to do the unibus configuration task in a knowledge-intensive manner. To scale this result up to a full expert system (such as all of ***R1***) we must know: (1) whether the rest of ***R1*** is similar in its key characteristics to the portion already done; and (2) the effects of scale on a system built in ***Soar***. With respect to the unibus configuration task being representative of the whole configuration task, qualitative differences between portions of ***R1*** would be expected to manifest themselves as differences in amount of knowledge or as differences in problem-solving methods. The task that ***R1-Soar*** performs is atypical in the amount of knowledge required, but requires more knowledge, not less — 15.7 rules per subtask for ***R1-Soar***'s task, versus 10.3 for the entire task. The problem-solving methods used for the unibus configuration task are typical of the rest of ***R1*** — predominantly match, supplemented fairly frequently with multi-step look ahead. With respect to the scaling of ***R1-Soar*** up to ***R1***'s full task, ***Ops5***, from which ***Soar*** is built, scales very well — time is essentially constant over the number of rules, and linear in the number of modifications (rather than the absolute size) of working memory (Gupta and Forgy, 1983). Additional speed is also available in the form of the ***Ops83*** production-system architecture, which is at least 24 times faster than Lisp-based ***Ops5*** (on a VAX-780) (Forgy, Gupta, Newell, & Wedig, 1984), and a production-system machine currently being designed that is expected to yield a further multiplicative factor of between 40 and 160 (Forgy, Gupta, Newell, & Wedig, 1984), for a combined likely speed-up of at least three orders of magnitude.

The second objective of this article is to show how base reasoning and expertise can be combined in ***Soar*** to yield more expertise and a smooth transition to search in problem spaces when the expertise is incomplete. Towards this end we ran two more before-learning versions of ***R1-Soar*** on tasks 1 and 2: (1) the base version, which has no search-control rules; and (2) the partial version, which has two hand-crafted search-control rules. The base version sits at the knowledge-lean end of the problem-solving spectrum; the partial version occupies an intermediate point between the base system and the more knowledge-intensive versions already discussed.

Task 1 took: (1) 1731 decisions for the base version; and (2) 243 decisions for the partial version. Examining the trace of the problem solving reveals that most of the search in the base version goes to figuring out how to put the one module into the backplane. For the 9-slot backplane (of which 7 slots were compatible with the module's two boards), there are (7 choose 2) = 21 pairs of slots to be considered. The two search control rules added in the partial version have already been discussed in the previous section: (1) make operators that configure boards of equal size be equal, and (2) prefer to put a board in a slot rather than skip the slot. These two rules reduce the number of decisions required for this task by almost an order of magnitude. With the addition of these two search control rules, the second task could also be completed, requiring 1064 decisions.

In summary, the base system is capable of performing the tasks, albeit very slowly. If appropriate search control exists, search is reduced, lowering the number of decisions required to complete the task. If enough rules are added, the system acts like it is totally composed of expertise knowledge. Where such knowledge is missing, as some is missing in the partial version, the system falls back on search in its problem spaces.

The third objective is to show that knowledge acquisition via ***Soar***'s chunking mechanism could compile computationally intensive problem solving into efficient rules. In ***Soar***, chunks are learned for all goals experienced on every trial, so for exact task repetition (as is the case here), all of the learning occurs on the first trial. The *during learning* column in Table 4-1 shows how many decisions were required on the trial where learning occurred. The bracketed number is the number of rules learned during that trial. These results show that learning can improve performance by a factor of about 1.5 to 3, even the first time a task is attempted. This reflects a large degree of within-trial transfer of learning; that is, a chunk learned in one situation is reused in a later situation during the same trial. Some of these new rules become part of search control, establishing preferences for operators or states. Other rules become part of the implementation of operators, replacing their original implementations as searches in subspaces, with efficient rules for the particular situations.

In task 3, for example, three operator-implementation chunks (comprising four rules) were learned and used during the first attempt at the task. Two of the chunks were for goals solved in the Configure Boards space. Leaving out some details, the first rule says that if the module has exactly one board and it is of width six, and the next slot in the backplane is of width six, then put the board into the next slot and move the slot pointer forward one slot. This is a macro operator which accomplishes what previously required two operators in a lower problem space. The second rule says that if the module has two boards, both of width six, and the current slot is of width four (too small for either board), and the two subsequent slots are of width six, then place the boards in those slots, and point to the last slot of the three as the current slot. The third rule is a more complex one dealing with the reservation of panel space.

Comparing the number of decisions required before learning and after learning reveals savings of between a factor of 20 and 200 for the four unibus configuration tasks. In the process, between 12 and 109 rules are learned. The number of rules to be learned is determined by the number of distinct subgoals that need to be satisfied. If many of the subgoals are similar enough that a few chunks can deal with all of them, then fewer rules must be learned. A good example of this occurs in the base version of task 1, where most of the subgoals are resolved in one problem space (the Configure Boards space). Likewise, a small amount of general hand-crafted expertise can reduce significantly the number of rules to be learned. For task 1, the base version plus 59 learned rules leads to a system with 291 rules, the partial version plus 14 learned rules has 248 rules, and the full version plus 12 learned rules has 254 rules (some of the search control rules in the full version do not help on this particular task). All three systems require the same number of decisions to process this configuration task.

In summary, chunking can generate new knowledge in the form of search-control and operator-implementation rules. These new rules can reduce the time to perform the task by nearly two orders of magnitude. For more complex tasks the benefits could be even larger. However, more work is required to deal with the problem of overgeneralization.

5. Conclusion

By implementing a portion of the ***R1*** expert system within the ***Soar*** problem solving architecture, we have provided evidence for three hypotheses: (1) a general problem solving architecture can work at the knowledge intensive end of the problem solving spectrum; (2) such a system can effectively integrate base reasoning and expertise; and (3) a chunking mechanism can aid in the process of knowledge acquisition by compiling computationally intensive problem solving into efficient expertise-level rules.

The approach to knowledge-intensive programming can be summarized by the following steps: (1) design a set of base problem spaces within which the task can be solved; (2) implement the problem-space operators as either rules or problem spaces; (3) operationalize the goals via a combination of rules that test the current state, generate search-control information and compute evaluation functions; and (4) improve the efficiency of the system by a combination of hand crafting more search control, using chunking, and developing evaluation functions that apply to more states.

References

Bachant, J. and McDermott, J. R1 revisited: four years in the trenches. *AI Magazine*, 1984, Vol. 5(3).

Chandrasekaran, B. & Mittal, S. Deep versus compiled knowledge approaches to diagnostic problem-solving. *International Journal of Man-Machine Studies*, 1983, *19*, 425-436.

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

Forgy, C., Gupta, A., Newell, A., & Wedig, R. Initial assesment of architectures for production systems. In *Proceedings of AAAI-84, National Conference on Artificial Intelligence*. American Association of Artificial Intelligence, 1984.

Gupta, A. and Forgy, C. *Measurements on Production Systems* (Tech. Rep.). Computer Science Department, Carnegie-Mellon University, December 1983.

Hart, P. E. Directions for AI in the eighties. *SIGART Newsletter*, 1982, *79*, 11-16.

Hayes-Roth, F., Waterman, D. A., & Lenat, D. B. An overview of expert systems. In F. Hayes-Roth, D. A. Waterman, & D. B. Lenat (Ed.), *Building Expert Systems*. Reading, MA: Addison-Wesley, 1983.

Laird, J. E. *Universal Subgoaling*. Doctoral dissertation, Carnegie-Mellon University, 1983. (Available as Carnegie-Mellon University Computer Science Tech. Rep. #84-129).

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

Laird, J. E., and Newell, A. A universal weak method: Summary of results. In *Proceedings of the Eighth IJCAI.*, 1983.

Laird, J. E., Rosenbloom, P. S., & Newell, A. Towards chunking as a general learning mechanism. In *Proceedings of AAAI-84, National Conference on Artificial Intelligence*. American Association for Artificial Intelligence, 1984. (Also appears as the first half of this technical report).

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

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

Newell, A. & Rosenbloom, P. S. Mechanisms of skill acquisition and the law of practice. In J. R. Anderson (Ed.), *Cognitive Skills and Their Acquisition*. Hillsdale, NJ: Erlbaum, 1981. (Also available as Carnegie-Mellon University Computer Science Tech. Rep. #80-145).

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

Shortliffe, E. H. *Computer-Based Medical Consultation: MYCIN*. New York: American Elsevier, 1976.