Why Some Chunks Are Expensive

Milind Tambe and Allen Newell

11 January 1988
CMU-CS-88-103

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order Nbr. 4976, Amendment 20, under contract number F33615-87-C-1499, monitored by the Air Force Avionics Laboratory and by the Encore Computer Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as represeting the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Why Some Chunks Are Expensive

Milind Tambe and Allen Newell
Department of Computer Science,
Carnegie Mellon University,
Pittsburgh, PA 15213

11 January 1988

Abstract

Soar is an attempt to realize a set of hypothesis on the nature of general intelligence within a single system. One central hypothesis is that chunking, a simple experience-based learning mechanism, can form the basis for a general learning mechanism. It is already well established that the addition of chunks improves the performance in Soar a great deal, when viewed in terms of subproblems required and number of steps within a subproblem. But this high level view does not take into account potential offsetting costs that arise from various computational effects. This paper is an investigation into the computational effect of expensive chunks. These chunks add significantly to the time per step by being individually expensive. We decompose the causes of expensive chunks into three components and identify the features of the task environment that give rise to them. We then discuss the implications of the existence of expensive chunks for a complete implementation of Soar.

1. Introduction

Soar is a system that attempts to realize a set of hypothesis on the nature of general intelligence [8]. One central hypothesis is that chunking [6], a simple experience-based learning mechanism, can form the basis for a general learning mechanism.

Chunking improves performance in Soar by reducing the amount of problem solving required to achieve a goal. At each step during problem solving, the system searches its knowledge base for knowledge relevant to the current problem-solving situation. The knowledge base is encoded as a production system, i.e., a set of condition-action rules, each of which fires whenever its conditions match elements in working memory. The chunking mechanism creates new productions (chunks), based on the results of problem solving, and adds them to the existing set of productions. These chunks then fire in appropriate later situations, to provide these results again. Thus, chunking provides both a practice mechanism and, when similar situations arise, a learning-transfer mechanism.

In Soar all activity is expressed as problem solving, whether selecting an operator to apply, implementing an operator, deciding what space to use for problem solving, or whatever. Thus, chunking applies to all the aspects of Soar's operations, which is a necessary condition for chunking to be the basis for all of Soar's learning. Soar is to build chunks continuously, and at a roughly constant rate, throughout its lifetime. Current experience places the rate of chunk building at more than 1 chunk per 100 production cycles. Thus, a long-running version of Soar will ultimately comprise a large number of productions, essentially all of which will be chunks, created by the system itself.

It is important to understand the computational properties of such a growing system. What happens to the running time of Soar as it continues to add chunks? It is already well established that the addition of chunks improves the performance a great deal, when viewed in terms of subproblems required and numbers of steps within a subproblem [18]. But this high level view does not take into account potential offsetting costs that arise from the growth of productions. The literature contains analyses of this tradeoff for Strips-like systems [9, 10], which suggest that there may be real problems for Soar.

This paper is an investigation into one of the computational issues for Soar, namely that of individually expensive chunks. To pose this issue, we need to separate it from other aspects of what happens to performance when learning takes place. It is appropriate to think in terms of an *ideal computational model* for the production system, one that says how much time is taken per step, as a function of relevant parameters. The simplest ideal model is the *constant-step model*, namely that each step takes a constant amount of time, independent of the composition of the production system, i.e., independent of the number of productions, the number and type of conditions, etc. Given this ideal model, the effects of learning can be divided into two major types, which we will call *cognitive* and *computational*, just to have two simple names. Cognitive effects are all the changes in the steps to solve a problem, assuming the ideal model. These may be improvements, if the system does less problem solving because of the recalled prior results; these may be failures and inefficiencies, if the recalled results lead the system down inappropriate paths. These improvements can be measured by the number of steps as specified by the ideal model, since per assumption these steps take constant time.

Computational effects are all those that distort the model and invalidate it in various ways. There may be many such effects, which may themselves need to be separated for purposes of study. For instance there is the *average growth effect*, which is the average increase in time per step as the number of chunks grows. Some such effect must exist asymptotically for any system with limited parallelism. However, the increase may not be large enough to be of importance (and there exist some models of growth that support such a conclusion [5]). There may be other effects. For instance, more condition elements (productions with more condition elements) could take more time, with certain tasks or environments continually creating chunks with more conditions, even though the average growth effect over all environments is constant.

Each such effect, if it is appreciable, may modify substantially any conclusion about learning made on the basis of an analysis of cognitive effects alone. In the extreme, as conjectured in [9], it might be that there is no overall improvement, with the losses in the computational model offsetting completely the gains in numbers of steps. Even if this is the case, it may be possible to replace the simple constant-step model with another model that exhibits the important dependencies, and hence permits a useful total analysis of learning in terms of the ultimate measure, namely time.

The computational question that this paper deals with is the nature of *expensive chunks*. These are individual chunks that add significantly to the total time per step, and hence invalidate the ideal model. Such expensive chunks do not necessarily say anything about the average growth effect. We seek first to establish that such expensive chunks exist and then to establish their causes. Behind such an investigation is always the question of what can be done to alleviate such chunks, to avoid their being expensive or to avoid building them. We will comment on such issues, but the main intent of the paper is understanding.

Before we start, we must settle the question of the ideal constant-step model, for there are two possibilities for what a step should be. The *constant-cycle model* posits that each cycle of the production system takes a constant amount of time, independent of the composition of the production system. The *constant-action model* posits that each righthand side action of a production takes a constant amount of time, independent of the composition of the production system. These models are related, of course, since a given production has a certain number of righthand side actions (typically 3-4), and it might seem of little point to choose (unless the actions per production increased over time or something). However, implementations define one quantity or the other (cycles or actions) as the fundamental unit, and hence it is useful to choose the one dictated by the implementation. The implementation used for Soar (the Rete net, described below) takes righthand side actions as fundamental, so we will use the constant-action ideal model.

We will first establish that expensive chunks occur. Then we will step back to discuss the aspects of the implementation that are necessary to the analysis. This will permit us to decompose the causes of expensive chunks into three components. For each of these we will show something of the size of the effect and of the features of the task environments that gives rise to them. Finally, we will discuss what the existence of expensive chunks implies for a complete implementation of Soar.

2. Expensive Chunks Exist

Table 2-1 shows the effect of chunking in a number of tasks taken from our current experience with Soar (a description of these tasks is given in the Appendix). The first column gives the name of the task. The second column gives the number of steps in terms of right-hand side actions (called *actions* henceforth) when the system performs the task without (i.e., *before*) chunking. The third column gives the number of actions when the system performs the same task *after* chunking on that task. The fourth column gives the speedup caused by chunking in terms of number of actions. The fifth column gives the total match time for the task before chunking, i.e., the run corresponding to column 2. The sixth column gives the total match time corresponding to the run in the third column i.e., the run after chunking. The seventh column gives the speedup in terms of the total match time. The eighth column gives the total number of chunks that were added to the system by chunking. These measurements were done on the Soar/PSM-E [20], a system that uses the OPS83 software technology [3] for performing the match.[1]

[1]The Soar/PSM-E is implemented on the Encore Multiprocessor which uses the NS32032 processors, which are about 0.75 MIPS. The numbers presented here are for a uniprocessor configuration, so this paper does not address any issues of multiprocessing.

Task	Number of actions Before chunking	Number of actions After chunking	Speedup in terms of nbr. of actions	Total match time Before chunking (sec)	Total match time After chunking (sec)	Speedup in terms of total match time	Number of chunks added
8-Puzzle	3025	463	6.53	26.68	26.93	0.99	11
Queens	767	147	5.21	2.79	8.64	0.32	3
Path	5512	407	13.54	17.95	20.90	0.85	14
Magic-Square	2142	325	6.59	12.09	46.74	0.25	5
Syllogisms	1055	91	11.59	7.50	0.73	10.27	10
Monkey	1321	213	6.20	5.01	0.97	5.16	4
Waterjug	2623	287	9.13	10.51	2.01	5.22	11
Farmer	3427	309	11.09	17.64	3.50	5.04	14

Table 2-1: Effects of chunking on number of actions and total match time.

We see that chunking has caused a big speedup in the number of actions in all the tasks. This speedup is what we have called the *cognitive effect*. However, we see that for the tasks in the upper half of the table, i.e. the 8-puzzle, Queens, Path and Magic-Square[2] the total match has actually *increased* after chunking. For example, the Magic-square task shows almost a four fold slowdown after chunking. For other tasks, (we have included Syllogisms, Monkeys and Bananas, Waterjug and Farmer), the speedups in terms of number of actions due to chunking are followed up by a speedup in the total match time.

Comparing the total match time before and after chunking, independent of the number of actions, does not present a clear picture of the impact of chunking on the matcher. As mentioned in Section 1, it is better to think in terms of the time/step, i.e. time/action instead of the total match time. Table 2-2 gives the effect of chunking on the time/action for the set of eight tasks presented in Table 2-1. The first column gives the name of the task. The second column gives the time/action, T_B, when the system performs the task without *(before)* chunking. The third column gives the time/action, T_A, when the system performs the same task *after* chunking on that task. These runs correspond to the runs in Table 2-1.

It is clear that, out of T_A, the time/action after chunking, some fraction is spent in processing the original set of productions and some fraction is spent in processing the chunks. The implementation used in Soar (the Rete net, described below) makes it difficult to make a precise measurement of the effort applied to match an individual production or a subset of a given set of productions. However, we can obtain an approximate measure T_C, of the time/action spent in processing the chunks by assuming that processing the original set of productions still consumes T_B amounts of time/action. Thus T_C, which is the difference between T_A and T_B gives us the approximate time/action spent in processing the chunks. Column 4 in Table 2-2 presents T_C for all the tasks.

We had observed in Table 2-1, that for 8-puzzle, Queens, Path and Magic-square, the total match time had increased in spite of the decrease in the number of actions, after chunking. T_C in Table 2-2 shows that chunking has caused a drastic increase in time/action in these four tasks, completely distorting the ideal computational model of constant time/step. For the tasks in the lower half of Table 2-2, the time/action is seen not to increase nearly as

[2]The Magic-square task requires a very large amount of memory after chunking. Due to virtual memory limitations of the current version of the kernel running on the Encore, this task was therefore run only until memory was exceeded.

Task	T_B Time/action Before Chunking (ms)	T_A Time/action After Chunking (ms)	$T_C = T_A - T_B$ (ms)
8-Puzzle	8.82	58.09	49.27
Queens	3.63	58.77	55.14
Path	3.25	51.35	48.10
Magic-Square	5.64	143.81	138.17
Syllogisms	7.11	8.02	0.91
Monkey	3.79	4.56	0.77
Waterjug	4.01	7.03	3.02
Farmer	5.14	11.32	6.18

Table 2-2: Effect of chunking on time/action for a few Soar tasks.

much after chunking. Though T_C is an approximate measure of time/action spent in processing the chunks, we are interested only in establishing the existence of expensive chunks and in dealing with the order of magnitude differences between the time spent in processing expensive and cheap chunks. Therefore, a very precise measure of the time spent in processing chunks is unnecessary and we will use T_C as a measure of the cost of processing chunks in the rest of the paper.

Thus the addition of a few chunks has caused a very large increase in time/action in some tasks. However, this increase, per se might not allow us to establish that these few chunks are expensive. For instance, some chunks might appear to be expensive because they were fired a lot of times; while other chunks may appear to be cheap because they fired only a very few times. To address this issue, we present a graph in Figure 2-1, which plots T_C on the vertical axis against the number of firings of the chunks[3] for the set of eight tasks examined above.

It can be seen from the graph that T_C, the increase in time/action caused by chunking is independent of the number of chunk firings. Thus the expensive chunks are not somehow caused by the fact that they fire a lot of times. Likewise, the tasks with cheap chunks i.e. with a small increase in time/action, are cheap independent of the number of times they fired.

There is another possibility: the system may have unsuccessfully attempted firing some chunks a large number of times. This could cause those chunks to appear expensive, though individually, firing those chunks may actually require very little processing. In general for production systems, measuring the number of times the system attempted to fire a chunk can be very difficult. This issue is simplified in Soar because every task done in Soar consists of a set of states representing possible situations in the task domain and a set of operators that transform one state into another one (described in detail in the next section). The system attempts to fire a chunk in a given state if at all and then it is only in the next state that another attempt is made to fire that chunk again. Thus the number of states that the problem solver goes through can serve as an approximate estimate of the number of times the system tried to process the chunks. Figure 2-2 plots T_C on the vertical axis against the number of states used in problem-solving.

Figure 2-2 shows that the differences in T_C cannot be due to the fact that the system has processed some chunks

[3]Note that a chunk firing implies firing of all instantiations of that chunk in that production cycle.

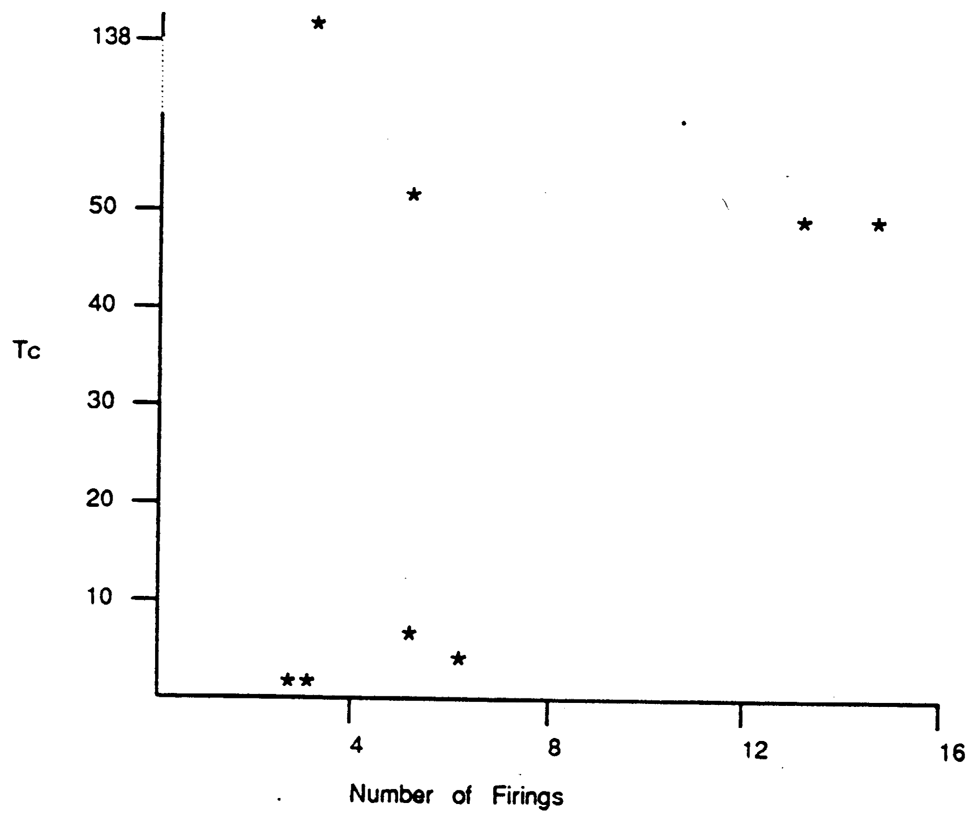


Figure 2-1: T_C and the number of chunk firings.

more times than others. Thus the graphs in Figures 2-1 and 2-2 establish that the increase in time/action, T_C, is due to individually expensive chunks that require very large amounts of time to process.[4] We have seen that expensive chunks occurred in 8-puzzle, Queens, Path and Magic-Square tasks, but the chunks in Syllogisms, Monkeys and Bananas, Waterjug and Farmer were seen to be relatively cheap. In sum, expensive chunks arise naturally, but not universally. It is important to track them down and understand why they exist, how bad they might be, and what might be done about them. They do not arise often — these four cases constituted the verifiable and analyzable set that came to light upon investigation[5]. But when they do happen, they are dramatic in their effect.

3. Soar

The goal of the Soar project is to build a system capable of general intelligent behaviour. Soar has been applied to a wide variety of tasks: many of the classic AI toy tasks, such as the blocks world and towers of Hanoi; tasks that appear to involve non-search-based reasoning, such as syllogisms; and large tasks such as algorithm design and the R1 computer configuration task [18, 19]. The next two subsections describe the two major components of the Soar architecture, the performance system and the mechanism of chunking, just enough to support the further analysis.

[4]Alternatively, the number of firings can viewed as providing a lower bound and the number of states as providing an upper bound on the number of attempts made by the system in processing the chunks. The graphs in Figures 2-1 and 2-2 together allow us to establish that the difference in the cost of expensive and cheap chunks is not due to a difference in the number of times the chunks are processed.

[5]Other putative expensive chunks were profferred or rumored, but could not be tracked down or could not be replicated in the C-based Soar/PSM-E system rather than the Lisp-based versions used mostly.

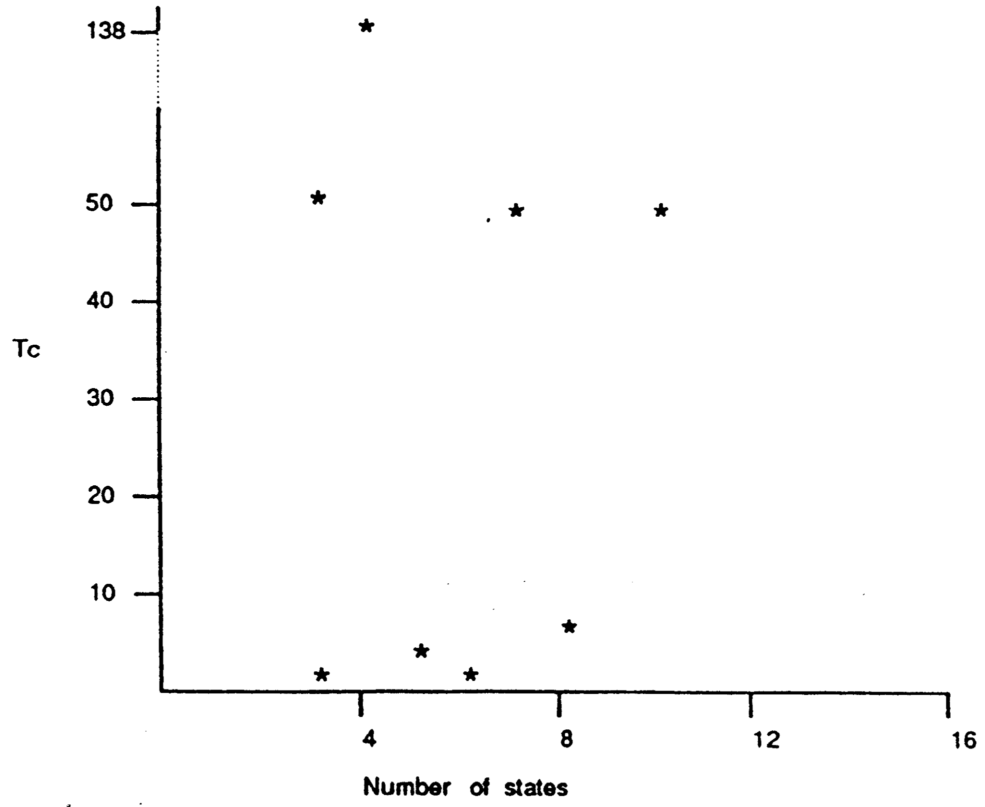


Figure 2-2: T_C and the number of states used in problem-solving.

3.1. The performance system

Soar is based on formulating all symbolic goal-oriented processing as search in problem spaces [12]. The problem space determines the set of states and operators that can be used during the processing to attain a goal. The states represent situations. There is an initial state, representing the initial situation, and a set of desired states that represent the goal. An operator, when applied to a state in the problem space, yields another state in the problem space. The goal is achieved when a desired state is reached as a result of a sequence of operator applications starting from the initial state.

Each goal defines a problem-solving context. A context is a data structure in Soar's working memory — a short-term declarative memory — that contains, in addition to a goal, roles for a problem space, a state and an operator. Problem solving for a goal is driven by the acts of selecting problem spaces, states, and operators for the appropriate roles in the context. Each such deliberate act of the Soar architecture is accomplished by a two-phase decision cycle. First, during the elaboration phase, the description of the current situation (that is, the contents of the working memory) is elaborated with relevant information from Soar's production memory — a long-term procedural memory. The elaboration phase proceeds in a sequence of synchronous cycles. During each cycle of the elaboration phase, all of the productions in the production memory are matched against working memory, and then all of the resulting production instantiations are executed, i.e., all the instantiated right-hand side actions[6]. The net

[6]Thus Soar has a degree of logical parallelism. However, we are analyzing serial implementations, so that this parallelism is not reflected at the implementation level, i.e., time/action is still (total time)/(total number of actions).

effect of these production firings is to add information to the working memory. New objects are created, new knowledge is added about existing objects, and preferences are generated.

There is a fixed language of preferences, which is used to describe the acceptability and desirability of the alternatives being considered for selection. By using different preferences, it is possible to assert that a particular problem space, state or operator is acceptable (should be considered for selection), rejected (should not be considered for selection), better than another alternative, and so on. When the elaboration phase reaches quiescence — that is, no more productions can fire — the second phase of the decision cycle, the decision procedure, is entered. The decision procedure is a fixed body of code that interprets the preferences in working memory according to their fixed semantics. If the preferences uniquely specify an object to be selected for a role in a context, then a decision can be made, and the specified object becomes the current value of the role. The decision cycle then repeats, starting with another elaboration phase.

If, when the elaboration phase reaches quiescence, the preferences in working memory are either incomplete or inconsistent, an impasse occurs in problem solving because the system does not know how to proceed. When an impasse occurs, a subgoal with an associated problem-solving context is automatically generated for the task of resolving the impasse. The impasses, and thus their subgoals, vary from problems of selection (of problem spaces, states, and operators) to problems of generation (e.g., operator application). Given a subgoal, Soar can bring its full problem-solving capability and knowledge to bear on resolving the impasse that caused the subgoal. When impasses occur within impasses, then subgoals occur within subgoals, and a goal hierarchy results (which therefore defines a hierarchy of contexts). The top goal in the hierarchy is a task goal: such as to recognize an item. The subgoals below it are all generated as the result of impasses in problem solving. A subgoal terminates when its impasse (or some higher impasse) is resolved.

3.2. Chunking in Soar

Chunking [6] in Soar is a learning mechanism that acquires new productions that summarize the processing that leads to results of the subgoal. The conditions are based on those aspects of the pregoal situation that were relevant to the determination of the results. Relevance is determined by using the traces of the productions that fired during the subgoal. Starting from the production trace that generated the subgoal's result, those production traces that generated the working-memory elements in the condition elements are found, and so on, until elements are reached that existed prior to the subgoal. Productions that only generate preferences do not participate in this backtracking process — preferences only affect the efficiency with which a goal is achieved, and not the correctness of the goal's result.

An example of this chunking process is shown schematically in Figure 3-1. The circled letters are objects in working memory. The two vertical bars mark the beginning and ending of the subgoal. The objects to the left of the first bar (A, B, C, D, E, and F) exist prior to the creation of the subgoal. The objects between the two bars (G, H, and I) are internal to the subgoal. P1, P2 and P3, and P4 are production traces; for example, production trace P1 records the fact that a production fired which examined objects A and B and generated object G. The highlighted production traces are those that are involved in the backtracing process.

Chunking in this figure begins by making the result object (J) the basis for the action of the chunk. The condition finding process then begins with object J, and determines which production trace produced it — trace P4. It then determines that the conditions of trace P4 (objects H and I) are generated by traces P2 and P3, respectively. The condition elements of traces P2 and P3 (objects C, D, E and F) existed prior to the subgoal, so they for the basis for the conditions of the chunk. The resulting chunk is:

```
C & D & E & F --> J
```

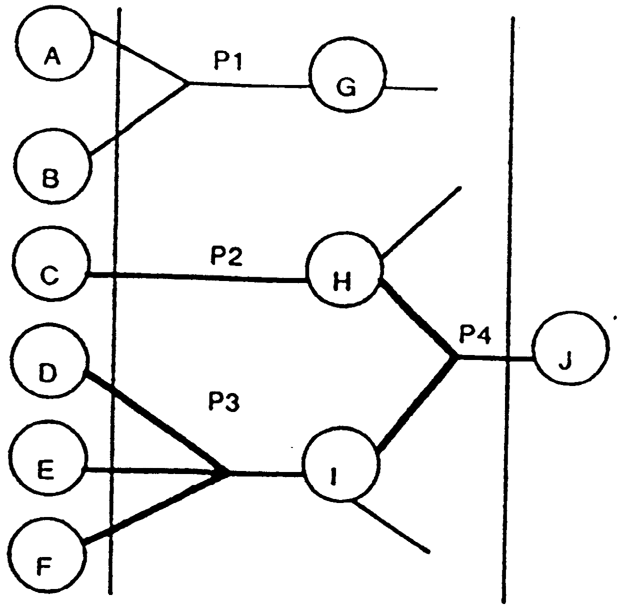


Figure 3-1: Schematic view of the chunking process in Soar.

Once a chunk has been learned, the new production will fire during the elaboration phase in relevantly similar situations in the future, directly producing the required information. No impasse will occur, and problem solving will proceed smoothly. Chunking is thus a form of goal-based caching which avoids redundant future efforts by directly producing a result that once required problem solving to determine.

4. The Matcher

Soar uses the Rete [2] algorithm for matching productions. The next subsection briefly describes the Rete matching algorithm. Using this description, we develop a model of the matcher which we will use throughout the rest of this paper to analyze expensive chunks. The important features of current production-system matching algorithms that are relevant in the analysis of expensive chunks are captured by this model and therefore our analysis here is independent of the matching algorithm used.

4.1. The Rete matching algorithm

Rete is a highly efficient matching algorithm for production systems. The Rete algorithm gains its efficiency from two optimizations. First, it exploits the fact that only a small fraction of working memory changes each cycle, by storing results of matching from previous cycles and using them in subsequent cycles. Second, it exploits the commonality between condition elements of a production (both within the same production and between different productions) to reduce the number of tests that it has to perform to do match. It does so by performing common tests only once.

The Rete algorithm uses a special kind of a data-flow network compiled from the left hand side of productions. To generate a network for a production, the compiler begins with the individual condition elements in the left-hand side. For each condition elements it chains together test nodes that check for the intra-condition constraints that have to be satisfied by that condition. Each node in the chain performs one such test. Once the algorithm has finished with the individual condition elements, it adds nodes that check for consistency of variable bindings across the multiple condition elements in the left hand side. Finally the algorithm adds a special terminal node to represent the production corresponding to this part of the network.

The production system in Figure 4-1 illustrates the Rete algorithm. The production system consists of one production *length-2*, consisting of three condition elements and one action element. The working memory elements (wmes) in Figure 4-1 describe the map shown in the figure.

```
(Production length-2
(goal ^goto-point x ^from-point y)
(exists-path ^from y ^to z)
(exists-path ^from z ^to x)
-->
(write exists-path of length 2 from x to y))

/** The working memory **/
(exists-path ^from B ^to C)
(exists-path ^from B ^to D)
(exists-path ^from B ^to E)
(exists-path ^from C ^to A)
(exists-path ^from D ^to A)
```

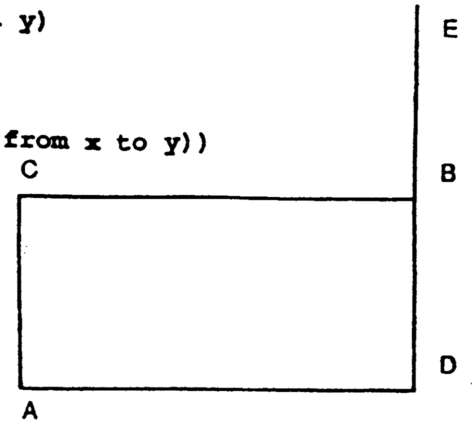


Figure 4-1: An example Production System

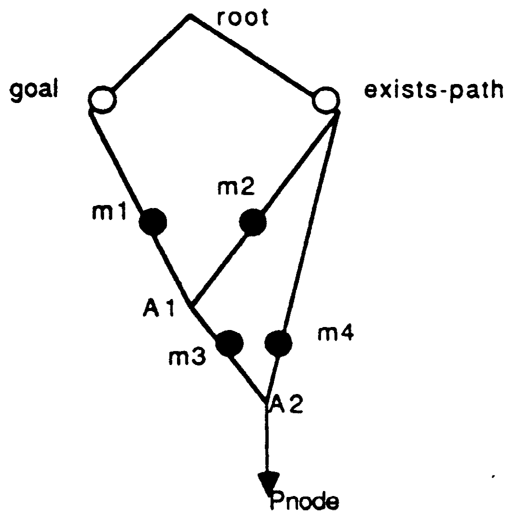


Figure 4-2: Rete net for production length-2

The Rete network built for the production *length-2* is shown in Figure 4-2. The branch of the network shown along path-1 tests for wmes matching the first condition element. Since conditions 2 & 3 have common tests only one branch along path-2 exists for both of them. The nodes marked as *m* nodes are the memory of the network which store the match information, e.g., m2 stores the wmes that matched condition 2. Rete thus saves state in the memory nodes. The functions of other nodes will become clear as we go along.

Consider the wme *(exists-path from B to E).* The first attribute of this wme equals *exists-path*, the second attribute is *from* and the fourth attribute is *to*. It will therefore pass the tests shown along path-2 and get stored in the memories m2 and m4 (which stores wmes that have matched the 3rd condition element of the production). All the other existing wmes will also get stored in m2 and m4. These wmes form the *right-activations* of nodes A1 and A2.

Suppose now a wme *(goal goto-point A from-point B)* is created. It will travel along path-1 and get stored in m1. The node A1 tests if the variable bindings for wmes in m1 and m2 match each other. A1 then creates *tokens* for matching wmes. Tokens indicate what conditions have matched and under what bindings. Thus three tokens will be created at A1: (2; x = B, y = A, z = C), (2; x = B, y = A, z = D) and (2; x = B, y = A, z = E). The number 2 indicates that the token has matched the first 2 condition elements. These tokens will be stored in m3. The node A2 will test wmes stored in m4 with the tokens in m3 and create 2 new tokens (3; x = B, y = A, z = C) and (3; x = B, y = A, z = D). This will create two instantiations for the production *length-2*. Thus the two instantiations generated required the creation of 5 tokens. The next subsection presents a high-level view of this activity in the matcher.

4.2. The cost of matching productions

In the previous subsection we described the flow of tokens in matching the production *length-2*. In [5], the cost of matching an expensive production has been attributed to two effects associated with the flow of tokens in the Rete network, shown in Figure 4-3.

- *The Long-chain effect:* If a production has a large number of condition elements, then that leads to a long chain of token creation i.e. one token creation leads to the creation of a successor token, which in turn causes the creation of its successor token & so on.
- *The Cross-product effect:* If a single token flowing into a two input node finds a large number of elements in the opposite memory with consistent variable bindings, then it generates a large number of tokens to be processed at the successor two input node.

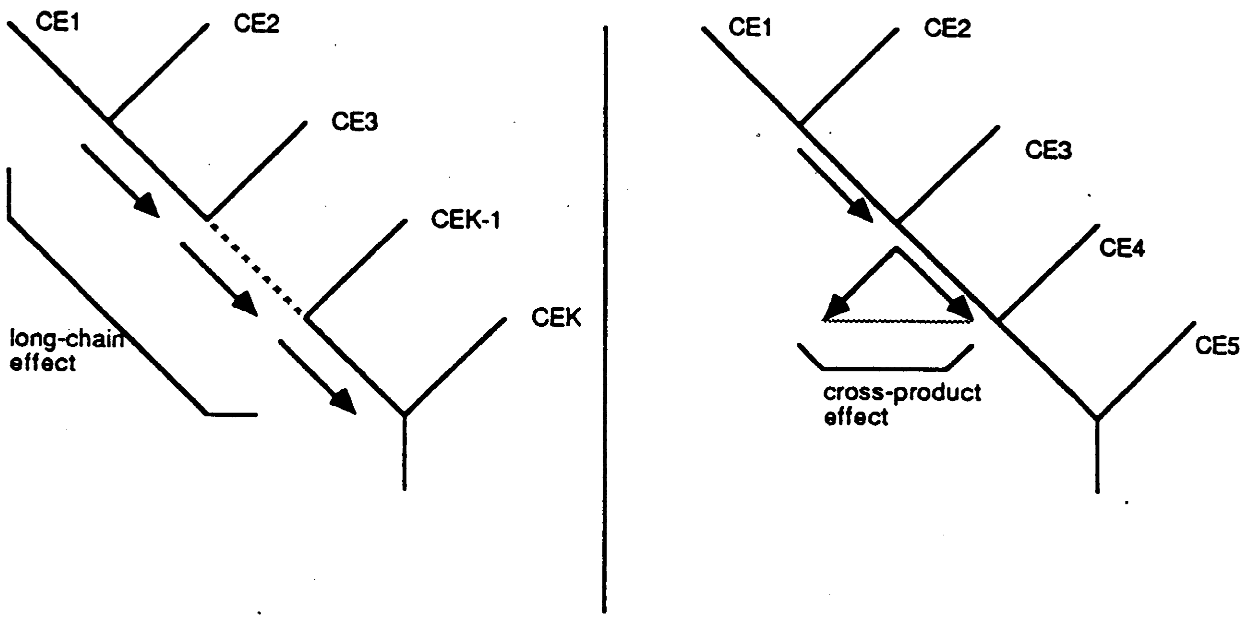


Figure 4-3: The token flow model.

A combination of these two effects can be used as a model for making qualitative predictions about the cost of a production e.g. the production length-2 is cheap since it does not have either a long-chain or a cross-product. We will refer to this model as the *token flow* model.

We will, however, use a somewhat different model to analyze expensive chunks, which subsumes the token-flow model and can make approximate quantitative predictions about the cost of a production. This model seems to provide a better insight into the activity of the matcher for the purpose of analyzing expensive chunks. The alternate model is developed from two observations:

- Measurements on Soar/PSM-E indicate the time spent in the match per token generation i.e. time/token, is approximately a constant (about 1 ms per token).[7] This provides a constant-operation base for the computational effects, so that the analysis can be independent of physical machines.
- The tokens generated in the Rete network create a *match tree* of tokens. In the previous subsection we described how 5 tokens are generated in matching the production length-2. These tokens generated in the Rete network create a match tree as shown in Figure 4-4.

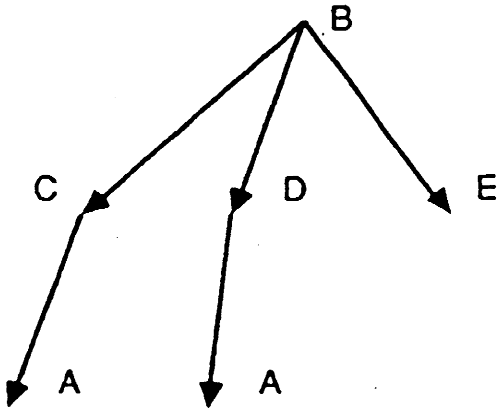


Figure 4-4: The match tree of tokens.

Therefore, for Soar productions, the number of tokens in the match tree is a good estimate of the work done in performing match. This match tree represents the search done by the Rete net to match productions. This search is done in the knowledge base of the system, so we will call it *k-search*, to distinguish it from the search done in the problem space.

The state saving in the Rete net makes sure that a partially done k-search from previous cycles is saved for use in the future cycles. Therefore the k-search is not repeated from scratch with addition of every working memory element (wme). Once a chunk fires, most of the state is removed and the next firing of this chunk requires a different k-search. This effect of repeating the k-search for every firing is seen in most Soar productions. Sharing makes sure that k-search for similar productions is shared. However, compared to the differences in match times we are dealing with, the speedup obtained from sharing in the Rete net is fairly limited [5, 11][8], hence for the purposes of this paper, we can ignore this effect. Thus performing k-search on each production in isolation for every firing of a production is a reasonable way of modeling the activity that goes on in the Rete net to match a production.

There are two important features of the k-search, which can be seen in the example in Figure 4-4.

- *The matcher has to find all possible solutions of the k-search*: All the possible instantiations for a production are found in matching that production. In the example shown in Figure 4-4, both the instantiations for the production *length-2* are found.
- *No heuristics are available to the matcher in performing k-search*: The matcher does not have any knowledge about the semantics of the production it is trying to match. For example, the Rete matcher in Figure 4-2 does not have any knowledge about the objective of matching the production *length-2*, i.e. of finding a path of length 2. Further, no other knowledge such as the maximum number of instantiations for a production or the maximum number of different bindings for a variable is available to the matcher.

[7]The time/token before chunking is usually more than the time/token after chunking due to the large number of right activations before chunking, that do not produce any new tokens. In general, the maximum variation in time/token has been seen to be about a factor of two.

[8]In [5], a speedup of 1.63 is reported due to sharing, while [11] reports a factor of 1.1. But in any case, the speedup is much smaller than the order of magnitude differences we are dealing with.

Therefore, the matcher cannot prune its k-search.

The matcher is therefore forced to perform k-search by doing an exhaustive search to find all possible ways of instantiating productions. The features of the k-search performed by the Rete matcher are not unique to the Rete matcher alone, but are also seen in all the other match algorithms that are known to the authors, such as Treat [11], which saves less state than Rete, and Oflazer's algorithm [13], which saves more state than Rete. A naive algorithm that matches all the productions with all the wmes at every step, (does not save any state at all) also has these features. Thus, this model embodies the features of these algorithms essential for our analysis. Therefore we can use it with an understanding that the analysis based on this model will hold true for all these algorithms.

5. Expensive Chunks: The internal view

As noted in the previous section, the k-search finds all solutions and proceeds without any heuristics. The following example demonstrates how the nature of k-search causes chunks to be expensive.

In this example, a Robot has to move from point A to point B. The various edges connecting A and B are shown in Figure 5-1. The operators available to the Robot allow it to move from a point X to a point Y if points X and Y are connected by path of length 1. A depth-first search is performed in the problem space to reach B, from A. The k-search tree for this task is also shown in Figure 5-1. A path of length 4 (highlighted in the figure) is found between A and B. The conditions of one of the chunks formed in performing this task will test for the existence of a path of length 4 between the source and the destination. If a path of length 4 is found, then the actions of the chunk will recommend taking the first step along that path.

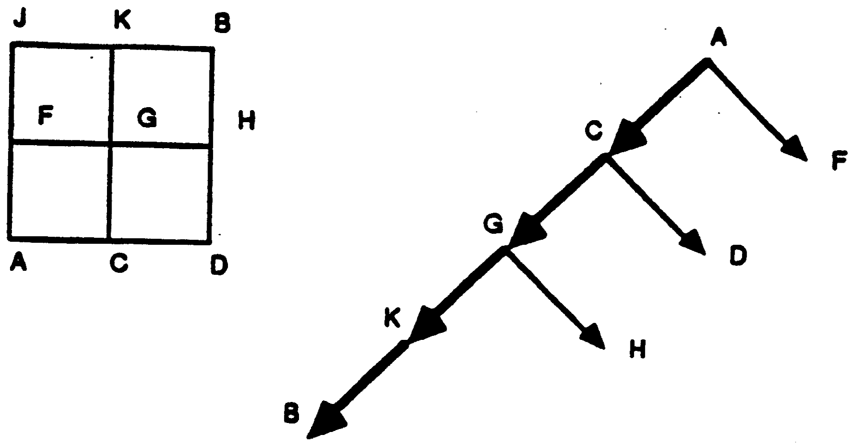


Figure 5-1: Moving from point A to B.

Given the same problem again, the chunk tries to identify a path of length 4 from A to B, giving rise to the k-search tree shown in Figure 5-2. In this example we see that before chunking 7 tokens are generated in 4 steps to solve the problem. However, the chunk consumes 19 tokens in 1 step, thus becoming more expensive than the original multi-step solution.

We refer to the above explanation as the *internal view* of the cost of a chunk. Given a chunk and the working memory, this view states that the chunk is expensive because the k-search tree of that chunk has a large number of

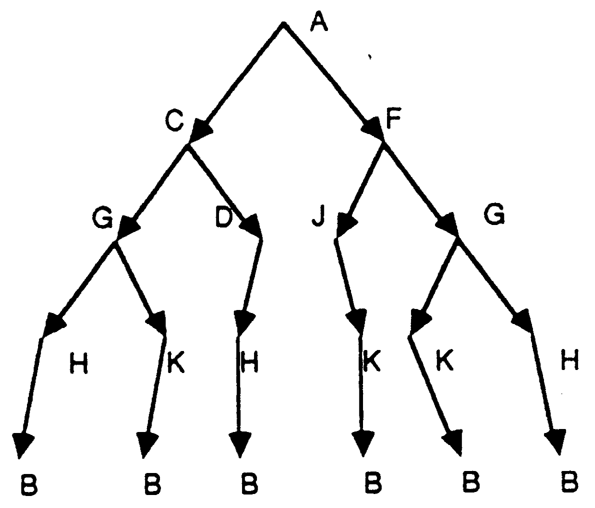


Figure 5-2: Match tree after Chunking.

tokens. We call it *internal*, because it explains the cost in terms of the k-search tree of the chunk; but it does not explain the origin of the chunk or the working memory, which define the size of the k-search tree.

The number of tokens in the k-search tree is dependent on the height of the tree, which is determined by the number of condition elements in the chunk. The number of tokens is also dependent on the breadth of the k-search tree at various levels, which in turn depends on the presence of condition elements that can match a large number of wmes that pass the test associated with that condition element. Presence of two or more such condition elements can lead to a combinatorial explosion i.e. a big increase in the breadth of the k-search tree at some level.

A third factor that determines the number of tokens in the k-search tree is the order of node expansion in the tree. The order of node expansion depends on the order of condition elements in the chunk. Figure 5-3 shows an example of how the order condition elements can impact the size of the k-search tree. The figure shows the production system of Figure 4-1, with a modified version of the map. It shows the production *length-2* and the k-search tree of the production *length-2*. The figure also shows the production *length-2-mod*, which is production *length-2* with a change in the order of condition elements. We see that the number of tokens in the k-search tree has been reduced. This shows why the order of condition elements in the k-search tree is important.

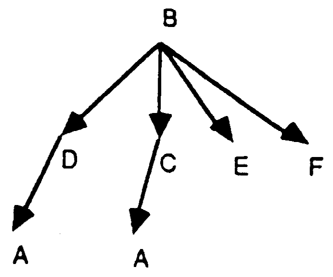
Using the internal view we have identified three factors that can cause chunks to be expensive:

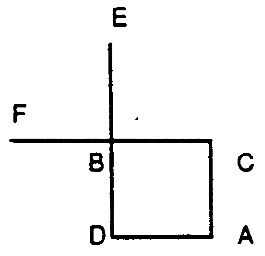
- A large number of condition elements in the chunk.
- Presence of two or more condition elements in the chunk that match a large number of wmes and increase the breadth of the k-search tree. The breadth thus also depends on the structure of the working memory.
- A bad ordering of condition elements in the chunk.

6. Expensive Chunks: The external view

In the previous section we used the *internal view* to identify three characteristics of a chunk and the working memory that determine the cost of a chunk. In this section, we identify the causes relating to things *external* to the matcher that determine these characteristics. These are aspects of the task structure and it is in their terms that one

```
(Production length-2
(goal ^goto-point x ^from-point y)
(exists-path ^from y ^to z)
(exists-path ^from z ^to x)
-->
(write exists-path of length 2 from x to y))
```





```
/** reordered production **/

(Production length-2-mod
(goal ^goto-point x ^from-point y)
(exists-path ^from z ^to x)
(exists-path ^from y ^to z)
-->
(write exists-path of length 2 from x to y))
```

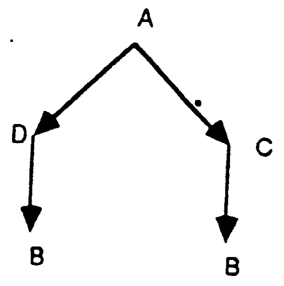


Figure 5-3: The effect of ordering condition elements.

can understand how expensive chunks arise. We refer to this as an *external view* of the cost of a chunk. We try to account for the expense of the expensive chunks we have come across.

6.1. A big footprint

If a large number of aspects of the pre-subgoal situation are examined during processing in the subgoal to produce results, then the chunk has a large number of conditions. We refer to this as a *big footprint*. The size of the footprint determines the number of condition elements in the chunk and hence the height of the k-search tree. Examples below illustrate the effect of the size of the footprint.

The first example demonstrating the effect of the footprint is the 8-puzzle task. A description of the task appears in the appendix. It has eight numbered tiles in a 3x3 frame. One cell is always blank and there is a single general operator to move adjacent tiles into the blank cell. For a given state, an instance of this operator is created for each of the cells adjacent to the blank cell. This gives rise to an impasse to select the appropriate instantiated operator to do next. The present state and desired state are given and thus form part of the pre-subgoal situation. To resolve the impasse, the instances of these operators are evaluated using comparison of the tiles in the present state and the desired state. In this formulation, the initial set of productions create an evaluation of positive 1 if a move causes a tile to go from out-of-place to in-place, negative 1 if a move causes a tile to go from in-place to out-of-place and 0 if a move causes a tile to go from out-of-place to out-of-place. In-place and out-of-place positions are determined by explicit comparison of the position of tiles in the present state and the desired state. This evaluation is used in selecting the operator to be performed next. Thus, to decide the better of two instantiated operators, the problem-solver creates a big footprint by touching all the tiles in the given and the desired states for both the operators. The big footprint leads to large number of condition elements in the chunk, causing the chunk to be expensive. The first row of Table 6-1 presents the average number of condition elements in the chunks, the number of chunks added to the system during a run of the 8-puzzle, and the increase in time/action due to chunking T_C.

Task	Avg. number of conditions in the chunks that fire	Number of chunks added	Increase in time/action due to chunking T_C (ms)
8-Puzzle	34	11	49.27
New-8-Puzzle	22	11	17.35

Table 6-1: Effect of changing representation of the 8-puzzle.

The representation of the 8-puzzle can be changed, such that explicit in-place and out-of-place augmentations (attributes) are used to describe the position of each tile relative to the desired state. Thus for any given state, the in-place or out-of-place status of each tile is known, without a comparison with the desired state[9]. Therefore, the operator selection does not require the examination of the desired state. This reduces the size of the footprint, and hence reduces the cost of the chunks. Let us call this new version of the 8-puzzle with a changed representation: *New-8-puzzle*. The second row of Table 6-1 presents the average number of condition elements in the chunks of the New-8-puzzle. The number of chunks added in performing this task can be seen to be the same as the 8-puzzle. The increase in time/action due to chunking is also shown. The time/action has reduced from 49.27 to 17.35 a factor of 2.5.

A second example demonstrating the effect of a big footprint is the Queens task, which requires placing two queens on a 3x3 board, so that no queen can take another. A forward (depth first) search is performed to obtain two positions that do not take one another. The conditions of the chunk formed test for two positions that do not take one another and then recommend these positions. Each position in the task is described using horizontal, vertical and diagonal attributes, and since these descriptions are touched to solve the problem, the chunk contains the descriptions of each position in terms of these horizontal, vertical and diagonal attributes. The description of these two positions creates a chunk with a large number of condition elements. The first row of Table 6-2 presents the number of condition elements in the chunks, the number of chunks added during a run of the Queens task and the value of T_C.

Task	Avg. number of conditions in the chunks that fire	Number of chunks added	Increase in time/action due to chunking T_C (ms)
Queens	21	3	55.14
New-Queens	6	3	1.36

Table 6-2: Effect of changing representation of the Queens task.

We can change the representation of the positions of the queens on the board by not describing them in terms of

[9]We thank John Laird for this representation.

the horizontal, vertical or diagonal attributes. Instead, each position X is augmented by *Safe* positions i.e., by positions that cannot be taken by a queen placed at X. Again the chunk formed tests for two positions that do not take one another and then recommends those positions. However, the size of the chunk is reduced, since the conditions just test if one of the positions has the other one as a *Safe* position. Let us call the Queens task with a changed representation: *New-Queens*. The second row of Table 6-2 shows the effect of changing representation in the Queens task. The number of condition elements in the chunks has reduced. This leads to a large reduction in T_C, from 55.14 to 1.36 a factor of 25.

Table 6-3 shows the average number of condition elements in the chunks in various tasks. We see that in general the expensive chunks have a larger footprint, although the separation is not large and the inexpensive waterjug has a big footprint. Since the size of the footprint explains only the height of the k-search tree, we need to examine the other factors to be able to explain all the data.

Task	Average number of conditions in the chunks that fire	Increase in time/action due to chunking T_C (ms)
8-puzzle	34	49.27
Queens	21	55.14
Path	20	48.10
Magic-Square	18	138.17
Syllogisms	15	0.91
Monkey	14	0.77
Waterjug	26	3.02
Farmer	18	6.18

Table 6-3: Number of condition elements in the chunks formed in various tasks.

6.2. Multi-Objects: Multi-attributes and Preferences

We have seen in Section 5 that one of the factors that determine the cost of a chunk is the presence of condition elements and working memory that produce a large number of tokens at a particular level in the k-search tree i.e., that cause an increase in the breadth of the k-search tree. Before proceeding to understand how this breadth arises in Soar, it is necessary to make a brief digression into the structure of condition elements and working memory in Soar. A detailed description of both can be found in [7].

Soar uses a variation on OPS5 [1] as the basic representational scheme of working memory and productions provided in OPS5. In Soar, there are two different types of data representations in working memory: *objects* and *preferences*. Certain restrictions of the OPS5 scheme force Soar to represent objects using multiple wmes, each containing only four fields: name (or class) of the object, the identifier of the object, and one attribute-value pair for that object. Thus a block, with a identifier B1, with two attributes color and size will be represented as two wmes: *(Block B1 ^Color Blue)* and *(Block B1 ^Size 10)*. Thus the general form of a wme in Soar is:

```
(Object-name Identifier ^Attribute Value)
```

A condition element that matches the above kind of wmes also has four fields in it. The condition element always contains a variable in the field that matches the identifier, and sometimes contains a variable in the value field.

Among the other two fields, the object-name field and the attribute field almost always contains a constant (none of the condition elements in the chunks in this paper has a variables in those fields).

A preference is a wme that asserts the relative or absolute worth of an object for a context slot. Preferences are special wmes of length nine and the condition elements that match preferences are also nine elements long.

Given the above structure of working memory, we can now understand the source of breadth in the k-search tree in Soar. As we have seen, the breadth of the k-search tree is dependent on the order of condition elements. Soar uses a fixed condition ordering mechanism to order all its productions. The condition ordering mechanism makes sure that the identifier variable of a condition element is bound before the condition appears in the chain of conditions in the production, i.e., the identifier variable appears in the value field of some other condition preceding this condition. This chain is then anchored in the conditions matching the current goal-context. This makes sure that even if a large number of wmes related to the old goal-context stack are present in the system (as they are likely to be), they do not affect the k-search.

Consider a condition element as mentioned above, i.e., with a bound identifier variable and the only other variable possibly appearing in the value field. When a token is passed on to this condition element, the only way it can generate more than one token, introducing breadth in the k-search tree, is if there are multiple wmes that have the same symbols in the object-name identifier and attribute fields, and different symbols in the value fields. Such a collection of wmes which differ only in the value field are used in representation of sets in Soar and an attribute that has such multiple values is called a ***multi-attribute***. Thus in the Magic-square task, the state has one multi-attribute: ***squares***, with nine values. Internally, this will be represented as a collection of nine working memory elements of the form ***(State M1 ^Squares S1), (State M1 ^Squares S2)***, etc.

The only other source of breadth in the k-search tree are the preferences. Under the current ordering scheme, the identifier of a preference is not required to be bound in a condition preceding the preference. Thus many different preferences can match a condition testing a preference. If the preferences are for objects with different names or if some other easy to test characteristic of the preferences is different, then very few k-search nodes are required to select the right preference. On the other hand, if the preferences are for different instantiations of the same object, such as different instantiations of the same operator, a large number of k-search nodes are required to be expanded to select the right preference. Thus, it is only the preferences for multiple instantiations of the same object that are of any consequence. We will refer to the multi-attributes and preferences for multiple instantiations of the same object as ***multi-objects***.

If the problem-solver uses multi-objects in resolving an impasse, then the chunk formed will have condition elements that test for the presence of these multi-objects. The above analysis predicts that expensive chunks, i.e., chunks with a large breadth in their k-search tree have such condition elements, that match multi-objects. The analysis also predicts that if a chunk does not test the presence of multi-objects or tests only a few multi-objects with 2-3 values each, then the breadth of the k-search tree will be small and the chunk will be cheap.

Table 6-4 presents data on our set of eight tasks that establishes the validity of the above analysis. The first column gives the name of the task. The second column gives the maximum breadth of the k-search tree in the task i.e. the maximum number of tokens generated during the course of the run, for matching a single condition element in the chunk. Though it certainly is not a precise measure of the total breadth of the k-search tree, a comparison of the maximum breadth in expensive and cheap chunk tasks indicate that it is a good enough estimate for our analysis. The third column contains the average number of conditions, that match multi-object, in the chunks. The fourth column contains the maximum number of multi-objects that match a condition element testing their presence. The details of the representation of these tasks appear in the appendix.

Task	Maximum breadth of k-search tree	Average number of condition elements in the chunk that match multi-objects	Maximum number of multi-objects matching conditions in Col. 3
8-Puzzle	108	8	8
Queens	648	14	9
Path	440	7	24
Magic-Square	4536	4	9
Syllogisms	2	1	2
Monkey	5	0	0
Waterjug	6	1	3
Farmer	4	2	3

Table 6-4: Maximum breadth of the k-search tree in the eight Soar tasks.

Though no precise relation obtained between the maximum breadth of the k-search tree and the numbers in column 3 and 4, Table 6-4 allows us to state that expensive chunks i.e. chunks with a large breadth of their k-search tree are formed by presence of conditions matching multi-objects.[10] The above analysis of the breadth of the k-search tree is dependent on the particular condition-ordering algorithm used, where the identifier variable of a condition element is bound before the condition is used in the production. In the new version of Soar, the old state structures objects will not be maintained. Even if the ordering mechanism is then changed, multi-objects will remain responsible for the breadth of the k-search tree. The reason becomes clear if we consider the fact that even if the identifier field of the condition element is not bound, a large number of wmes will match the condition element if the symbols appearing in their object-name (first) slots and attribute (third) slots are identical. Such a collection of wmes can either be multi-objects, or can arise indirectly as a result of multi-attributes when the wmes are attribute-value statements about the elements of the set represented as multi-attributes. For example, in the case of Magic-square, such wmes contain attribute-value statements about each element of the set of squares used to represent a state. Such wmes would be of the type *(Square S1 ^ Contains Zero)*, *(Square S2 ^ Contains one)*, etc.

We have seen how the breadth of the k-search tree depends heavily on the condition ordering mechanism. In the next subsection we examine the impact of the ordering mechanism in detail.

6.3. Ordering condition elements

The ordering of the condition elements of a production determines the order in which nodes get expanded in the k-search done to match that production. A suboptimal ordering of the condition elements of a chunk can generate a large number of k-search nodes and can cause a cheap chunk to appear expensive. The problem of generating optimal ordering is, unfortunately, an NP-complete problem [21]. Therefore, in the worst case, it is necessary to examine a large number of possible orderings to come up with an optimal one.

Soar has a fixed process, the *Reorderer*, that orders the conditions of the original productions as well as the

[10]The breadth of the k-search tree in Monkeys and Bananas and Waterjug is breadth is due to the presence of preferences. However, these preferences are for operators with different names and as explained above, such preferences are not counted as multi-objects.

productions added by chunking. Since Soar productions can have a very large number of condition elements, (e.g., Cypress-Soar [19] has over 100 condition elements in some productions), guaranteeing optimality could mean examining a large number of condition orderings, causing a slow-down. To avoid this slow-down, the Reorderer sacrifices guaranteeing optimality and cuts down the number of orderings it has to search through, by using various heuristics, described in detail in [15].

To determine the factor contributed by possible suboptimal orderings to the cost of the chunks generated in the tasks examined in this paper, it is necessary to produce an optimal ordering for those chunks. We therefore created another Reorderer that would perform an exhaustive search and produce an optimal ordering. This new Reorderer was given complete information about cost of matching each individual condition element in the chunk — the number of wmes that condition element will match, if none of its variables is bound; or if one of its variables is bound; if both the variables in the condition element are bound by conditions that ,are already ordered. An exhaustive search would require examining a very large number of condition orderings, therefore we used the following heuristics to cut down the search for an optimal ordering.

- If the cheapest condition element — one that matches the least number of wmes given the variable bindings in the conditions ordered so far — does not expand the size of the k-search space, then that condition element is chosen first. The resulting ordering is guaranteed to be optimal [17].
- If the cost of a partial order exceeds the current minimum, then that partial order is not examined any further (Branch and Bound).
- The most expensive condition element in the set of condition elements that remain to be ordered is never chosen to be the next one in the ordering. According to [17] this heuristic preserves optimality.
- Conditions relating to the current goal context are ordered as the first few condition elements of the chunk. In the tasks examined in this paper, these conditions match only single wmes and therefore placing them at the beginning is justified by the first heuristic.

Thus if the estimates of the cost of matching individual condition elements are correct, the new Reorderer is guaranteed to produce an optimal order. However, producing exact estimates is difficult, because Soar does not remove wmes relating to the old state structures, and as the problem solving progresses, the number of such wmes keeps increasing. Our estimates are based on the number of wmes related to a single state; but, since conditions testing for the current state are placed at the beginning of the chunks and since the tasks examined in this paper use very few states in problem-solving after chunking, the effect of the old state structures should be extremely small.

Table 6-5 shows the result of the new ordering algorithm. The first column gives the name of the task. The second column gives the average number of condition elements in the chunks in that task. The third column gives the original value of T_C. The fourth column gives the increase in time/action when the new reordering is used. This is denoted by T_{CNew}. The fifth column gives the speedup due to the new reordering.

The cheap chunk tasks show no speedup due to the new reordering algorithm. We have seen in previous subsection that the breadth of the k-search tree in the cheap chunks is very low. Reordering therefore cannot make any measurable difference in these tasks, since very little further reduction is possible in the breadth of the k-search trees of the chunks, in these tasks. The number of condition elements in the chunks in New-8-Puzzle is small compared to the 8-puzzle and hence the gain due to reordering is less. The same explanation holds for the Queens task.

In the expensive-chunk tasks, we can see that about 50% to 75%of the cost is attributable to a bad ordering by the old Soar Reorderer. This is because, due to multi-objects and a large number of condition elements, whenever the heuristics for ordering chunks do not work, a heavy price is paid.

Task	Avg. number of condition elements	T_C Old Reorder (ms)	T_{CNew} New Reorder (ms)	Speedup T_C/T_{CNew}
8-Puzzle	34	49.27	15.93	3.09
Queens	21	55.14	29.49	1.86
Path	20	48.10	13.06	3.68
Magic-Square	18	138.17	10.17	13.58[11]
New-8-Puzzle	22	17.35	9.05	1.91
New-Queens	6	2.18	1.55	1.40
Syllogisms	15	0.91	0.91	1
Monkey	14	0.77	0.77	1
Waterjug	26	3.02	3.02	1
Farmer	18	6.18	6.18	1

Table 6-5: Effect of using the new reordering algorithm

7. Discussion

We have addressed the issue of expensive chunks in this paper. It was necessary to understand the activity performed by the matcher in order to analyze expensive chunks. We therefore presented a model of the matcher, which shows that the cost of a chunk will be determined by the number of condition elements in the chunk, the presence of condition elements in the chunk matching a large number of working memory elements and the ordering of conditions. We have shown that the size of the footprint affects the cost of a chunk by determining the number of condition elements and that the condition elements matching large number of multi-objects are the only possible source of a combinatorial explosion. We have also shown that the Soar Reorderer manages to produce a fairly optimal ordering of condition elements.

We now address two issues raised by the k-search model and our analysis of expensive chunks:

- Can some sources of knowledge be made available to the matchers or can some restrictions be placed on the matcher so that the matchers do not have to perform an exhaustive search to find all instantiations of a production?
- Will such *smart* matchers deal with expensive chunks effectively?

7.1. Smarter matchers

Performing a better k-search to match productions does not necessarily require understanding the semantics of the production. For example, one could use a scheme such as selective backtracking [14] to prune the search. Other sources of knowledge such as the number of possible instantiations of a production or the number of possible variable bindings for a solution could also be used to limit the search. A method for cutting off reasoning when all of the answers to a problem have been found is described in [16].

[11]The extremely large speedup in the Magic-Square is because the original ordering required a very large amount of tokens, which cluttered up some hash tables used in our implementation, causing almost a 3 fold increase in the time/token as compared to other tasks. The new reordering lowers the cost of time/token besides reducing the number of individual tokens.

An obvious source of knowledge is the occurrence of variables on the left hand side that do not appear on the right hand side. Soar treats its working memory as a set. Therefore, firing two instantiations of a production carrying two different bindings for a variable on the left hand side that does not appear on the right hand side results in putting into working memory the results generated by the actions of only one of those instantiations.[12] This effect can only occur in tasks with multi-objects and is in fact seen in three of the four expensive chunk tasks examined in this paper. Table 7-1 shows the impact of this effect when it is at its worst. The first column contains the name of the task. The second column presents the number of chunk instantiations in one particular elaboration cycle. The third column presents the number of instantiations of that chunk that actually produce an action, in that cycle. The other instantiations are fired without producing any actions. Some improvement in the execution time can certainly be expected by preventing such extra instantiations.[13]

Task	Number of instantiations generated	Number of instantiations that produce actions
Queens	64	8
Path	10	1
Magic-Square	463	9

Table 7-1:. Left-hand side variables that do not appear in the right-hand side: worst case situations.

It is clear that some changes should occur in the matcher, because the requirement of finding all solutions causes the matcher to perform more work than is necessary to solve the problem in some cases. This effect was seen in the example from the Path problem in Section 5, where, after chunking, all paths from point A to point B were found despite the fact that finding only one path was all that was required. This effect is also seen in the 8-puzzle task — in a particular situation, the best operator among four available operators is to be found; but after chunking, 8 possible relations among the 4 operators are delivered by the matcher — it may certainly have been possible to work with a fewer relations. In the Magic-square task, the chunk presents as many positions as are available for placing the next number, when only 1 would have been good enough. Similar situation also arises in the Queens task, where only one placement of the 2 queens on the 3x3 board is to be found; but after chunking, the matcher delivers all 8 positions of placing the first queen on the 3x3 board and after placing the first queen, 2 positions for placing the second queen are found. It is only the expensive-chunk tasks that show this effect, because the presence of multi-objects in these tasks generates situations where the chunk can apply to more than one object. Such extra work can be avoided by limiting the number of possible instantiations per production, possibly to only one instantiation. Limiting the number of instantiations would certainly help in reducing the search done to match productions.

Though such improvements in the matcher will reduce the cost of expensive chunks, it will clearly introduce more complexity into the matcher, incurring overheads. Thus smarter matchers can exist at the expense of an increase in the complexity of their operation.

[12]The case of an extra variable on the right hand side that would allow the two instantiations to fire and produce results is not included.

[13]A mechanism to prevent such extra instantiations will be introduced in the next version of Soar.

7.2. Can Smart Matchers Eliminate Expensive Chunks

If very smart matchers really existed, then they would certainly reduce the cost of the expensive chunks in the examples we presented. However, they may not eliminate those cases. In fact, it can be proved that the smarter matchers will not be able to eliminate all expensive chunks. This can be illustrated by an example involving directed graphs.

Given a directed graph, the task is to compute the transitive closure of that graph. Assume initially, the system has solved no graph problems and therefore cannot provide the transitive closure of the given graph. An impasse is reached, and a subgoal is created to solve the problem. This subgoal chooses a problem space called *fundamental.* There is one operator available in this problem space: *add-edge.* Given that there is an edge from node A to node B, and an edge from node B to node C, the add-edge operator adds an edge from A to C (if A to C does not already exist). Figure 7-1 shows the effect of the application of the operator *add-edge.*

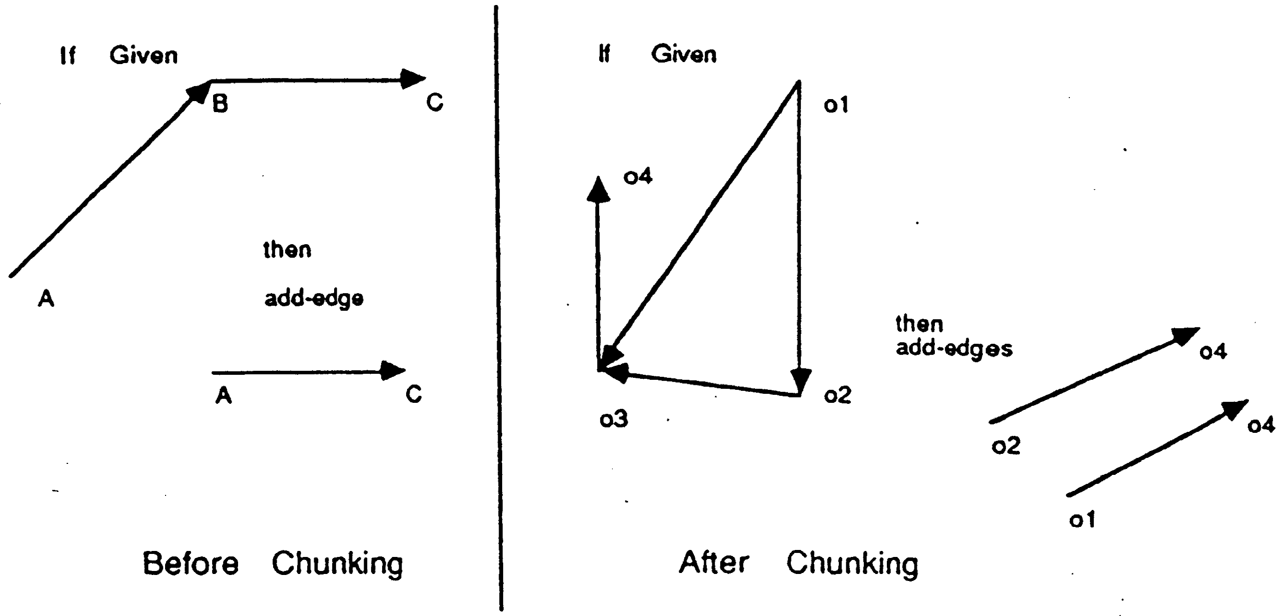


Figure 7-1: The transitive closure problem: The operator before chunking and the chunk formed.

When the transitive closure is complete, no more operators can be selected and so an impasse is reached. This means that the task is completed. This task structure fits our template of the task structures that generate expensive chunks perfectly. The domain is full of multi-objects, in fact only one type of object exists: *edge.* Every edge is described in terms of the two vertices it touches. The entire context from the original description is touched, since for every two edges, a third edge is added. The chunk created is *very* expensive. It encodes in its lefthand side a description of the graph in terms of the edges and the two vertices for edge. Figure 7-1 shows the effect of firing the chunk. Given any graph, this chunk will try to match that graph. This, however, is subgraph isomorphism on digraphs, which is a well-known NP-complete problem [4]. We have thus transformed a polynomial-time problem ($O(N^3)$ number of tokens would be generated before chunking), to an NP-complete problem. The chunk has as its action a description of the closed graph. The NP-completeness shows that, in the worst case, no fundamental improvement in the k-search is possible. This is hardly the only task of this type. Consider another problem on directed graphs, this time to add an edge in the opposite direction of every existing edge in the graph. If the problem-spaces are set up in the same way as above, we would again get a chunk that encodes in its left hand side the entire graph. These are not isolated examples. Many such cases can be constructed where an arbitrary sized footprint would lead to the formation of extremely expensive chunks. Smart matchers will not be able to deal with these chunks in the worst cases.

The chunks generated in these artificial tasks and in the naturally occurring expensive-chunk tasks examined in this paper cannot be dismissed as results of peculiar representations. If Soar is to build its own representations and keep building chunks using those representations, then it is important to guarantee that such extreme slow-downs do not occur under any circumstances.

This analysis suggests that ultimately, it may be necessary to modify the computational model employed by Soar in order to deal with expensive chunks effectively. Such modifications could appear in several places: matching (the matcher may not complete the match every time); chunking (some chunks may not be created); or representations (some problem spaces that cause expensive chunks may not allowed). However, it is important to understand all the different computational effects before modifications are made to the computational model. Thus, the research of this paper is just a first installment on finding a better computational model.

8. Acknowledgements

We thank Lanny Forgy, Anoop Gupta, John Laird, Steve Minton, David Steier, Paul Rosenbloom and Gregg Yost for providing very useful comments on earlier drafts of this paper. We also thank members of the Production System Machine (PSM) group — Brian Milnes, Dirk Kalp and Ken Hughes — for many interesting discussions on this topic.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order Nbr. 4976, Amendment 20, under contract number F33615-87-C-1499, monitored by the Air Force Avionics Laboratory and by the Encore Computer Corporation. The views and conclusions contained in this document are those of the authors and should not be interpreted as represeting the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Appendix I. A Description of the Eight Tasks

Eight Puzzle

Problem-statement: There are eight numbered movable tiles in a 3x3 frame. One cell of the frame is always blank, making it possible to move an adjacent tile into the blank cell. The problem is to transform one configuration to a second by moving the tiles.

States: The state is described in terms of nine bindings each of which connects a cell from a static 3x3 structure of cells to a tile from a dynamic structure of individual tiles. There is only one operator: move-tile. The instances of this operator are the only instantiated operators and there can be up to four of them at a time. These instantiated operators move the dynamic structure around until the desired configuration is reached.

Queens

Problem-Statement: Placing two queens on a 3x3 chess board such that no queen takes another.

States: The states are represented as a 3x3 array of positions. Each position has one horizontal, one vertical and two diagonal attributes. There is only one operator: place-queen. Up to nine instantiations of this operator may be created at a time.

Path

Problem-statement: Find a path between two points on a 4x4 grid.

States: There are 24 paths that connect nodes. There is only one operator: goto . There can be up to four instantiated goto operators at one time.

Magic-Square

Problem-statement: Completing the 3x3 magic-square.

States: A state has nine bindings that associate a number with a square. Thus we have nine squares each of which is 0 initially. There is only one operator and it can create up to nine instantiations.

Syllogisms

Problem-statement: A syllogism is a logic puzzle of two assertions involving pairs of terms (e.g. All P are Q; All Q are R) from which some conclusion (in this example : All P are R) is to be drawn. The problem-solving is done using mental models.

States: Each state is made up of two premises or statements, one model built out of two to three objects and the focus (on one object in the model). There are four different operators that can add an object, focus on a premise, focus on an object, and augment an object. But these are operators with different names.

Monkeys and Bananas

Problem-statement: A monkey has to get the bananas hung from the ceiling in a room. A ladder is available and the Monkey can move around.

States: The position of the Monkey in terms of its position on the ground and its height, and the position of the bananas. There are five different operators are available for the Monkey to climb the ladder, eat the bananas, get the bananas, climb down the ladder, and move.

Waterjug

Problem-statement: Given a five gallon jug and a three gallon jug, how can precisely one gallon of water be put into the three gallon jug. There is a well nearby, but no measuring devices are available, other than the jugs themselves.

States: The amounts of water in the five gallon and the three gallon jug. Six operators are available, one for each combination of pouring water between the well and the jug. Each of them specifies what container it is pouring water to and what it is pouring water from. Since each jug is named after the amount of water it contains, separating out the desired operator from the collection of six available operators does not require much effort.

Farmer

Problem-Statement: A farmer has to cross a river with a wolf, a sheep and some cabbage. There is a boat that can carry him and one more load at a time. The problem is to take the wolf, the sheep and the cabbage across without letting the wolf eat the sheep or the sheep eat the cabbage.

States: The status of four different objects: The farmer, the sheep, the wolf and the cabbage. Two operators are available, but only three or four instantiations of the two operators combined are available in any one state.

References

[1] Forgy, C. L.
OPS5 User's Manual.
Technical Report, Carnegie Mellon University Computer Science Department, 1981.

[2] Forgy, C. L.
Rete: A fast algorithm for many pattern/many object pattern match problem.
Articificial Intelligence 19:17-37, 1982.

[3] Forgy, C. L.
The OPS83 Report.
Technical Report 84-133, Carnegie Mellon University Computer Science Department, May, 1984.

[4] Garey, M. R. & Johnson, D. S.
Computers and Intractability: A guide to the theory of NP-completeness.
W. H. Freeman and Company, San Francisco, CA, 1978.

[5] Gupta, A.
Parallelism in Production Systems.
PhD thesis, Carnegie Mellon University, March, 1986.

[6] Laird, J. E., Newell, A., & Rosenbloom, P. S.
Towards chunking as a general learning mechanism.
In *Proceedings of AAAI-84.* Austin , Texas, 1984.

[7] Laird, J. E.
Soar User's Manual.
Technical Report, Xerox Palo Alto Research Center, 1986.

[8] Laird, J. E., Newell, A., & Rosenbloom, P. S.
Soar: An architecture for general intelligence.
Artificial Intelligence 33:1-64, 1987.

[9] Minton, S.
Selectively generalizing plans for problem-solving.
In *Proceedings IJCAI-9*, pages 596-599. Los Angeles , CA, August, 1985.

[10] Minton, S., Carbonell, J. R., Etzioni, O., Knoblock, C., Kuokka, D.
Acquiring effective search control rules: Explanation-based learning in the Prodigy system.
In *Proceedings of the Fourth International Machine Learning Workshop.* Irvine, CA, 1987.

[11] Miranker, D. P.
Treat: A better match algorithm for AI production systems.
In *Proceeding of AAAI-87.* Seattle, Washington, 1987.

[12] Newell, A.
Reasoning, problem solving and decision processes: The problem space as a fundamental category.
In Nickerson, R. (editor), *Attention and Performance VIII.* Hillsdale, N.J.:Erlbaum, 1981.

[13] Oflazer, K.
Partitioning in Parallel Processing of Production Systems.
PhD thesis, Carnegie-Mellon University, March, 1987.

[14] Pereira, L. M. & Porto, A.
Selective backtracking.
In Clark, K. L. & Tarnlund, S.-A. (editor), *Logic Programming.* Academic Press, 1982.

[15] Scales, D. J.
Efficient Matching Algorithms for the Soar/Ops5 Production System.
Master's thesis, Stanford University, June, 1986.

[16] Smith, D. E.
Finding all of the solutions to a problem.
In *Proceedings of AAAI-83*. 1983.

[17] Smith, D. E. and Genesereth, M. R.
Ordering Conjunctive Queries.
Articificial Intelligence 26:171-215, 1986.

[18] Steier, D. S., Laird, J. E., Newell, A., Rosenbloom, P. S., Flynn, R., Golding, A., Polk, T. A., Shivers, O., Unruh, A. & Yost, G. R.
Varities of learning in Soar.
In *Proceedings of the Fourth International Machine Learning Workshop*. Irvine, CA, 1987.

[19] Steier, D.
Cypress-Soar: A case study in search and learning in algorithm design.
In *Proceedings IJCAI-10*. Milano, Italy, August, 1987.

[20] Tambe, M. S., Kalp, D., Forgy, C. L., Gupta, A., Milnes, B., Newell, A.,
Soar/PSM-E: Implementing Soar on the Production System Machine.
Technical Report, Carnegie Mellon University Computer Science Department, Forthcoming.

[21] Ullman, J. D.
Principles of Database Systems.
Computer Science Press, Rockville, MD, 1982.