Machine Learning, 5, 299-348 (1990)

The Problem of Expensive Chunks and its Solution by Restricting Expressiveness

MILIND TAMBE, ALLEN NEWELL
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

PAUL S. ROSENBLOOM
Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, CA 90292

Editor: Jack Mostow

Abstract. Soar is an architecture for a system that is intended to be capable of general intelligence. Chunking, a simple experience-based learning mechanism, is Soar's only learning mechanism. Chunking creates new items of information, called chunks, based on the results of problem-solving and stores them in the knowledge base. These chunks are accessed and used in appropriate later situations to avoid the problem-solving required to determine them. It is already well-established that chunking improves performance in Soar when viewed in terms of the subproblems required and the number of steps within a subproblem. However, despite the reduction in number of steps, sometimes there may be a severe degradation in the total run time. This problem arises due to *expensive chunks*, i.e., chunks that require a large amount of effort in accessing them from the knowledge base. They pose a major problem for Soar, since in their presence, no guarantees can be given about Soar's performance.

In this article, we establish that expensive chunks exist and analyze their causes. We use this analysis to propose a solution for expensive chunks. The solution is based on the notion of restricting the expressiveness of the representational language to guarantee that the chunks formed will require only a limited amount of accessing effort. We analyze the tradeoffs involved in restricting expressiveness and present some empirical evidence to support our analysis.

Keywords. Soar, chunking, explanation-based learning, expensive chunks, restricting expressiveness, utility of learning

1. Introduction

The goal of the Soar project is to build a system capable of general intelligent behavior and autonomous existence (Laird, Newell, and Rosenbloom, 1987). One central hypothesis is that chunking (Laird, Rosenbloom, and Newell, 1986), an elementary experience-based learning mechanism, can form the basis of a general learning mechanism. Soar uses a production system (rule-based system) to encode its knowledge base. Chunking creates new productions (chunks), based on the results of problem-solving, and adds them to the production system. These chunks then fire in appropriate later situations, directly producing a result in situations which once required problem-solving to determine. This chunking process is a form of explanation-based learning (EBL) (DeJong and Mooney, 1986; Mitchell, Keller, and Kedar-Cabelli, 1986; Rosenbloom and Laird, 1986).

It is already well-established that chunking improves performance in Soar when viewed in terms of the subproblems required and the number of steps within a subproblem (Steier, et al., 1987). However, despite the reduction in number of steps, there may sometimes be a severe degradation in the total run time. This problem arises due to *expensive chunks*, i.e., learned productions that consume large amounts of processing in the match. In the worst case, matching expensive chunks is NP-hard (Tambe and Newell, 1988). Expensive chunks pose a major problem for Soar, since in their presence, no guarantees can be given about Soar's performance.

This article is an investigation into expensive chunks.[1] We establish that expensive chunks exist and analyze their causes. This analysis reveals that expensive chunks are formed due to particular representations of tasks in Soar. We then present a solution to the problem of expensive chunks. The solution guarantees that chunking will create only *cheap*, i.e., inexpensive chunks, with a linear bound on the match. The central notion in the solution is to restrict the expressiveness of Soar's production system, so as to prohibit those task representations that lead to expensive chunks. As a result, tasks have to be encoded with a less expressive production system language. Thus, we trade off some amount of expressiveness for a guarantee of inexpensive chunks.[2]

Concern about degradation in performance due to learning has appeared widely in the EBL literature (Iba, 1989; Keller, 1987; Markovitch and Scott, 1988; Minton, 1985; Minton, 1988a). Various approaches have been used to deal with this degradation, most focusing on some form of a cost–benefit analysis of the learned material. In contrast, the overall goal of this work is to achieve the safety of chunking, where safety of chunking is defined as a guarantee that chunking will not degrade Soar's performance. To do this, our solution for expensive chunks focuses on reducing the cost of chunks to a negligible level. Though the solution provides no explicit guarantees about the benefits of chunking, empirical and analytical case studies of several tasks suggest that the solution does help Soar obtain performance benefits via chunking.

This article is organized as follows: Section 2 establishes that expensive chunks exist. Section 3 provides background information about Soar and its production matcher. Section 4 uses the background material to decompose the causes of expensive chunks into two components. Section 5 presents the solution for expensive chunks based on restricting expressiveness. Section 6 analyzes the tradeoffs involved in adopting the solution. Section 7 provides detailed experimental results bearing on expensive chunks. Section 8 presents a discussion of some issues related to expensive chunks. Section 9 discusses the relevance of this research to other research efforts. Finally, Section 10 summarizes the results and discusses the open issues remaining for future work.

2. The problem of expensive chunks

The problem of expensive chunks is Soar's particular version of a more general problem: the high cost of accessing learned knowledge in problem-solving systems (Minton, 1988a). This accessing cost arises from testing the applicability of the learned knowledge in a problem-solving situation. In the extreme, testing the applicability of a piece of learned knowledge can be so expensive that a problem-solver may slow down with learning, a clearly undesirable effect.

To analyze this problem in the context of Soar, we need to separate out various aspects of what happens to Soar's performance with learning. Intelligent systems such as Soar, that are based on symbolic architectures (Newell, Rosenbloom, and Laird, 1990), partition the complete system into two domains. Above the architecture is the cognitive domain of flexible symbol processing. Below the architecture is the implementation domain of fixed computational processes. In Soar, the cognitive domain consists of problem space search. It is a symbolic process that can itself be controlled by further symbol processing, i.e., the problem space search can be controlled using search control knowledge. The implementation domain, on the other hand, performs production match, i.e., it tests applicability of the productions in the knowledge base. Production match is a fixed process that runs to completion unaffected by the knowledge in the cognitive domain, i.e., search control is unavailable in the implementation domain.

Thus, the computations in the cognitive and implementation domains are quite distinct. Analogously, the phenomena in these domains that arise out of chunking are also qualitatively different in nature. Therefore, it is useful to partition the effects of learning on task performance in the two domains into two different effects: the *cognitive effect* (in the cognitive domain) and the *computational effect* (in the implementation domain). The cognitive effect is the change in the number of cognitive operations required to perform the task. The computational effect is the change in the amount of time required to perform the individual cognitive steps. For Soar, the cognitive effect of chunking is the change in the number of (right-hand-side) actions of productions that are executed, since actions are the smallest cognitive operations. The computational effect of chunking is the change in the time required per action that is executed. The implementation domain performs production match for each action in the cognitive domain. This match computation per action in the implementation domain cannot be terminated or altered by the addition of any amount of knowledge in the cognitive domain. Therefore, it is important to bound the match computation per action to guarantee the performance of the system. This constraint gives rise to the notion of an *ideal computational model*, which says that the time per action should be constant. This ideal computational model relates back to our goal of safety of chunking—achieving a constant time per action implies that chunking has not added any match cost. The computational effect is then a measure of the amount of distortion in the ideal computational model due to chunking.

Table 1 shows the effects of chunking for eight tasks implemented in Soar (a description of these tasks and the representations used in solving them appears in Appendix I). Column 1 gives the cognitive effect for the eight tasks. It is defined as the number of actions before chunking divided by the number of actions after chunking, i.e., the speedup, in number of actions, achieved due to chunking. Column 2 gives the computational effect, defined as the time-per-action before chunking divided by the time-per-action after chunking. Ideally the computational effect should be unity. Note that in calculating both the computational and cognitive effects, the before-chunking quantity is in the numerator and the after-chunking quantity is in the denominator. Thus, if the cognitive effect and the computational effects are multiplied, they provide the speedup in total match time. Column 3 gives this speedup in total match time. Column 4 gives the number of chunks added to the system in the course of the run. These measurements were done on Soar/PSM-E (Tambe, et al., 1988), a system that uses a highly optimized implementation of the Rete matcher, based on OPS83 software technology (Forgy, 1984).[3]

Table 1. Effects of chunking on performance.

Task	Cognitive Effect	Computational Effect	Total Speedup	Number of Chunks
Eight-puzzle	6.53	0.15	0.99	11
2-Queens	5.21	0.06	0.32	3
Grid	13.54	0.06	0.85	14
Magic-square	6.59	0.04	0.25	5
Syllogisms	11.59	0.89	10.27	10
Monkey and Bananas	6.20	0.83	5.16	4
Waterjug	9.13	0.57	5.22	11
Farmer	11.09	0.45	5.04	14

In all the tasks, chunking causes a large cognitive effect, i.e., it provides a big speedup in the number of actions. However, for the tasks in the upper half of the table, i.e., the Eight-puzzle, 2-Queens, Grid, and Magic-square, the speedup in terms of total match time is less than 1—the match time has actually *increased* after chunking. For instance, the Magic-square task shows a total speedup of 0.25, i.e., a four-fold slowdown after chunking. This anomaly occurs because of the computational effect, which shows that the time per action for these four tasks has increased by as much as a factor of 25 (for the Magic-square). For other tasks, e.g., Syllogisms, Monkeys and Bananas, Waterjug, and Farmer, the speedup in terms of the number of actions due to chunking is followed by a concomitant speedup in the total match time.

Thus, despite the optimized implementation, matching a few chunks causes a very large increase in time per action in some tasks, causing a gross violation of Soar's ideal computation model. These chunks are called *expensive chunks*.[4] The slowdowns caused by expensive chunks come from the large (combinatorial) match effort for individual chunks, rather than the number of chunks, or combinatoric firings of chunks. As shown later in this paper, in the set of tasks examined, this large match effort in the expensive chunks is due to the complexity of the match process rather than the total number of conditions in the chunks. Expensive chunks occur in the Eight-puzzle, 2-Queens, Grid, and Magic-square tasks, while the chunks in the other tasks are relatively cheap.

In Table 1, the four tasks containing cheap chunks are typical of cheap-chunks tasks. The four expensive-chunks tasks in Table 1 were the only ones found that exhibit expensive chunks, out of several dozen tasks in the history of the Soar project (Steier, et al., 1987). Thus, expensive chunks do not occur often.[5] But when they do occur, they clearly pose a big performance penalty for the system.

Problems similar to expensive chunks can also arise in hand-coding of productions (recall that chunks are not hand-coded, they are learned productions). The analysis presented here applies to such hand-coded productions as well. However, hand-coding allows the flexibility of a detailed analysis of the problem; the programmer involved may then restructure the task or rewrite the productions to remove inefficiencies (Brownston, Farrell, Kant, and Martin, 1985; Steier, 1986). Therefore the issues of expensive productions are not (and have not been) as imperative in hand-coding as they are in an automated process like chunking.

3. Background

The first subsection below presents a brief overview of Soar. This overview helps to ground the discussion of expensive chunks in the following sections. The overview is divided into three portions describing the performance system, the *non-penetrability of memory* assumption, and the chunking mechanism. Readers familiar with these issues may wish to skip this overview. The second subsection presents a simple model of Soar's production matcher—*the k-search model*—to free the analysis of expensive chunks from the complexities of the implementation.

3.1. Soar[6]

Soar is based on formulating all symbolic goal-oriented processing as search in problem spaces. 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 space, states, and operators for the appropriate roles in the context. Each such deliberate act of the Soar architecture is accomplished by a decision cycle that consists of two phases: an elaboration phase and a decision phase.

The elaboration phase proceeds in synchronous cycles. During each cycle of the elaboration phase, all of the productions in the production memory—a long-term procedural memory—are matched against working memory, and then all of the resulting production instantiations are fired. The net effect of these production firings is to add information to 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 (i.e., should be considered for selection), rejected (i.e., should not be considered for selection), better than another alternative, etc. 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. In this reflective process, Soar can access all of the structures in working memory and fire all of the productions which match this working memory. However, it cannot directly examine the production memory. Productions are compiled code, which is inaccessible, i.e., the production memory is *non-penetrable*. (This constraint has a major impact on the solution to the expensive chunks problem.) When impasses occur in the course of resolving other impasses, then subgoals occur within subgoals, and a goal hierarchy results.

Chunking is Soar's sole learning mechanism. It acquires new productions, called chunks, that summarize the processing that leads to results of subgoals. Chunking only creates new productions; it does not delete, modify, or replace productions. The actions of the chunk are based on the results of the subgoal. The conditions of the chunk are based on those aspects of the goals above the subgoal (the supergoals) that are 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 exist in the supergoals.

An example of this chunking process is shown schematically in Figure 1. (This is a highly simplified example of chunking. See (Laird, Rosenbloom, and Newell, 1986) for more details.) 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 in the supergoals. The objects between the two bars (G, H, and I) are internal to the subgoal. P1, P2, P3 and P4 are production traces; for example, production trace P1 records 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 in the supergoals, so they form the basis for the conditions of the chunk. The resulting chunk is:

$$C \,\&\, D \,\&\, E \,\&\, F \rightarrow J$$

(In the actual chunking process, the objects C, D, E and F would be included in the conditions of the chunk after some variabilization.) Once a chunk has been learned, the

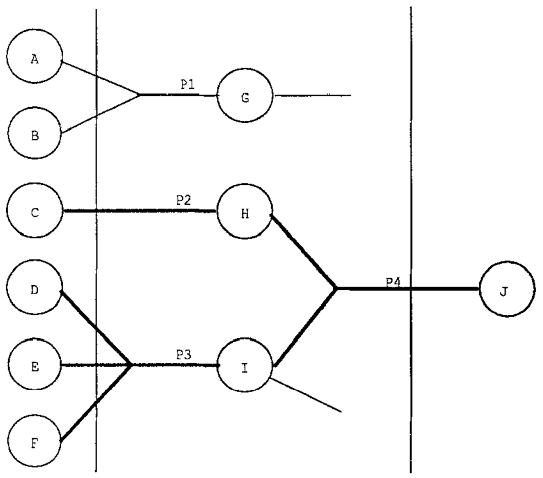


Figure 1. Schematic view of the chunking process in Soar.

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

3.2. Modeling Soar's production match

The k-search model of production system match algorithms is based on the notion of tokens, i.e., partial instantiations of productions. Tokens indicate what conditions have matched and under what variable bindings. They allow analysis of expensive chunks independent of the complexities of the physical machine or match algorithms typically used in production systems.

Consider the production shown in Figure 2(a). (This is not a real Soar production. See Figure 6 for an example of a real Soar production.) The production contains three condition elements (CEs or conditions) and one action. In the figure, up-arrows (^) indicate attribute names and angled brackets (< >) indicate variables. Figure 2(b) shows the working memory in the production system. The working memory describes the directed graph shown in Figure 2(c). Note that the conditions in the production contain variables (<x>, <y>, etc.) or constants, while working memory elements (or WMEs) can only contain constants (B, C, etc.). The production in Figure 2(a) cannot be instantiated for the working memory in Figure 2(b), since there is no match for the first CE.

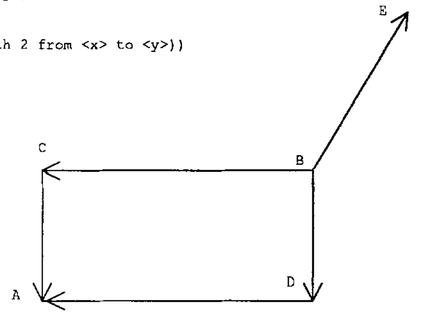
Now, suppose the WME (current-position B) is added to the system. The production will now match the working memory. While matching the production, tokens will be generated, some of which are: (2; <x> = B, <z> = C), (2; <x> = B, <z> = D), etc. The first number in the token indicates the number of CEs matched and the other elements indicate the

```
(Production::Length-2
      (current-position <x>)
      (point <x>  ^connected-to <z>)
      (point <z>  ^connected-to <y>)
   -->
      (write exists-path of length 2 from <x> to <y>))
```

(a)

```
/** The working memory **/
 (point B  ^connected-to C)
 (point B  ^connected-to D)
 (point B  ^connected-to E)
 (point C  ^connected-to A)
 (point D  ^connected-to A)
```

(b)



(c)

Figure 2. An example production system: (a) a production, (b) working memory, (c) the directed graph described by the working memory.

bindings for the variables. Thus, the first token shows that the first two condition elements were matched with the bindings B for variable <x>, and C for variable <z>.

The tokens generated in the match can be represented in the form of a *match* tree, as shown in Figure 3 (at every stage only the additional variable bindings are shown). This match tree represents the search conducted, using tokens, by the matcher in order to match the production. Since this search is done in the production system, i.e., in the knowledge base, it is called *k-search*, in order to distinguish it from problem space search.

Measurements on Soar/PSM-E (Tambe, et al., 1988) indicate that the time spent in match per token is approximately constant.[7] Therefore, for Soar productions, the *number of tokens* in the k-search tree is a reasonable estimate of the work done in performing match.

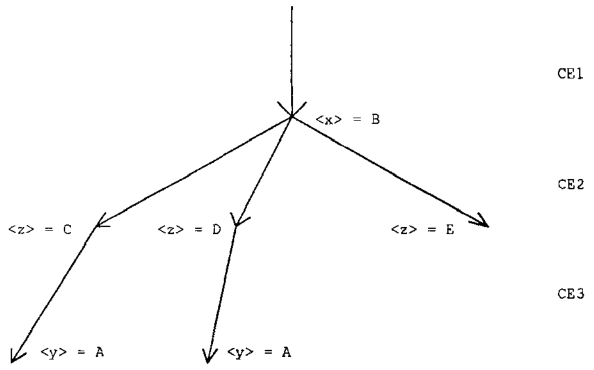


Figure 3. The match tree of tokens generated when the production in Figure 2(a) matches the working memory in Figure 2(b).

The k-search model extends to various match algorithms such as Rete (Forgy, 1982), Treat (Miranker, 1987; Nayak, Gupta, and Rosenbloom, 1988), and Oflazer's algorithm (Oflazer, 1987). There are two important optimizations done in these matching algorithms—*sharing* and *state saving*. Sharing common parts of CEs in a single production or across different productions reduces the number of tests required to do match. We do not do model sharing, since in practice, the effect of sharing has turned out to be quite limited (a factor of 1.1 to 1.6) both for hand-coded productions (Gupta, 1986; Miranker, 1987), and for learned productions (Tambe, et al., 1988). State saving accumulates partially completed k-search from previous decision/elaboration cycles for use in future cycles. Even if the WMEs generated in a cycle fail to match a production, the resulting k-search is saved. Thus, if a new WME is added in a new cycle, only the new WME has to be matched; the k-search from the previous cycles is not repeated. Typically, once the production fires, the state accumulated from previous cycles gets removed. The next firing requires a different k-search.[8] Therefore, performing k-search on each production in isolation, for every firing of a production, appears to be a reasonable way of modeling the activity of matching a production.

As shown in (Tambe and Newell, 1988), there are two key characteristics of k-search (as performed in Rete, Treat, and Oflazer's algorithm):

1. K-search does not allow heuristics.
2. K-search finds all possible solutions.

Thus, the matcher performs an exhaustive search to find all possible ways in which a production can match working memory. The number of tokens in the k-search tree generated in this process determines the cost of a production (chunk). Consider a k-search tree of depth D and a constant branching factor of B. The cost of this k-search tree in tokens is:

$$\texttt{Cost} = \sum_{k=1}^{D} B^k > B^D \texttt{ tokens}$$

Thus, the cost of this k-search tree is exponential in the depth D. A chunk with such a k-search tree is clearly an expensive chunk, since it consumes a large match effort (for large B and D).

4. Expensive chunks: The contributing factors

The previous section showed that the cost of matching a chunk is determined by the number of tokens in the k-search tree generated during the match. An expensive chunk generates a large number of tokens in the k-search tree. Two factors determine the number of tokens in the k-search tree—the height of the k-search tree and the branching of the k-search tree. An expensive chunk has a tall, branchy k-search tree. In this section, we look at these factors in detail.

4.1. Height of k-search tree

The height of the k-search tree is determined by the size of the *footprint*, where a footprint is defined as the set of WMEs in the supergoals examined during processing in a subgoal. These WMEs (after variabilization) form the conditions of the chunk that results from solving the subgoal. Thus, the size of the footprint determines the number of conditions in a chunk, which in turn determines the height of the k-search tree. Although the footprint size can have a big impact on the cost of a chunk (as shown in the example below), overall, the footprint size explains only a minor part of the expensive chunks phenomenon.

An example demonstrating the effect of the footprint can be found in the Eight-puzzle task. The representation used for this task is described in Appendix I. It has eight numbered tiles in a 3×3 frame with one blank cell. 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 apply next. To resolve the impasse, the instantiated operators are evaluated in a subgoal using comparison of the tiles in the current state and the desired state. This evaluation is used in selecting the operator to apply next. (The evaluation scheme used is that if an operator moves a tile into its location in the desired state, it is given a positive evaluation; if it moves a tile out of its location in the desired state, it is given a negative evaluation; otherwise it is evaluated as zero.) Thus, to decide the better of two instantiated operators, the problem-solver examines the tiles in the current and desired states for both the operators. This creates a big footprint, given the representation used, since the current state and desired state form part of the supergoal. Figure 4(a) shows the evaluation of one of the instantiated operators. The operator is indicated by an arrow (→). The figure shows how the tile to be moved (tile 2) is compared in the current and desired state. A similar comparison and evaluation is performed for the second instantiated operator. The big footprint generated via the two evaluations leads to a large number of CEs in the chunk (34), causing the chunk to be expensive, with a computational effect of 0.15 (recall from Section 2 that a low computational effect implies a big increase in time per action with chunking).

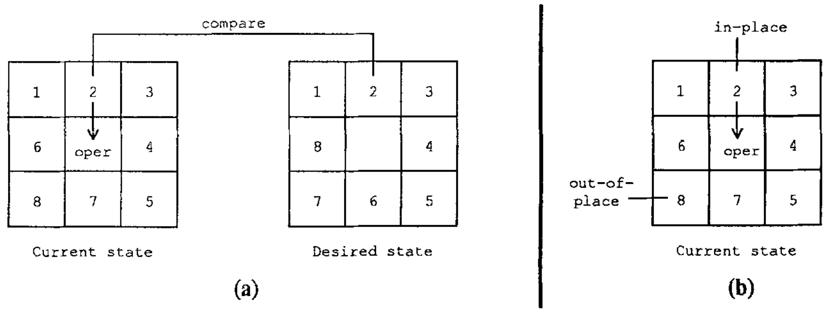


Figure 4. The footprint for the Eight-puzzle (a) when comparison of tiles in the current and desired state for evaluating an operator is required and (b) when a comparison of the tiles with the desired state is not required (in-place and out-of-place augmentations for only two of the tiles shown).

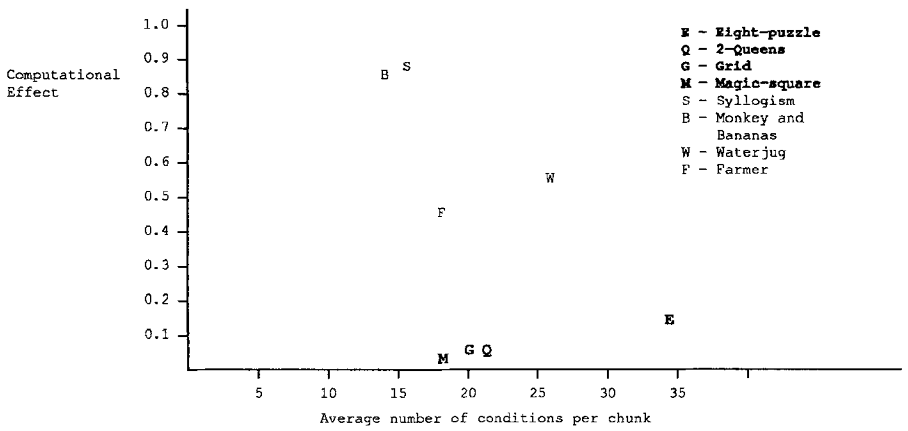


Figure 5. Average number of conditions in the chunks formed in various tasks (on the x-axis) and their corresponding computational effects (on the y-axis). The expensive-chunks tasks are highlighted.

The representation of the Eight-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.[9] Thus for any given state, the in-place or out-of-place status of each tile is known; and it is updated after every move by an explicit comparison with the desired state. Therefore, the operator selection does not always require an examination of the desired state. For instance, as shown in Figure 4(b), evaluation of the operator moving tile 2 does not require a comparison with the desired state—the operator is moving tile 2 out of its location in the desired state. For evaluating this operator, the chunk formed includes only the in-place augmentation of the current-state tile in its conditions; it does not include the conditions for the corresponding tiles from the desired state. This reduction in the size of the footprint reduces the cost of the chunks. With this changed representtion, the same number of chunks are added, but the average number of CEs in the chunks reduces from 34 to 22 and the computational effect increases from 0.15 to 0.25, i.e., the slowdown is lessened.

Figure 5 graphically depicts the relation between the number of conditions in the chunks and the computational effects for the tasks from Table 1. It shows that tasks with larger footprint size (more conditions per chunk) tend to be the expensive-chunks tasks (with low computational effects), but not always. On average, expensive chunks have a somewhat larger footprint size (23.2 CEs) than cheap chunks (18.2 CEs). However, this separation in the average footprint size is quite small (this small separation is also reflected in Figure 5) indicating that the footprint explains only a minor part of the expensive-chunks phenomenon.

4.2. Branching of the k-search tree

The branching of a production's k-search tree is a function of the number of WMEs that match each condition and the amount of constraint provided by the variable tests. All the WMEs in Soar's production system are *a priori* candidates to match a condition. This match would lead to a k-search tree with a number of tokens greater than $\#\text{WMEs}^{\text{conditions}}$. However, constants and variables bound in the conditions prior to the current condition can provide a strong filter on the set of WMEs that can possibly be bound. Most Soar conditions, referred to as object-conditions, have four fields: (*class identifier attribute value*). The condition (point <x> ^connected- to <z>) from Figure 2(a) is an example of an object-condition. Two of the fields in object-conditions are constant [class and attribute] (almost always), one of the fields is a prebound variable [identifier] (almost always), and the remaining field can be a constant, a prebound variable, or an unbound variable [value]. Here, a prebound variable is a variable bound in a previous condition. Thus, *an unbound varaible should only occur in the value field*, and branching only occurs in matching a Soar condition if there are multiple possible values corresponding to the three already fixed fields; or, in more semantic terms, if there is more than one value for an attribute.[10]

The other type of Soar conditions match preferences and are called preference-conditions. For the purposes of this article, it is not necessary to understand the details of these conditions, except that their identifier field is the only possible field which can be unbound. Thus, these conditions can give rise to multiplicity if similar preferences with distinct identifiers are present in the system.

Figure 6 presents a Soar production that demonstrates the restrictions described in the previous paragraphs. The conditions of the production test whether a block with an identifier matching <b1> is on top of a block with an identifier matching <b2>, and there is a preference for an operator called *put-down*. The single action of the production gives a best preference to the put-down operator. Conditions one through four, and six are object-conditions. Their class and attribute fields contain constants, their identifier fields contain prebound variables (except for the first condition) and their value fields contain unbound variables or constants. The fifth condition is a preference-condition. Its identifier field—the second field in the condition—is unbound.

```
              (Production::PUT-DOWN-is-best

CE1              (goal <g>  ^problem-space <p>)

CE2              (goal <g>  ^state <s>)

CE3              (state <s>   ^block <b1>)

CE4              (block <b1>  ^on-top-of <b2>)

CE5              (preference <o1>  ^role OPERATOR  ^value ACCEPTABLE
                             ^goal <g>  ^problem-space <p>  ^state <s>)
CE6              (operator <o1>  ^name PUT-DOWN)

              -->
                 (preference <o1>  ^role OPERATOR  ^value BEST
                             ^goal <g>  ^problem-space <p>  ^state <s>)
```

Figure 6. A production from the blocks world domain.

Given these restrictions on matching Soar's conditions, there remain only two sources of branching in the k-search tree: multi-objects (multi-attributes and preferences) and poor condition ordering. We discuss these two factors in detail below.

4.2.1. Multi-objects: Multi-attributes and preferences

Multi-objects refer to a combination of multi-attributes and preferences. A multi-attribute refers to a set of WMEs with multiple values for a fixed class, identifier, and attribute.[11] For instance, if there are three blocks in the blocks world, they can be represented as a multi-attribute of the state (this is also referred to as the state pointing to the three blocks):

```
(state S1 ^block B1), (state S1 ^block B2), (state S1 ^block B3)
```

Figure 7 shows how multi-objects are a source of branching in the k-search tree. Figure 7(a) shows the working memory, to be matched by the production in Figure 6. In Figure 7(a), `block` is a multi-attribute of the state, with the values `B1`, `B2`, and `B3`. The figure

```
(goal G1  ^problem-space P1)
(goal G1  ^state S1)

(state S1  ^block B1)
(state S1  ^block B2)
(state S1  ^block B3)

(block B2  ^on-top-of B1)
(block B3  ^on-top-of B1)

(preference O1 ^role OPERATOR ^value ACCEPTABLE
         ^goal G1 ^problem-space P1 ^state S1)

(operator O1  ^name PUT-ON-TOP-OF)

(preference O2 ^role OPERATOR ^value ACCEPTABLE
         ^goal G1 ^problem-space P1 ^state S1)

(operator O2  ^name PUT-DOWN)
```

(a)

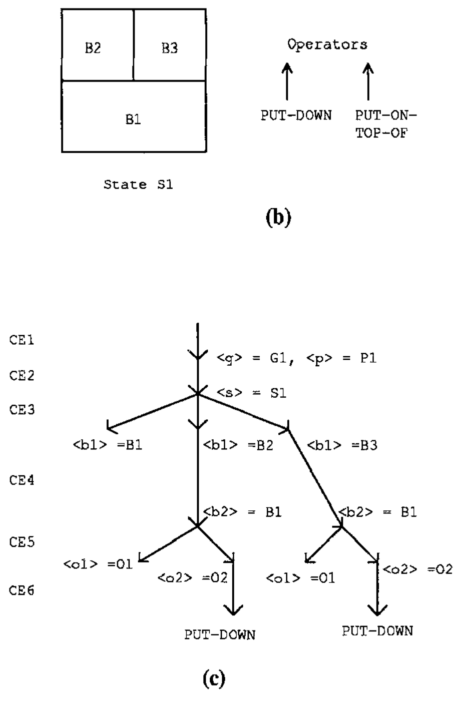


Figure 7. Blocks-world demonstration of branching in the k-search tree—(a) working memory, (b) situation represented by the working memory, (c) k-search tree for production PUT-DOWN-is-best.

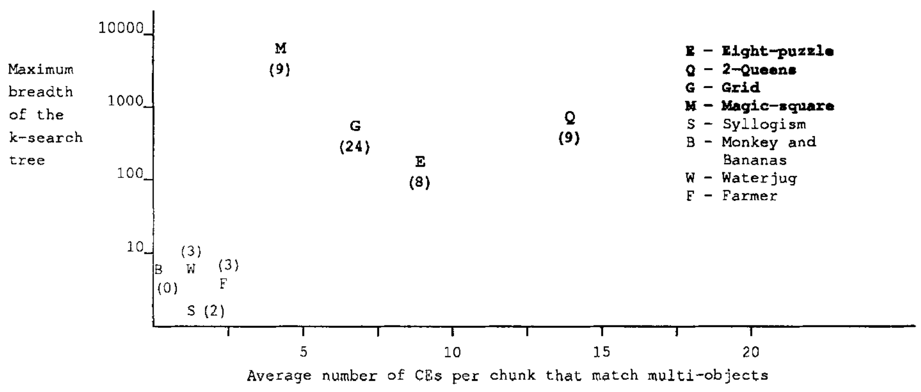


Figure 8. The impact of multi-objects on branching in the k-search tree for the eight tasks. The maximum breadth of the k-search tree is an indicator of the amount of branching in the k-search tree. The numbers in parentheses indicate the maximum number of elements per multi-object in that task representation. The expensive-chunks tasks are highlighted.

also shows two preference WMEs for the operators O1 (put-on-top-of) and O2 (put-down). Figure 7(b) shows the situation in the blocks world described by the working memory. Figure 7(c) shows the k-search tree generated in the match of the production in Figure 6. Since block is a multi-attribute of the state, the k-search tree branches at the third condition to match the blocks B1, B2, and B3. Similarly, since more than two matching preferences are presnt, the k-search tree branches in the fifth conditon. At every other point, the matcher is able to make a *unique* choice of what to match using the name of the attribute.

Thus, multi-objects are the only WMEs that can cause branching in the k-search tree. This conclusion suggests that expensive chunks, i.e., chunks with a large branching of the k-search tree, can only be formed in the presence of multi-objects. Figure 8 presents data on the eight tasks that supports the above analysis. The x-axis plots the average number of CEs per chunk that match multi-objects. The numbers in parentheses associated with each task are the maximum number of elements in any single multi-object used in the representation. The y-axis gives the maximum breadth of the k-search tree during the course of the run; i.e., the maximum number of tokens generated during the course of the run, for matching a single condition element in the chunk. The breadth of the k-search corresponds to the number of nodes at a particular height in the k-search tree, and maximum breadth is an indicator of the amount of branching in the k-search tree. The y-axis is actually plotted on a log scale. Figure 8 does not indicate any precise relation between branching and multi-objects. However, the figure shows that expensive chunks, i.e., chunks with a large amount of branching in their k-search trees, have a larger number of CEs matching multi-objects and a larger number of elements per multi-object.

This analysis shows why in the presence of multi-attributes it is possible for the matcher to generate an exponential (in the depth of the k-search tree) number of tokens. With multi-attributes, it is possible to encode a general graph structure in both working memory and productions, thus implementing a subgraph isomorphism problem, which is a well-known

NP-complete problem (Garey and Johnson, 1978). It is quite possible for chunks encoding such graph structures to be generated in Soar (Tambe and Newell, 1988). Thus, matching expensive chunks is NP-hard.

4.2.2. Condition element ordering

The second source of branching in the k-search tree is the poor ordering of condition elements. Recall that the intercondition variable tests in productions provide a constraint on the values of attributes. A poor condition ordering can reduce this constraint and thus generate a large number of tokens in the k-search.

As matching expensive chunks is NP-hard, in the extreme, the condition ordering mechanism cannot eliminate the exponential k-search. However, in practice, a good condition ordering can provide big speedups in production systems (Ishida, 1988). To investigate the impact of the ordering of CEs in the expensive chunks, an optimal ordering of CEs is helpful. The problem of generating an optimal ordering is, however, itself NP-complete (Ullman, 1982). Currently, Soar has a simple ordering algorithm (Scales, 1986), that orders the CEs of the original productions and the chunks (the analysis of the prior section depended on this simple algorithm). Since Soar productions can have a large number of CEs (e.g., Cypress-Soar (Steier, 1987) has over 100 condition elements in some chunks), guaranteeing optimality could be very expensive. Hence the present ordering algorithm cuts down the number of orderings it has to search through, by using heuristics that sacrifice guaranteed optimality.

We therefore created another ordering algorithm that inputs estimates of the number of WMEs matching the CEs and some data about the generation of tokens. It then performs a complete search and outputs an optimal ordering (Tambe and Newell, 1988). The new algorithm performs branch and bound search and uses heuristics from (Smith and Genesereth, 1986), which are guaranteed to preserve optimality. (Despite these heuristics, it may take up to an hour to order a production.[12] Therefore the new algorithm cannot replace the old one in running Soar tasks.)

Figure 9 shows the result of the new ordering algorithm across the eight tasks. The figure shows a graph identical to Figure 5, except for the arrows. In Figure 9, the x-axis depicts the average number of conditions in the chunks formed in various tasks and the y-axis depicts the corresponding computational effect. The arrows indicate the change in the computational effect with the new ordering algorithm. Thus, in the Eight-puzzle task, the computational effect has increased from 0.15 to 0.35 with the new ordering algorithm. However, there are no arrows associated with the cheap-chunks tasks. This is because the cheap-chunks tasks show no difference in the computational effect with the new ordering algorithm. The branching of the k-search tree in the cheap chunks is very low. Thus, ordering cannot make a measurable difference in these tasks. In the expensive-chunks tasks, about 50% to 75% of the cost is attributable to a bad ordering by the old Soar ordering scheme—the computational effect in these tasks has increased by a factor of two to three.[13] This is because multi-objects and a large number of CEs exact a heavy price if the ordering heuristics do not work. The figure shows that the computational effect in the expensive-chunks tasks is still quite pronounced compared to the cheap-chunks tasks—although it is now closer to the computational effect of the Waterjug and Farmer tasks.

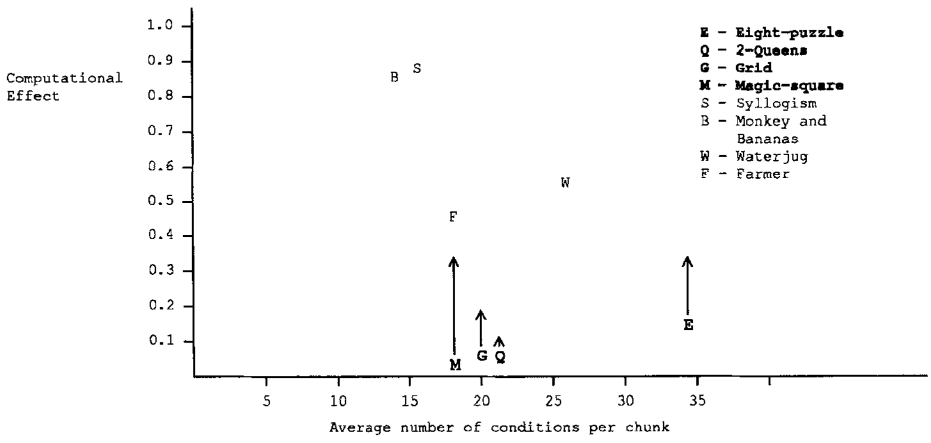


Figure 9. Average number of conditions in the chunks formed in various tasks (on the x-axis) and their corresponding computational effects (on the y-axis). The arrows indicate the effect of using the new ordering algorithm on the computational effect. The expensive-chunks tasks are highlighted.

At this point, we summarize the main points of this section: expensive chunks are caused by big footprints, multi-objects, and bad condition orderings. A big footprint causes a tall k-search tree to be generated in matching an expensive chunk, while multi-objects and bad condition ordering cause branching in the k-search tree. Out of these three factors, big footprints explain a relatively minor portion of the expensive chunks phenomenon.

5. Eliminating expensive chunks

Before discussing the strategies to eliminate expensive chunks, we need to consider the following question: What does it mean to solve the expensive chunks problem? An ideal solution to this problem would impose a fixed upper bound on the cost of each chunk, allowing the system to bound its time per action (there is still an issue if the number of chunks is unbounded, but we will deal with that issue later). Currently, this cost can be unbounded, since the cost of a chunk can be exponential in the number of conditions, and there is no bound on the number of conditions in a chunk. Even if a fixed bound cannot be imposed, it would be prefereable to let the cost of the chunks be a small polynomial function rather than the existing exponential function. Such chunks will require only a few additional tokens to match them (when compared to the original system) and hence will only minimally distort the constant time per action model.

5.1. Strategies for eliminating expensive chunks

Among the various strategies for eliminating expensive chunks, two that initially look like potential solutions can immediately be ruled out by the inherently exponential nature of the production match. One technique is the use of smarter match algorithms. This could involve either better automated condition orderings, or other techniques such as selective backtracking (Pereira and Porto, 1982). A better conditon ordering can definitely reduce the amount of k-search, as shown in the previous section, but cannot completely eliminate it, or even make it non-exponential in general. Unfortunately, other smart match algorithms suffer from a similar problem—they cannot make the k-search non-exponential in general. In fact, smart match algorithms may adversely affect optimizations like state-saving and thus introduce additional overheads. The other technique that looks like a potential solution is the use of massive parallelism. However, given any amount of parallelism it is always possible to have an exponential match that will exceed the capacity of the machine.

After eliminating the above approaches, the remaining known strategies addressing the problem of expensive chunks (or more generally, expensive learned rules) can be divided into two major categories:

1. *Selective solutions:* These approaches depend on selectively avoiding the accessing of expensive learned rules from the knowledge base. They can be further divided into three subcategories:
   a. *Selective learning and forgetting:* The learning system goes through the process of creating the rule to be learned after solving a problem. It then determines if the rule is expensive or cheap. If the rule is expensive, the system does not add it to its knowledge base. The system may have an additional capability of selective forgetting. With selective forgetting, the system adds the rule to its knowledge base, but if it finds out that the rule becomes expensive while in use, it throws the rule away.

      Typically selective learning/forgetting systems adopt the following criterion for determining the expense of a rule: a rule is expensive if its estimated processing cost exceeds its estimated benefits (Keller, 1987; Markovitch and Scott, 1988; Minton, 1985; Minton, 1988b).
   b. *Selective Matching:* The learning system creates the rule to be learned and adds it to the knowledge base. The matcher reasons about the expense of the learned rules and other rules that may be applicable and decides the best rule to apply at that point in time. Thus, the compiled rule always remains in the knowledge base (Markovitch and Scott, 1989a); it is not thrown away.
   c. *User Intervention:* In this approach, the user of the system monitors its performance. If he/she notices a performance problem during learning, then he/she intervenes and removes the efficiency bugs in the system.
2. *Transforming the pattern to be matched:* These approaches depend on uniformly reducing the cost of the learned rule by transforming it, so that expensive rules are not added to the knowledge base in the first place. They can be further divided into two subcategories:
   a. *Modifying learned rules:* This approach requires the system to modify learned rules. After learning a rule, its left-hand side is simplified to reduce its match cost. This reduction may be accomplished by processes such as compression (Minton, 1988b) and removing applicability conditions (Chase, et al., 1989).
   b. *Restricting expressiveness:* This approach requires the system to give up some expressiveness of its language to guarantee that all learned rules are cheap. Thus, no evaluation need be done to guarantee that the system will not degrade in performance.

Furthermore, the system is not required to modify learned rules. This tradeoff is similar to the tradeoff between the expressiveness of a representational language and its computational tractability (Levesque and Brachman, 1985).

Two important criteria for adopting a particular approach from the above list are: (1) the effectiveness of the approach in addressing expensive chunks and (2) how well the approach integrates with the rest of the system, given the assumptions underlying the system. The approach adopted in this article is one of restricting expressiveness. Meeting the first criterion above, i.e., demonstrating the effectiveness of this approach, is the focus of the rest of this article. Meeting the second criterion, i.e., establishing that this approach best conforms with the assumptions in Soar, compared to the other approaches presented, is the focus of the following four paragraphs.

Among the approaches presented, the user intervention approach is ruled out immediately, since the goal of the Soar project is to build a system capable of autonomous existence, i.e., a system that is not dependent on the user. In fact, the requirement for autonomy is one key way in which the problem of expensive chunks differs from the more general problem of expensive productions.

The selective learning/matching approaches depend on analyzing the cost and/or benefits of individual productions. This can be achieved by two different methods:

1. Using a fixed piece of code as part of the architecture, i.e., using a mechanism in the implementation domain.
2. By problem-solving, i.e., using a mechanism in the cognitive domain.

Including a mechanism for cost/benefit analysis in the architecture is against one of Soar's major design principles—no fixed *trap-state* mechanisms (Newell, 1990). A trap-state mechanism is one whose commands the system has to accept, but which has insufficient knowledge to cover all potential situations. The conflict-resolution phase in OPS5 is an example of a trap-state mechanism. When the knowledge encoded in the trap-state mechanism fails, as it necessarily has to if the enountered situations are diverse enough, the system can be led into problems from which it cannot recover—in this case matching a very expensive chunk. Resource bounds may help the system recover from such problems, e.g., they may impose some time-outs on the match. But these resource bounds are trap-state mechanisms in themselves. Consider the example of imposing time-outs on the match. Suppose there is a production that controls some important external device. The time-out may prevent the production from controlling the external device in a critical real-time situation, leading to a disaster. Thus, a time-out mechanism is a trap-state mechanism that could interfere in getting Soar to do real-time tasks sometime in the future.

The cognitive domain approach for cost-benefit analysis does not suffer from the trap-state problem. However, this approach cannot be used, since it would require keeping track of the costs and benefits of the productions in the cognitive domain, which violates the non-penetrable memory assumption mentioned in Section 3.1. A similar argument eliminates the possibility of modifying learned rules—it would be a trap-state mechanism in the implementation domain and would violate the non-penetrable memory assumption in the cognitive domain.

The approach based on restricting expressiveness is not dependent on any mechanism, in either the cognitive or the implementation domain. Thus, it entirely avoids the issues of non-penetrability or trap-states faced by the other approaches and is in fact, the only approach that conforms with the assumptions underlying Soar. But how should schemes for restricting expressiveness be selected or even devised? An important constraint that will be used for selection is the guarantee of *closure under chunking*, i.e., if the productions and working memory meet the restrictions imposed by a scheme before chunking, then the chunks should also meet the restrictions. If chunking violates the restrictions and creates expensive chunks, then that would clearly defeat the purpose of this exercise.

In devising schemes for restricting expressiveness, two obvious candidates are: restricting the size of the footprint, and restricting the use of multi-objects. A restriction on the size of the footprint will bound the depth of the k-search tree. This will make the cost of the chunk a polynomial in the branching factor fo the k-search tree. However, unless this bound is small, it will not be useful. But a small bound on the number of CEs will require extensive modifications to the current methods of task representations in Soar. It would also be difficult to guarantee closure under chunking since chunks usually have a higher number of CEs than hand-written productions (Tambe, et al., 1988). Providing such guarantees may require modifications to chunking, and perhaps also to the rest of Soar. Thus, it would be preferable no to impose an arbitrary bound on the size of the footprint.

The other alternative of restricting multi-objects seems more promising. One category of restrictions in this mode would be to impose a partial restriction on the use of multi-objects—for instance, to limit the number of conditions per production that can match multi-objects. However, it is difficult to guarantee closure under chunking for such partial restrictions. In the previous example, the chunks formed can easily exceed the limit on conditions per production that match multi-objects. A better approach is to eliminate multi-objects altogether, which clearly guarantees closure under chunking. Multi-objects control the branching of the k-search tree. Therefore, eliminating multi-objects eliminates the exponential from the cost (B^D) of the k-search tree—it reduces the branching factor of the k-search tree ot one. This limits the number of tokens in the k-search tree to the size of the footprint. Thus,

The cost of matching a chunk will be linear in the number of conditions.

As stated earlier in this section, this is a very agreeable bound on the cost of the chunks. The size of the footprint grows at most linearly with the amount of time spent in the subgoal. Thus, eliminating multi-objects imposes a very reasonable limit on the cost of a chunk—these chunks are cheap chunks. However, it is still possible for a chunk with a very large footprint to distort the constant time per action model. Dealing with such footprints will be the subject of future work.

5.2. Eliminating multi-objects

Recall that multi-objects refer to a combination of multi-attributes and preferences. The elimination of multi-attributes implies that the system is not allowed to assign two or more values to any attribute. The new restricted representation without the multi-attributes is

```
(state S1   ^block B1)   
(block B1   ^next B2)      A linked list of blocks
(block B2   ^next B3)      is introduced
(block B2   ^on-top-of B1)
(block B3   ^on-top-of B1)
```

Figure 10. Unique-attribute encoding of the blocks world from Figure 7.

referred to as the *unique-attribute* representation. Figure 10 shows how the blocks in the blocks world, represented originally as multi-attributes in Figure 7(a), are represented as unique-attributes. Note that the blocks are no longer a multi-attribute of the state. Instead, a linked list is introduced, with the state pointing the head of the list. The issues in adopting such unique-attribute representations will be discussed in detail in Section 8.1.

Preferences are dealt with somewhat differently. Recall that preferences cause branching in the k-search tree due to their unbound identifiers. Therefore, a restriction is imposed to require that all preference identifiers be prebound. However, chunking may violate this restriction, and create chunks which include preferences with unbound identifiers. Soar's chunking algorithm has therefore been modified (on an experimental basis) so that preferences with unbound identifiers are not included in the chunks. This modification prevents the preferences from causing branching in the k-search tree.

The restrictions for the cheap chunks are summarized below.[14]

1. Multi-attributes are not allowed; only unique-attributes may be used.
2. Preference identifiers must be prebound.

One of the flexibilities not restricted is the ability to match multiple goals in the hierarchy. The number of goals in the goal-hierarchy can grow only linearly with time, as measured by the number of decision cycles. In fact, typically there are only about three to four goals in the goal-hierarchy. This causes a maximum k-search breadth of three to four tokens in the first one or two CEs of a chunk. This low cost appears to be worth the flexibility of matching multiple goals in the goal hierarchy, which allows a chunk to transfer to any of the goal-contexts in the context hierarchy. However, if matching multiple goals turns out to be a problem, then this capability can also be restricted.

Except for the experimental modification to chunking, the other restrictions can be adopted in the current version of Soar using a set of conventions. We have used this version for the empirical analysis presented in this paper. In the actual implementation, assigning multiple values to a single attribute should lead to the creation of an impasse, allowing the architecture to enforce a representation without multi-attributes. The new version of Soar (Soar 5) will allow a very straightforward incorporation of this mechanism (interestingly, unique-attributes did not influence the design of Soar 5).

Table 2 presents data from nine different Soar tasks.[15] Except for the Tree task to be introduced in Section 6.2, these tasks are from Table 1. The main message of this table is that the unique-attributes eliminate expensive chunks. The first column gives the total run time before chunking, using the multi-attribute representation. The second column gives

Table 2. The total run time after chunking with two different representations.

Task	Total run time multi-attr. before chunking (sec.)	Total run time multi-attr. after chunking (sec.)	Total number of chunks multi-attr.	Total run time unique-attr. before chunking (sec.)	Total run time unique-attr. after chunking (sec.)	Total number of chunks unique-attr.
Eight-puzzle	28.69	34.56	11	26.67	8.88	86
2-Queens	4.48	13.52	3	3.55	0.40	3
Grid	23.44	12.65	17	19.81	3.09	142
Magic-square	13.92	18.72	5	12.62	2.72	5
Tree	10.28	1.55	11	9.94	1.39	72
Syllogisms	8.46	0.82	11	7.92	0.82	11
Monkey	7.13	1.73	5	8.21	1.83	5
Waterjug	21.43	3.56	11	19.84	2.25	11
Farmer	26.94	4.39	14	20.25	2.07	14

the total run time after chunking, using the multi-attribute representation. The third column gives the number of chunks formed in the multi-attribute representation. The last three columns give comparable data for the unique-attribute representation.

The table shows that the time to complete the task without chunking in both representations is comparable. However, in the four expensive-chunks tasks, the multi-attribute representation goes on to form expensive chunks. Chunks formed in the unique-attribute representation are cheap and the total run time after chunking is much lower than the total run time before chunking. The two cheap-chunks tasks (Waterjug and Farmer) that had a significant computational effect also show a speedup with the unique-attributes.

Note that, in some tasks, a significantly larger number of unique-attribute chunks are learned. This effect is due to a secondary impact of unique-attributes—the chunks that are learned may be less general. This reduction in generality occurs because a multi-attribute condition can match any element of a set, while a unique-attribute conditon can match only one possible element. When the unique-attribute chunks were less general than the multi-attribute ones, additional trials were run on the same task until enough chunks had been learned to cover the same scope as the multi-attribute chunks. For this experiment, these trials were selected by hand so as to cover chunks not yet learned. (This issue is picked up in more detail in the following sections.) However, the table shows that even after learning the larger number of chunks, the unique-attribute run times are much lower than the multi-attribute run times.

6. Complexity analyses of two illustrative tasks

The previous section presented the unique-attributes restriction on the expressiveness of Soar's production system language. Unique-attributes allow us to guarantee that only cheap chunks will be produced by the system. This section presents complexity analyses of two simple tasks to demonstrate the expressiveness-efficiency tradeoff involved in unique-

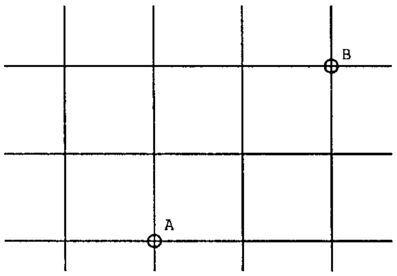


Figure 11. The Grid task.

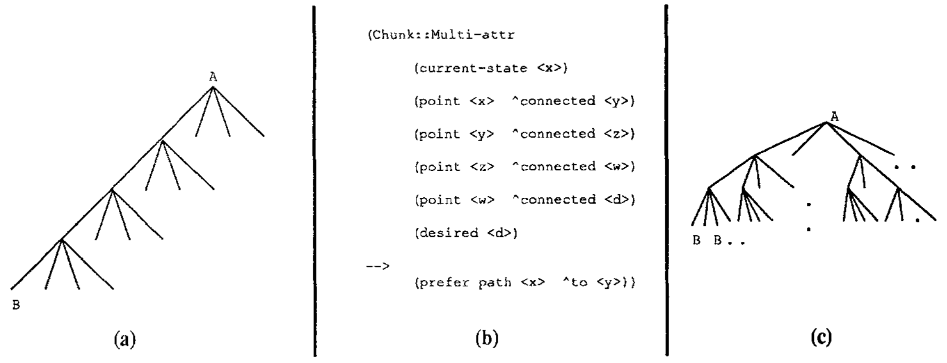
attributes. The first subsection use the Grid task to illustrate this tradeoff. The second subsection uses the Tree task to present a best- and worst-case analysis of the expressiveness-efficiency tradeoff.

6.1. The Grid task: An illustration of the expressiveness-efficiency tradeoff

Consider an example from the Grid task, shown in Figure 11. To simplify some of the following analysis, assume that this grid is infinite. The problem is to go from point A to point B, a path of length four. This problem is solved first using multi-attributes and then using unique-attributes.

In the multi-attribute version, the grid is represented using *connected* as a multi-attribute of a *point* on the grid. Any point Y adjacent to a point X on the grid is represented as: (point X ^connected Y). Thus, the multi-attribute *connected* represents an unstructured set of connections between a point and all of its immediate neighbors. The problem space has only one operator: *move*. The state contains a pointer to the current position on the grid. If the current position is at point x, then for each point y connected to point x, the operator *move* will be instantiated. The problem-solver will solve the problem using some heuristics, or outside guidance, generating a k-search tree of tokens as shown in Figure 12(a). This process generates 16 tokens, with four tokens per each step generated from the four options available to the problem-solver at each point on the grid. Even if the heuristics do not directly lead the problem-solver to the solution, the matcher will generate only 4 tokens per step. Note that this is a highly simplified version of the real Soar production system. For instance, we have not accounted for the cost of subgoaling, or for the cost of the heuristics. However, since the purpose here is to analyze the cost of multi- and unique-attribute chunks, we will use this simple model.

The chunk formed in solving the task is shown in Figure 12(b). The chunk says that if the goal is to reach a point <d>, and if the current position is point <x>, and if there is a path of length four between them, then prefer the instantiated *move* operator along that path. (The prefer action indicates a preference for a path from <x> to <y> over all other paths; recall from Section 3.2 that < > indicates variables.) This chunk does not consider the points along which the path goes or the direction the path takes. The chunk will therefore transfer to all pairs of points with a path length of four between them.



```
(Chunk::Multi-attr
    (current-state <x>)
    (point <x>  ^connected <y>)
    (point <y>  ^connected <z>)
    (point <z>  ^connected <w>)
    (point <w>  ^connected <d>)
    (desired <d>)
-->
    (prefer path <x>  ^to <y>))
```

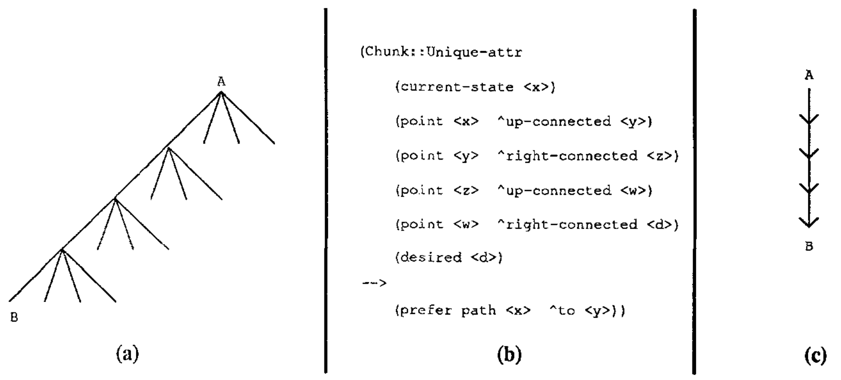
Figure 12. The Grid task with multi-attributes: (a) the k-search tree for the entire task consisting of four problem solving steps; (b) the chunk formed (prefer action indicates a preference for a path from < x > to < y > over all other paths); and (c) the k-search tree formed in matching the chunk (generated for one problem-solving step).

Figure 12(c) shows the k-search tree formed in matching the chunk. For the sake of simplicity in the analysis, only the conditions of class-name *point* in the chunk are considered. (Considering other conditions does not change the results.) Here, each condition has multiplied the number of tokens by four, which is the number of points connected to any given point. Since there are four conditons in the chunk (for a path of length four), the total number of tokens (4 + 16 + 64 + 256 =) 340 tokens. These 340 tokens are generated in one step. Comparing this with the four tokens per step in the original problem-solving, we see that the chunk is *expensive*. The 340 tokens correspond to all possible paths originating from point A that have a length of four.

In the unique-attribute version, the state points to the current location on the grid, similar to the multi-attribute version. However, each location points to its four adjacent locations using specific unique-attributes; *up*, *down*, *left*, and *right*. Instead of one *move* operator, there are four different operators, *move-up*, *move-left*, *move-right*, and *move-down*. Again, the problem-solver moves from A to B using heuristics or outside guidance, generating the tree of tokens shown in Figure 13(a).

However, the chunk formed in this process is different. The chunk is shown in Figure 13(b). It says that if the goal is to move to point < d > from point < x > and if the connection between the two points is through the specific relation (*up-right-up-right*) described, then choose the appropriate operator: *move-up*. The k-search tree formed is shown in Figure 13(c). There are only four tokens per step formed in this case. The chunk formed is much cheaper than the chunk in the multi-attribute case. However, the chunk will transfer only if the two points are connected in a specific manner—*up-right-up-right* in this case, as opposed to any arbitrary connection of length four in the earlier case.[16]

Table 3 summarizes the cost and generality of the two representations. The generality is measured in terms of the number of transfers in an *nxn* grid, i.e., the number of source destination pairs that the chunk can transfer (or apply) to. The length of the path traversed in the grid is assumed to be *p*. However, boundary effects are ignored for simplicity. (See Appendix II for the derivations.)



```
(Chunk::Unique-attr
   (current-state <x>)
   (point <x>  ^up-connected <y>)
   (point <y>  ^right-connected <z>)
   (point <z>  ^up-connected <w>)
   (point <w>  ^right-connected <d>)
   (desired <d>)
-->
   (prefer path <x>  ^to <y>))
```

Figure 13. The Grid task with unique-attributes: (a) the k-search tree for the entire task consisting of four problem-solving steps; (b) the chunk formed (prefer action indicates a preference for a path from < x > to < y > over all other paths); and (c) the k-search tree formed in matching the chunk (generated for one problem-solving step).

Table 3. The cost and generality of the multi- and unique-attribute representations for the Grid task. Here n is the number of nodes in the grid and p is the path length.

Representation used	Cost in tokens per chunk	Generality in number of transfers per chunk	Number of chunks required for same level of generality as multi-attr.	Cost in tokens after achieving the same level of generality
Multi-attributes	$(4^{p+1} - 4)/3$	$n^2*(p+1)^2$	1	$(4^{p+1} - 4)/3$
Unique-attributes	p	n^2	$(p+1)^2$	$(p+1)^2*p$

Comparing the multi-attributes and unique-attributes, we see that the multi-attributes allow generality that is $(p + 1)^2$ times more than the generality achieved by a single unique-attribute chunk. Thus to achieve the same generality, the unique-attribute system has to learn $(p + 1)^2$ chunks. However, even after learning all those chunks, the cost of matching all of the productions is only a polynomial number of tokens in the unique-attribute system. It is exponential (in p) in the multi-attribute system.

Thus, the unique-attribute system has paid polynomial space to eliminate exponential time—which may, at first, appear counterintuitive. This apparent anomaly can be explained as follows. Figure 14 shows the number of positions tht can be reached in the Grid task by the chunk shown in Figure 12(b) (a path length of four). The node marked x indicates the source location. The nodes marked with small circles indicate the positions that can be reached. The chunk shown in Figure 12(b) is an expensive chunk, based on multi-attributes. This single expensive chunk can generalize to all the situations shown, with the source fixed at the location marked x. There are only a polynomial number of positions that can be reached from x: $(p + 1)^2 = 25$; however, there are an exponential number of paths of length four, to these positions: $4^p = 256$. When given the goal of reaching one particular position, the chunk from Figure 12(b) finds all possible paths of length four, discovering all 256 paths to all 25 positions—an excessive amount of k-search, since only

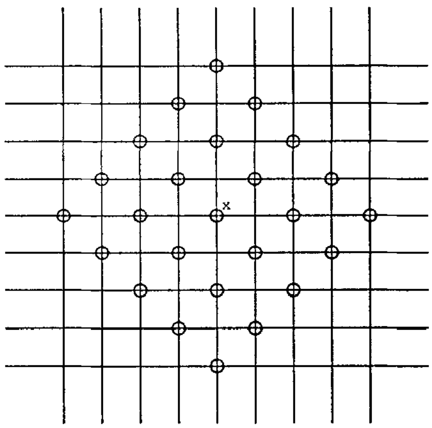


Figure 14. Locations reached by a path of length four in the Grid task.

a single path to any particular position is required. This k-search of all paths to each of the positions gives rise to the exponential factor.

In contrast, a chunk learned by the unique-attribute representation performs the minimal k-search of finding a single path to a single position, as shown in Figure 13(c). Even after learning all $(p + 1)^2$ chunks in this representation, the total amount of k-search done is proportional to the number of destinations (25), and not to the total number of paths to each of the destinations (256). The unique-attribute system avoids the useless computation of finding all paths to each position.[17] Thus, it is able to trade off polynomial space for exponential time.

This example also illustrates the relation between k-search and generality. Consider a single unique-attribute chunk. It can perform a limited amount of k-search. In the Grid task, it can find a single path to a single position. The multi-attribute chunk performs more k-search. In the Grid task, it uses k-search to reach all the locations marked with small circles in Figure 14. Thus, the difference in the two representations is the amount of k-search that can be performed to find situations in which the conditions of a chunk match. The inability of unique-attributes to perform k-search manifests itself as an effective loss of generality. To make up for this, the unique-attribute system learns more chunks, where each chunk performs a limited amount of k-search.

Thus, the multi-attribute representation uses k-search to gain generality. Comparing the k-search performed by the multi- and unique-attributes in the Grid example, we see that the amount of k-search done (and the generality obtained thereby) by the multi-attributes is composed of two portions: (1) an essential portion, (2) an excessive portion. The essential portion of the k-search consists of a single k-search path to each of the destinations on the grid. The excessive portion of the k-search, a typical characteristic of the multi-attribute representation (Tambe and Newell, 1988), consists of everything except the essential portion. This excessive portion does not provide any useful generality. The unique-attribute system only performs the essential portion of the k-search; it entirely avoids the excessive portions.

In summary, more expressiveness provides greater generality in chunking. However, the price paid for the higher level of generality is an increase in the amount of k-search at performance time. The k-search for a chunk in the two representations can be characterized as follows:

1. *Multi-attributes:* The number of tokens in the k-search tree can be exponential in the number of CEs. However, the depth of the k-search is bounded by the number of CEs.
2. *Unique-attributes:* The number of tokens in the k-search tree is bounded by the number of CEs in the chunk.

The restriction on multi-attributes does not imply that a Soar system has lost all of its sources of generality. Other sources of generality, which are independent of the amount of k-search done, can still be exploited. For example:

1. *Implicit generalization:* Chunks are based only on those aspects of the situation that were referenced during problem-solving in the subgoal to produce results (Laird, Rosenbloom, and Newell, 1986). For example, in subsection 3.1, the CEs of the chunk are based on only a fraction of the existing working memory.
2. *Focus:* The notion of focus is based on the notion of relative representation, i.e., representing objects or positions relative to a particular object or a position called focus. For example, positions on the grid are *up-down-left-right* relative to the current focus, i.e., the current position. Chunks learned reflect the representation relative to the current focus, allowing transfer if the focus shifts.
3. *Decomposition:* If a task is decomposed into smaller subtasks, then chunking the smaller subtasks independently provides another source of generality. For instance the Seibel task requires reacting to the on-off conditions of ten lights (Rosenbloon and Newell, 1986). Instead of considering all the lights at once, if small subgroups (of 2–4 lights) are considered independently, then chunking on those provides an alternative source of generality. The smaller subgroups will transfer to subparts of some other combination of lights.

The unique-attribute representation for the Grid task exploits all three sources of generality listed above:

1. *Implicit generalization:* Chunks are based only on the path traversed. The rest of the grid does not appear in the chunks. The chunks can therefore transfer irrespective of the grid formation, as long as the given path exists.
2. *Focus:* The chunk uses a path relative to the current focus, i.e., the current position on the grid. This allows a *translational* transfer, if the focus is moved to a different source.
3. *Decomposition:* The process of solving a particular problem for a path length of four generates subgoals for all intermediate path lengths, and hence generates chunks for all intermediate path lengths.

6.2. The best and the worst case for the expressiveness-efficiency tradeoff

The previous subsection illustrated the expressiveness-efficiency tradeoff involved in the Grid task. This subsection presents a best and worst case analysis for the tradeoff. In the best case of the expressiveness-efficiency tradeoff, a single unique-attribute chunk is just

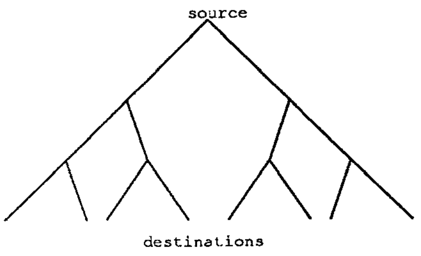


Figure 15. The Tree task.

as general as the corresponding multi-attribute chunk, but a lot cheaper (the unique-attribute chunk saves time). In the worst case for the expressiveness-efficiency tradeoff, a single unique-attribute chunk is quite specific and a large number of unique-attribute chunks are required to reach the same level of generality as a multi-attribute chunk. However, the large number of unique-attribute chunks are just as expensive as the multi-attribute chunk. A good example for demonstration of these cases is the Tree task shown in Figure 15. The Tree task is just like the Grid task except that the structure to be searched is a binary tree of uniform depth, and the path to be found is always from the root to one of the leaves.

The multi-attribute and unique-attribute representations in this task are similar to those in the Grid task, except for the use of *left-connected* and *right-connected* rather than *up*, *down*, *left*, and *right* for the unique-attributes. The most general chunk (with multi-attributes) is shown in Figure 16(a). It covers all the destinations (the leaves), given the root of the tree as the source. One particular unique-attribute chunk is shown in Figure 16(b). The unique-attribute chunk will cover *only one* particular leaf of the tree. This observation reveals an important characteristic of the Tree task—the only source of generality available is k-search. The unique-attribute system does not benefit from any other sources of generality. It does not benefit from using a focus, implicit generalization, or decomposition, since (1) the source for this task is always the root, (2) the destination is always one of the leaves, (3) the path length is fixed, and (4) the path to each destination is unique.

```
(Chunk::Multi-attr
        (current-state <x>)
        (point <x>  ^connected <y>)
        (point <y>  ^connected <z>)
        (point <z>  ^connected <d>)
        (desired <d>)
 -->
        (prefer path <x>  ^to <y>))
```

(a)

```
(Chunk::Unique-attr
        (current-state <x>)
        (point <x>  ^left-connected <y>)
        (point <y>  ^right-connected <z>)
        (point <z>  ^left-connected <d>)
        (desired <d>)
 -->
        (prefer path <x>  ^to <y>))
```

(b)

Figure 16. Chunk formed in the Tree task with the two representations.

Table 4. The cost and generality of the multi-attribute and unique-attribute representations for the Tree task, assuming a branching factor of B and a depth of D. In deriving the cost for unique-attributes worst case, sharing (Section 3.2) is assumed for conditions in the chunks. The best and worst case refers to the best and worst case of the expressiveness-efficiency tradeoff.

Representation used	Cost in tokens per chunk	Generality in number of Transfers	Number of chunks required for same level of generality as Multi-attr.	Cost in tokens after learning all chunks
Multi-attributes	$B^*(B^d-1)/(B-1)$	B^d	1	$B^*(B^d-1)/(B-1)$
Unique-attributes best case	d	1	1	d
Unique-attributes worst case	d	1	B^d	$B^*(B^d-1)/(B-1)$

Since k-search is the only source of generality in this task, it exhibits the expressiveness-efficiency tradeoff in its clearest form. Table 4 presents data on the cost and the generality of the chunks for the multi- and unique-attributes (for both the best and worst cases) assuming a Tree task with a branching factor of B and a depth of D. In the best case for the unique-attributes, the task is to reach only one of the leaves, i.e., only a single destination has to be chunked. This task is accomplished by learning a single chunk, with a cost proportional to the depth of the tree. In this case, almost all of the exponential k-search of the multi-attributes—except for the one required k-search path—is excessive. Thus, the unique-attribute system exhibits a big efficiency gain, without any losses in terms of generality.

In the worst case for the unique-attributes, all of the destinations (or leaves) of the tree have to be chunked. In this case, the cost of matching all unique-attribute chunks (one chunk per path) is equal to the cost of matching one multi-attribute chunk. There is no excessive k-search involved in matching the multi-attribute chunk. Since there is no excessive k-search, this task demonstrates the worst-case for the expressiveness-efficiency tradeoff. Furthermore, the lower generality of the unique-attribute chunks demands an exponential number of chunks (exponential in the depth D) to cover the level of generality of one multi-attribute chunk.

An obvious question that the worst case analysis raises is: If the unique-attribute version is going to have to acquire an exponential match anyway (to match the exponential number of productions), why not acquire it all at once via the multi-attribute chunk? The answer to this question lies in the issue of the safety of chunking, i.e., the issue that chunking should not hurt Soar's performance. The multi-attribute chunk can add an arbitrarily large exponential cost in a single learning trial. In contrast, the unique-attribute version learns about the individual branches as they are encountered. The match cost always increases gradually (at worst), and remains bounded by the number of branches that have been encountered. At worst the number of branches that have been encountered is equal to the number in the tree, but in many domains only a small portion of the entire exponential space is ever encountered. A related point is that the system is also protected from learning an exponential number of chunks by its finite lifetime. If the chunking rate is approximately constant over time (see Section 8.2), then there is a finite number of chunks that the system will ever be able to acquire. Under these circumstances the system can work in arbitrarily large exponential domains, but it will never have enough time to learn everything about the domain (as opposed to learning everything about the domain quickly, but never having enough time to use it). Of course, not having enough time to learn everything about a domain also implies that Soar may not actually benefit from chunking.

In the interest of clarity of exposition, this section analyzed a simple tree-structure (with uniform branching factor B) to demonstrate the best and worst cases of the expressiveness-efficiency tradeoff. The actual complexity expressions for the best and the worst cases can vary according to the individual problem being solved e.g., instead of the simple tree-structure, it is possible to consider a complex tree-structure where the branching factor grows exponentially with depth.

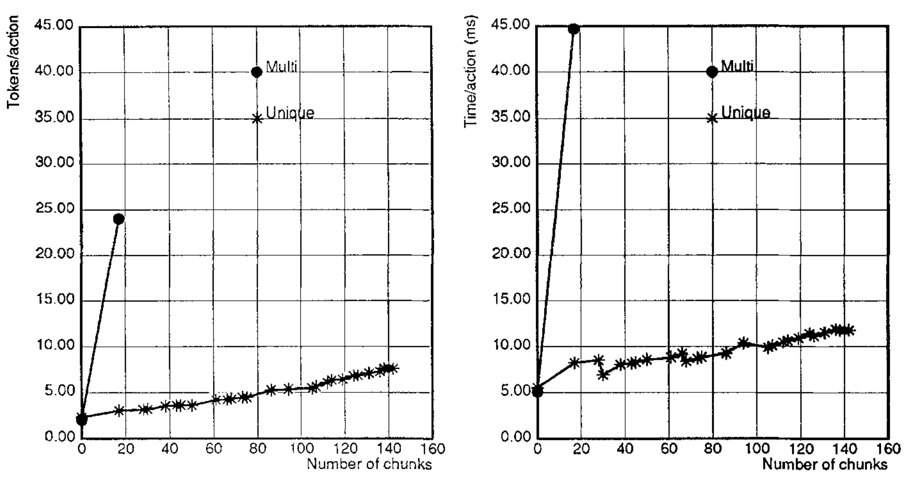
It is useful to analyze how unfavorable the expressiveness-efficiency tradeoff can get in performing the Tree task. Note that for performing this task, the tree-structure must be present in working memory. If this tradeoff is to be heavily set against the unique-attribute system, the tree-structure present in working memory must be very large. However, the tree-structure is exponential in the depth of the tree. An exponential amount of time must be spent in generating such an exponential structure. (In this respect, the Tree task is to be contrasted with the Grid task, where the size of the structure is limited, but matching the structure requires exponential time due to connectivity.) The unique-attribute system is thus protected by its finite lifetime—it is unlikely that the problem-solver will be able to generate such large tree-structures in its lifetime. If the tree-structure is not large, the expressiveness-efficiency tradeoff is not very unfavorable to begin with; the unique-attribute system will be able to obtain the required coverage fairly quickly. Thus, this tradeoff is expected to not be set heavily against unique-attributes. Interestingly, none of the tasks from Table 2 have exhibited the worst case of the expressiveness-efficiency tradeoff in the unique-attribute representation.

In conclusion, in the worst case (the tree search) of the expressiveness-efficiency tradeoff, there is no excessive k-search involved. However, from the example of the best case in this section, the Grid task in the previous subsection, and the analysis of the expensive-chunks tasks (Tambe and Newell, 1988), we expect that in the general case, multi-attribute chunks will generate excessive k-search. There is no tradeoff involved in this excessive k-search; the unique-attribute representation simply gets rid of it.

7. Experimental analysis

This section provides a detailed comparative performance analysis of the multi- and unique-attribute based systems. It first compares the computational effects of multi- and unique-attribute representations for the four expensive chunks tasks from Table 1: Grid, Eight-puzzle, 2-Queens, and Magic-square. It then presents a finer grained analysis of the impact of chunking on the overall performance of the multi- and unique-attribute systems.

For each representation and task, the system was run without chunking. The tokens per action and the time per action in performing the task were noted. The system was then allowed to chunk on the problem. It was then run on the same problem, i.e., after having chunked on the problem, and the tokens per action and the time per action in performing the task were noted. A sequence of such experiments was performed with the unique-attribute representation to accumulate a set of chunks yielding the same level of generality as the multi-attribute system. (See Sections 6.1 and 6.2 for examples of how this same level of generality is determined.)



(a) Tokens per action (b) Time per action

Figure 17. Computational effects in the Grid task.

In the Grid task, a 5 × 5 grid was chosen, with p, the length of the path, set at six. Recall that we had assumed an unbounded grid in obtaining the results in Table 3. The results of our experiments will be somewhat different from those results, since these experiments are run with a bounded grid. However, a large grid size would make the chunks in the multi-attribute representation too expensive to allow running the experiments. The multi-attribute and unique-attribute representations used for this task were the ones introduced in the previous section. Figure 17 shows the change in tokens per action and time per action with the addition of chunks for the two representations. The time is measured in milliseconds. The multi-attribute representation learns 17 chunks and causes an increase in tokens per action from about 3 tokens per action to 24 tokens per action[18] and causes a computational effect of about 0.11; i.e., the time per action has gone up by about a factor of 9. The unique-attribute representation requires learning on more problems to reach the same level of generality (each asterisk (*) represents one problem). It accumulates 142 chunks in this process, but its tokens per action and time per action are seen not to increase as much. Even after achieving the same level of generality as the multi-attribute system, the computational effect here is much more limited (about 0.46). These graphs thus support the analysis of the Grid task presented in the previous section—the unique-attribute system is able to avoid large portions of the k-search performed by the multi-attribute system.

Although the computational effect of the unique-attribute system in the Grid task is limited to 0.46, it is not unity, as required for the ideal computational model. This deviation from the ideal computational model occurs because, even though the chunks in the unique-attribute system are individually cheap, each chunk does add something to the match cost. This issue is picked up in more detail in Section 8.2.

The Eight-puzzle task requires arranging eight numbered tiles in a 3×3 frame in a specific order. One of the cells in the 3×3 frame is always blank and adjacent tiles can be moved into the blank cell. In the multi-attribute representation, a state points to nine bindings, each of which connects a cell from the static 3×3 structure of cells to a tile. For example, (*binding B1* ^*cell C1*) (*binding B1* ^*tile T1*) connects cell C1 to tile T1. A cell points to all

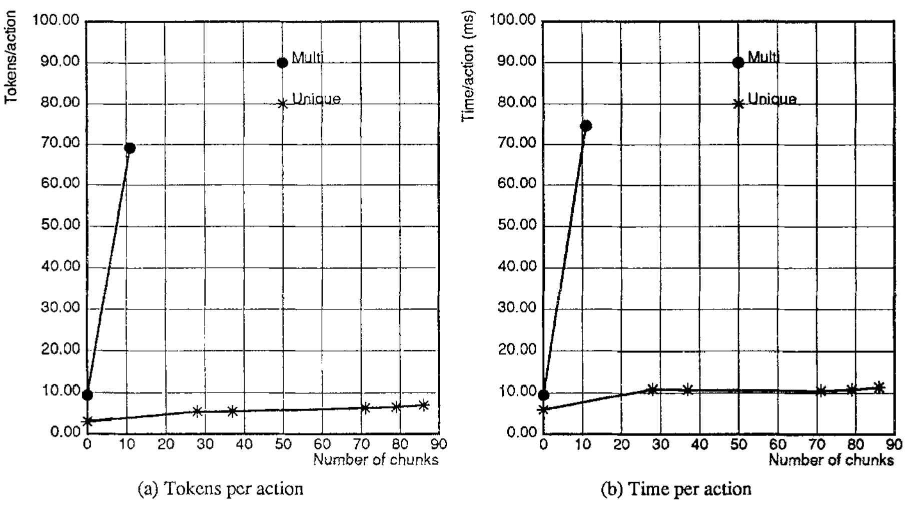


Figure 18. Computational effects in the Eight-puzzle task.

its neighboring cells. For example, (*cell C1 ^next C2*), (*cell C1 ^next C3*), etc. In the unique-attribute representation, bindings are not used. The state points to the blank cell. The blank cell points to its neighbors, which in turn point to their neighbors, using attributes *up, down, right* and *left*. For example, (*cell C1 ^up C2*), (*cell C1 ^down C3*), etc. The cells directly point to tiles. For example, (*cell C1 ^tile T1*). Figure 18 shows data from runs with both representations. The analysis of these graphs is very similar to the analysis of the graphs in Figure 17 for the Grid task. There is only a limited (less than a factor of 2) increase in time per action in the unique-attribute system.

It is important to analyze the sources of generality available in the Eight-puzzle task, in a manner analogous to the Grid task. The chunks learned in the Eight-puzzle are search control rules that give a preference to one operator over the other. For instance, in Figure 19, it is possible to move the blank in two directions: right and up. Depending on the tiles in the current state and their situation in the desired state, the search control chunk formed will prefer one operator over the other. There are three sources of generality available to the multi-attribute version:

- *Implicit Generalization:* The chunk does not take into account the position of the other tiles and cells, except the ones affected by the move operator. It also does not take into account what numbers are on the tiles.
- *Focus*: The chunk is relative to the position of the blank cell. The chunk will transfer to other situations based on the position of the blank cell.
- *K-search*: The chunk will transfer to any other configuration of two operators; it is not sensitive to directions, such as right and up.

Of the three sources listed above, the unique-attribute version cannot use k-search for generalization. The chunks learned will be specific to the direction of the operators. For instance, in Figure 19 the operators under consideration have to move the blank right and up. However the important point is not how much generality is lost. Rather it is how much has remained, in comparison to a no-transfer situation (no-transfer chunks refer to chunks

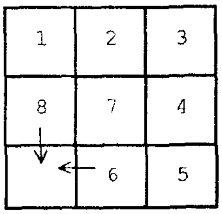


Figure 19. Move operators in the Eight-puzzle task.

Table 5. Number of chunks required in three representations to achieve the same generality for the Eight-puzzle.

Eight-puzzle Representations	Multi-attributes	Unique-attributes	No-transfer
Number of Chunks	5	60	8 million

which have absolutely no source of generality available. Calculations using a somewhat simplified version of the chunks learned in this task show the ratio in Table 5 (see Appendix III for the derivations).

Thus, a substantial amount of transfer is still available to the unique-attribute representation. Although the unique-attribute system forms chunks of lower generality, it obtains a big reduction in the match effort by avoiding excessive k-search.

The results for the 2-Queens and Magic-square tasks are presented in Figure 20. The tokens-per-action graphs in these two cases are similar. Both graphs show the advantage of going with unique-attributes. (For details of the representations employed see (Tambe and Rosenbloom, 1988).) An important point here is that these two tasks are such that

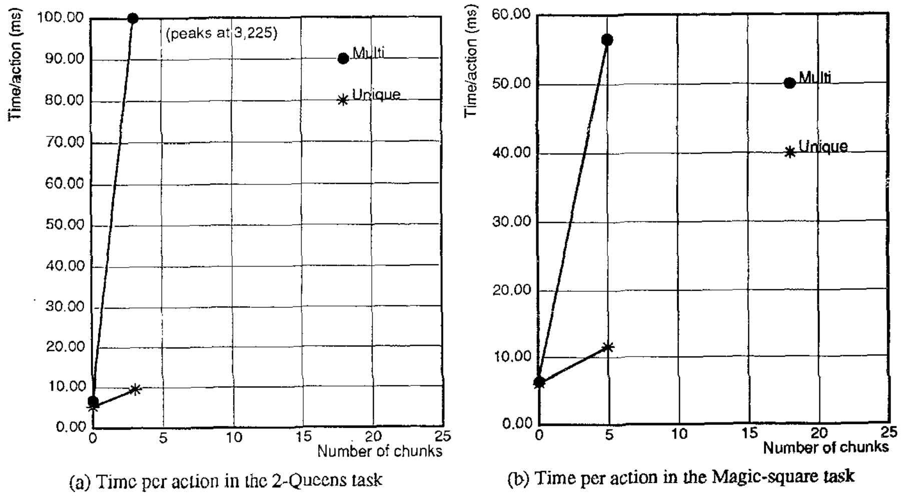


Figure 20. Computational effects in the 2-Queens and Magic-square tasks.

the unique-attribute system *does not have to learn any extra chunks* to achieve the same level of generality as the multi-attribute system.

The results presented in this section so far, along with those in Table 2, show that the unique-attribute version is more efficient after reaching the same level of generality as the multi-attribute version, but they don't show the finer-grained behavior of what happens during the learning process. Specifically, they don't address the issue of the extra time spent by the unique-attributes version in acquiring the extra chunks. To understand this issue, a set of randomly generated problems in the Eight-puzzle domain were run with both versions. Both versions started off with no chunks, and solved the set of problems with chunking turned on, i.e., chunking continuously across the set of problems. Thus, the systems used the chunks learned in one problem to solve the subsequent problems, simultaneously learning more chunks in situations where the earlier chunks did not apply.[19]

Figure 21(a) shows the cumulative times for the two systems on the 20 problems. The unique-attribute system consistently outperforms the multi-attribute system. Figure 21(b) compares the time required by the two versions for the individual problems. This graph shows that even in each individual problem, the performance of the unique-attribute version dominates the performance of the multi-attribute version. Figure 21(c) shows the time per decision for the two systems. Decisions are typically used in Soar to measure the amount of problem-solving effort. The point corresponding to the zeroth problem shows the time per decision prior to learning. The main message of this graph is that time per decision remains fairly constant in the unique-attribute version, while it takes some fairly large jumps in the multi-attribute version. Thus, the number of decisions does not accurately reflect the problem-solving time in the multi-attribute system, but it does so in a unique-attribute system. Finally, Figure 21(d) compares the number of decisions required for solving the problems in the two systems. This graph shows that unique-attributes require more decision cycles than multi-attributes in achieving the goal. The chunks learned by the multi-attribute system are more general, and hence the reduction in the number of decisions per problem happens more quickly than with unique-attributes. However, as more problems are solved, and more unique-attribute chunks are acquired, the difference in number of decisions decreases. Similar results were obtained for the Grid task (Tambe and Rosenbloom, 1988).

8. Discussion

A detailed comparative analysis of the efficiency (and efficiency-related tradeoffs) of the more expressive multi-attributes and the less expressive unique-attributes was presented in Sections 5, 6, and 7. This section focuses on various issues related to unique-attributes: the difficulty of encoding tasks in unique-attributes, the average growth effect, the bounded elaboration phases, and others.

8.1. Difficulty of encoding tasks with unique-attributes

The approach adopted in this article to deal with expensive chunks is to restrict the expressiveness of the production system language. Such restrictions are to be contrasted with the trends in some EBL systems to make the language more expressive (Cohen, Mostow,

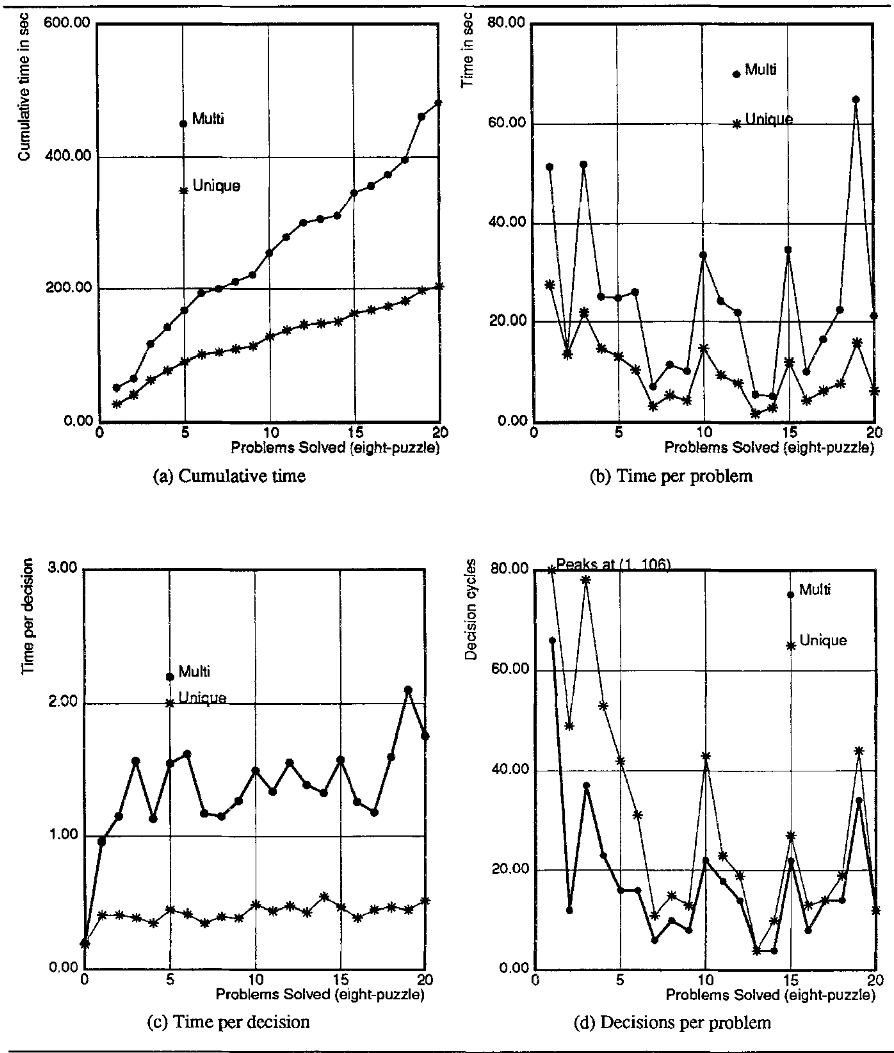


Figure 21. Results for the set of random problems from the Eight-Puzzle task.

and Borgida, 1988; Shavlik and DeJong, 1987). This contrast raises an important question: with the elimination of multi-attributes, would it be considerably more difficult to encode tasks in Soar?

Multi-attributes are used for representing unstructured sets in working memory. In the blocks-world example (see Figure 7(a)), it is possible to keep all the blocks as a structureless collection of items. A single production accesses all three of the blocks via match (see Figure 6). Thus, from a task-encoding perspective, the following features of a multi-attribute based set representation emerge:

1. The elements of the set are accessed using production match, i.e., using k-search in the implementation domain.
2. Since the elements of the set are accessed using production match, they are accessed in parallel, i.e., the matcher returns the set of all matching elements simultaneously.
3. Some set operations, such as search, add, delete, etc., are facilitated.

The elimination of multi-attributes implies that unstructured sets cannot be represented directly in working memory. All sets in working memory have to be structured as lists, trees, or some other task-specific structures. Figure 10 showed how the set of blocks can be structured as a list. (Note that the list structure is chosen in the interests of clarity. This might not be the most suitable encoding for performing this task.) To access individual elements of such structures via production match, multiple productions would have to be written, one for each element in the structure. This requirement poses a problem—a large number of productions would have to be written—especially if the structure contains a large number of elements. This problem can be avoided by accessing the structure in the cognitive domain, i.e., using states and operators.[20] In the example from Figure 10, the execution of an operator can examine one block from the list of blocks (the current focus), and advance the focus to the next block in the list. Thus, from a task-encoding perspective, the following features of a unique-attribute representation for sets emerge:

1. The elements of a set are accessed using problem space search, i.e., using search in the cognitive domain.
2. Since the elements of the set are accessed using problem space search, they are accessed in a serial fashion. (Problem space search is serial because it requires moving from state to state in a serial fashion.)
3. Set operations such as search, add, delete, etc. are performed using operators in the problem space.

As revealed by the features listed above, the principal impact of encoding tasks with unique-attributes is the removal of the combinatorial k-search from the implementation domain. Some of the combinatorics is transferred to the cognitive domain in the form of problem space search, while some, like the excessive portion of the k-search in the Grid task, just disappears. A task encoded with unique-attributes must bear the overheads associated with problem space search (selection of states, operators, etc.). These overheads can cause the system to slow down by a constant factor. The advantage of carrying out the search in the problem space lies in the ability to use search-control knowledge to terminate or control it. This ability can avoid the possibility of exponential slowdowns, which can occur in the implementation domain if multi-attributes are used. Moreover, chunking in a space encoded using unique-attributes will gradually reduce and ultimately eliminate the overheads associated with the problem space search (selection of states, operators, etc.). In the process of eliminating the overheads of problem-space search, chunking will increase the match effort in the implementation domain; however, this increase will always be gradual at worst (the next subsection discusses this issue in more detail). Chunking will also return to the implementation domain the capability of retrieving all elements of a (structured) set in parallel. Consider the structured set shown in Figure 10. If chunks are formed in that domain,

```
(chunk::access-block-B1
  (goal <g>  ^problem-space <p>)
  (goal <g>  ^state <s>)
  (state <s>  ^block <bl>)
          .
          .
          .
          .
```

```
(chunk::access-block-B2
  (goal <g>  ^problem-space <p>)
  (goal <g>  ^state <s>)
  (state <s>  ^block <bl>)
  (block <bl>  ^next <b2>)
          .
          .
          .
```

```
(chunk::access-block-B3
  (goal <g>  ^problem-space <p>)
  (goal <g>  ^state <s>)
  (state <s>  ^block <bl>)
  (block <bl>  ^next <b2>)
  (block <b2>  ^next <b3>)
          .
          .
```

Figure 22. Access with unique-attributes.

they will access the three blocks as shown in Figure 22, without any problem space search to access the three blocks. Furthermore, by matching the three chunks, the matcher can access the blocks B1, B2, and B3 in parallel. But now, the structure of the set represented by the unique-attributes is reflected in the chunks formed; the chunks encode the *next* relation. The structure of the set is used by the matcher to restrict the k-search branching factor to one, e.g., only one block is next to the current block, guaranteeing the chunks to be cheap.

To further facilitate the encoding of common set operations like search, add, delete, etc., a set of operators which can perform many common set operations has been implemented. These operators are expected to effectively replace the functionality provided by multi-attributes.

To gain a better understanding of the difficulties in encoding tasks, some complex task domains need to be converted to unique-attributes. Toward this end, R1-Soar (Rosenbloom, et al., 1985), a large Soar task with about 450 productions, which forms part of an expert system task for computer configuration, was converted into the unique-attribute representation. This conversion was expected to take about a week or two. However, the conversion took only about two person days. This was mainly because R1-Soar uses only four different multi-attribute-based unstructured sets. Furthermore, R1-Soar does not form expensive chunks; thus, there was only about a 5–10% change in decisions and run time due to the conversion. Conversions of other Soar tasks are not always expected to be as easy—especially if a particular Soar task makes more extensive use of multi-attributes. In any event, it would obviously be preferable to automate such conversions. We are currently working on this automatic conversion problem in collaboration with an independent research effort in Soar called RTAQ (Yost and Newell, 1989; Yost and Altmann, 1989), which aims at acquiring new tasks (or problem spaces) from external descriptions.

8.2. *Average growth effect*

This article has considered techniques whereby, instead of learning a single expensive chunk, the system may learn a number of individually cheap chunks. For the cheap chunks, the growth of tokens is at worst linear in the number of chunks. However, learning a large number of cheap chunks could clearly overwhelm the system after some time. This effect is called the *average growth effect*—the distortion in Soar's computational model due to the addition of a large number of cheap chunks. The average growth effect is seen in all of the unique-attribute tasks examined in Section 7. For instance, in Figure 17, the accumulation of 142 chunks causes a computational effect of 0.46.

The point to be noted here is that the average growth effect increases the intrinsic or available parallelism in the system (Tambe, et al., 1988). Therefore, we speculate that this problem could be solved by parallelism (Tambe, 1988). More specifically, we expect that with future research in parallel production systems (Gupta, 1986; Gupta and Tambe, 1988; Tambe, et al., 1988), it will be possible to convert the increase in concurrency into real parallelism, allowing Soar to preserve its ideal computational model (modulo the footprint size issue mentioned in Section 5.1).

However, even at this level of speculation about parallelism, some important issues need to be addressed. The first issue is bounding the number of chunks in the system. As in Section 6.2, if we assume a finite lifetime for the problem-solver, then the number of chunks that the system will be able to acquire is bounded. Given the finite lifetime, if the rate of chunking is known, it is possible to estimate the increase in concurrency with chunking over the lifetime. It is then possible to estimate the quantity of processor and memory resources that will be required for converting that increase in concurrency into real parallelism and provide those processor and memory resources ahead of time. Note that the discussion here is only regarding cheap chunks, hence the argument about exponential matches for individual chunks raised in Section 5.1 does not apply.

The previous paragraph raises a second important issue: establishing a polynomial bound on the rate of growth of cheap chunks. An exponential growth in the number of chunks, even with a finite lifetime, would be highly problematical in terms of processor and memory resources. In Soar, the growth of chunks is assumed to be linear in the number of decision cycles—the constant rate of chunking is one of the bases of the chunking theory of learning (Newell and Rosenbloom, 1981). This article is not focused on establishing the validity of this assumption. However, some evidence for the assumption is provided in Figure 23.

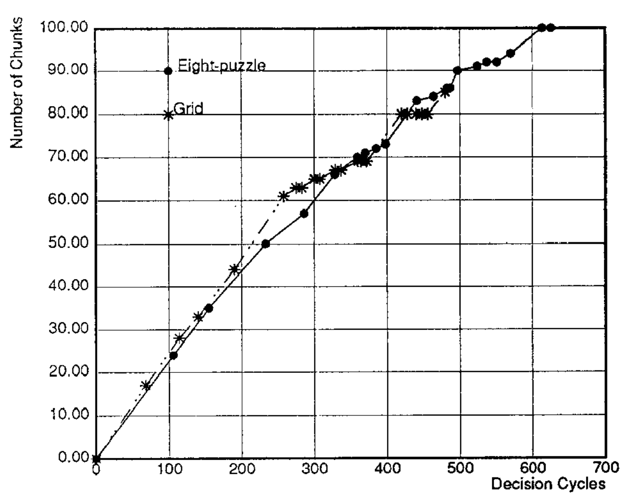


Figure 23. The rate of growth of chunks for the Eight-puzzle and Grid tasks.

It shows the growth in the number of chunks in the two unique-attribute tasks that learn a large number of chunks. For both tasks, the increase in number of chunks plotted against the number of decision cycles for learning on 20 different random problems presented to the system. For the Eight-puzzle these problems are the same as those shown in Figure 21; for the Grid task, the problems are from (Tambe and Rosenbloom, 1988). In both these tasks, the growth of chunks is seen to be linear (or sublinear) in the number of decision cycles. Since these chunks are cheap, the graph indicates that the average growth effect will cause the time per action to grow linearly with the number of decision cycles.

In fact, the following two effects would further reduce the impact of the average growth effect:

1. *Sharing*: As mentioned in Section 3.2, the effect of sharing in general is quite limited in the multi-object based matcher. However, for cheap chunks, this property could become important. For example, in (Prieditis and Mostow, 1987) a problem about learning *membership* of an item in a list is presented. This task leads to learning a large number of chunks, one for each position of the item in the list. However, the chunks have a large number of conditions that are common. All these common conditions get shared if the Rete algorithm is used. Thus, with matching algorithms like Rete, the addition of large numbers of similar chunks requires very little additional processing.
2. *Problem-space decomposition*: Large Soar tasks are composed of many distinct search spaces (called problem spaces). It is unlikely that all the chunks learned will belong to a single one of these problem spaces. Thus, only a fraction of the chunks learned, i.e., those belonging to the current problem space, will be actively matched at a time. The rest will require only one comparison (of their problem space) to determine that they do not belong to the current problem space and will not participate in the match. Furthermore, the total number of problem spaces itself is not fixed—new problem spaces can be constructed at run time (Newell, 1990, Chapter 8), thus reducing the match effort.

Thus, the growth in the processing requirement with chunking is at worst linear, making an effective elimination of the average growth effect with parallelism seem plausible.

8.3. Unique-attributes bound the elaboration phase

Recall from Section 3.1 that a decision cycle in Soar consists of an elaboration phase followed by a decision phase. An elaboration phase consists of multiple elaboration cycles, where multiple productions are fired in each elaboration cycle. With a multi-attribute representation, these elaboration phases are not bounded, i.e., it is possible for elaboration cycles to follow each other in an unbounded fashion in a single elaboration phase. For instance, it is possible to start counting the natural numbers in a single elaboration phase. Figure 24 presents a single production for counting the natural numbers in a single elaboration phase. This production will repeatedly add 1 to the existing value of the attribute *count* and fire with the new value. Note that count is a multi-attribute, which increases in size with each firing of the production.

```
(Production::Count-natural-numbers
      (current-state <x>)
      (state <x> ^count <y>)
-->
      (state <x> ^count (add 1 <y>) ))
```

Figure 24. Counting the natural numbers in a single elaboration phase.

Here, unique-attributes provide another useful bound—they bound the elaboration phase. This bound follows from the following argument. Since the k-search tree of a unique-attribute production has a branching factor of unity, it is clear that a unique-attribute production can generate only a single instantiation at a time (assuming a single goal). Furthermore, the WMEs (and the values) that instantiate the production cannot change within a single elaboration phase, so that new instantiations for this already instantiated production cannot be generated within a single elaboration phase. Since only a single instantiation is generated, a unique-attribute production can fire only once in a single elaboration phase. For instance, with unique-attributes, the production in Figure 24 can fire only once in a single elaboration phase, since only a unique value of count is possible in a single decision cycle. (In reality, this production's action would have to be changed with unique-attributes, since firing the production introduces a multi-attribute.) Thus, the number of productions that can fire in an elaboration phase is bounded by the number of productions in the production system.[21] However, bounding the elaboration phase also implies that arbitrarily large working memory structures cannot be processed in a single decision. These structures require multiple decisions for processing. Again, if these large structures are fixed, then with chunking, the system can learn more productions to deal with them.

8.4. Effect of macro-operator learning on branching factor

A problem related to expensive chunks is that of the increase in the branching factor of a search space with macro-operator learning. This increase in the branching factor can potentially cause a slowdown in a system, as it tries to apply the macro-operators in situations where they are incapable of efficiently solving the problem at hand. In (Mooney, 1989; Iba, 1989; Markovitch and Scott, 1989b; Greiner and Likuski, 1989), various strategies are described to deal with this branching-factor problem, to avoid the slowdown with learning.

The branching-factor problem is related to expensive chunks in that both can cause a slowdown with learning. However, the problem of expensive chunks is not concerned with the branching factor of the search space—it is only concerned with the match for individual productions/chunks. The branching factor problem could arise in Soar, if Soar learns new operators for a problem space, without learning the appropriate search control knowledge for that space. The research in (Mooney, 1989; Iba, 1989; Markovitch and Scott, 1989b; Greiner and Likuski, 1989) could potentially be relevant to Soar in that situation. However, the chunks learned in the various tasks (especially the expensive-chunks tasks) in this paper are search control chunks that select among various existing operators; these tasks do not learn new operators.

8.5. Restricting expressiveness versus enriching expressiveness

In sharp contrast to the research reported in this article, some research efforts have focused on enriching the expressiveness of the rule language. These efforts are aimed at learning iterative concepts, especially by allowing looping constructs in the rule language (Shavlik and DeJong, 1987; Cohen, 1988; Shavlik, 1989). Iterative rules will reduce the average growth effect, since they avoid the necessity of learning multiple rules, one for each value of the loop variable. However, iterative constructs raise difficulties in guaranteeing boundedness of the match process.

In (Shell and Carbonell, 1989), the problem of expensive chunks is explicitly addressed. They are also exploring ways of enriching the operator language to allow iterative and disjunctive macro-operators. As we noted in footnote 2 (on page 1), there are a number of paths that need to be explored to fully understand how to deal with expensive chunks. Furthermore, the advantages and disadvantages of going along different paths remain ill-understood. Thus, the issue of restricting expressiveness versus enriching expressiveness remains an interesting open research issue.

9. Relevance to other research efforts

An important question is the relevance of this research for the non-Soar community. The representation in Soar and the unique-attribute restriction presented in this paper are both based on attribute-value representations. Rule-based systems with attribute-values and a combinatorial match are fairly widespread in AI, e.g., OPS5 (Brownston, Farrell, Kant, and Martin, 1985) and Prodigy (Minton, 1988a; Minton et al., 1989). The unique-attribute representation should map over to some of those systems and help in eliminating combinatorics from the match. Even if the unique-attribute representation does not map over directly, it is possible to look at the idea of restricting k-search in the production match and map that over. Given the tradeoffs involved in the unique-attribute representation, it is not clear exactly when such mappings would be helpful. However, it appears that the mappings might at least be helpful in those situations where excessive k-search is involved.

Unique-attributes appear to be relevant to frame-based systems as well. Recently, Chalasani and Altmann (Chalasani and Altmann, 1989) pointed out that the knowledge representation scheme adopted by Theo, a frame-based architecture for problem-solving and learning (Mitchell et al., 1989), corresponds to the unique-attribute representation. This correspondence makes a plausibility argument for unique-attributes, since it shows that an entire symbolic architecture, considerably different from Soar, is based on the unique-attribute representation. Theo's knowledge-access language is restricted so that it works within the unique-attributes framework. Sets in the knowledge base are represented in Theo in the form of linked lists. Such lists are then processed by user-written Lisp routines (rather than by constructs in the query language).

10. Summary and future work

Expensive chunks are caused by three factors: multi-objects (multi-attributes and preferences), big footprints, and bad condition ordering. Multi-objects allow combinatorial searches to occur in the match. These searches can result in exponential slowdowns. Eliminating multi-objects from the representation bounds the cost of a chunk to be linear in the number of its condition elements. The new restricted representation is referred to as the unique-attribute representation. The principal impact of the unique-attribute representation is the removal of the combinatorics from the matcher. The combinatorics can still occur in the problem space, where they can (in principle) be controlled or terminated using search control knowledge. Analytical and empirical evidence was presented to show that the unique-attribute representation not only guarantees cheap chunks, but it actually eliminates some excessive (and expensive) match. Two issues arise with the representational restrictions in unique-attributes: (1) all sets in working memory have to be structured as lists, trees, or some other task specific structures, (2) the chunks formed are less general. The paper addressed both these issues in some detail.

However, the issue of big footprints remains to be addressed. A chunk can still have an arbitrary number of condition elements. This issue will be one of the subjects of investigation in the near future. The space of match algorithms that would be better suited for the unique-attribute system also needs to be explored. The issue of match algorithms has been addressed in the context of the Soar system with multi-objects (Nayak, Gupta, and Rosenbloom, 1988), with the conclusion that the state-saving Rete algorithm is better suited for Soar than the Treat algorithm, which saves less state. However, the conclusions in the absence of multi-objects are not clear.

The results presented in this paper are based on a variety of toy tasks, such as the Eight-puzzle, Waterjug, and others. To complete the analysis of the impact of unique-attributes, some of the larger Soar tasks need to be converted to the unique-attribute representation. Toward this end, Rl-Soar (Rosenbloom, et al., 1985), a large Soar task with about 450 productions was converted into the unique-attribute representation. The possibility of converting other large Soar tasks, such as Neomycin-Soar (Washington and Rosenbloom, 1988) and Merl-Soar (Hsu, Prietula, and Steier, 1989) to unique-attributes is being investigated.

Another interesting topic for future work is automating the conversion of tasks in Soar from the current representation to the unique-attribute representation. This work will be in collaboration with an independent research effort called RTAQ (Yost and Newell, 1989; Yost and Altmann, 1989), which aims at acquiring new tasks (or problem spaces) from external descriptions. We are currently also investigating other schemes besides unique-attributes for restricting expressiveness (Tambe and Rosenbloom, 1990). We hope that these investigations will allow us to gain a better understanding of the interaction between learning, representation, and efficiency.

Appendix I. Descriptions of the nine tasks

Eight-Puzzle

Problem-statement: There are eight numbered movable tiles in a 3×3 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 3×3 structure of cells to a tile from a dynamic structure of individual tiles.

Operators: 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.

2-Queens

Problem-statement: Placing 2 queens on a 3×3 chessboard such that no queen takes another.

States: The states are represented as a 3×3 array of positions. Each position has one horizontal, one vertical and two diagonal attributes.

Operators: There is only one operator: place-queen. Up to nine instantiations of this operator may be created at a time.

Grid

Problem-statement: Finding a path between two points on a 4×4 grid.

States: The grid-structure with nodes and the paths that connect nodes. The destination node is marked with a desired flag.

Operators: There is only one operator: goto. There can be up to four instantiated goto operators at one time, for moving to the points adjacent to the current position on the grid.

Magic-Square

Problem statement: Completing a 3×3 magic-square. In a magic-square, the sums of the numbers along each of the columns, rows and diagonals equal one another.

States: A state has nine bindings that associate a number with a square.

Operators: There is only one operator for placing a number in a square and it can create up to nine instantiations.

Tree

Problem-statement: Finding a path between the root and a destination point in a tree of height 4 and branching factor of 2.

States: The tree-structure with nodes and the paths that connect nodes. The destination node is marked with a desired flag.

Operators: There is only one operator: goto. There can be up to two instantiated goto operators at one time, for moving to the points adjacent to the current position on the tree.

Syllogisms

Problem-statement: A syllogism is a logic puzzle where two assertions involving pairs of terms (e.g., All P are Q; All Q are R) are given. From these given assertions some conclusion (in this example: All P are R) is to be drawn.

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). See (Polk and Newell, 1988) for details.

Operators: There are four different operators that can add an object, focus on a premise, focus on an object, and augment an object.

Monkeys and Bananas

Problem-statement: A monkey has to get the bananas hung from the ceiling in a room. A ladder is placed under the bananas. The task for the monkey is to climb the ladder and get the bananas.

States: The position of the Monkey in terms of its position on the ground and its height, and the position of the bananas.

Operators: There are five different operators available to the Monkey: 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 three gallon jugs.

Operators: Six operators are available, one for each combination of pouring water between the well and the two jugs. Each operator specifies what container it is pouring water to and what it is pouring water from. Each operator must empty the source or fill the destination container.

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 farmer cannot leave the sheep and the wolf together unattended and cross the river with the cabbage, since the wolf may eat the sheep. Similarly, he cannot leave the sheep and the cabbage together.

States: The status of four different objects: The farmer, the sheep, the wolf, and the cabbage.

Operators: Two operators are available: one for the farmer to cross the river alone and one for the farmer to cross the river with one load. Three or four instantiations of the two operators combined are available in any one state.

Appendix II. Cost and generality of chunks in the grid world

A grid of size $n \times n$ is assumed. In the interests of simplicity, boundary effects are ignored.

Multi-attribute representation

- *Cost*: As shown in Figure 12, each condition element multiplies the number of tokens in the chunk by four, the number of connections emerging from any given point. The cost of the chunk is the total number of tokens generated by the match. If the path length is p, and the cost of the chunk is C_m, then:

$$\begin{aligned} C_m &= 4 + 4^2 + 4^3 + \ldots + 4^p \\ &= \Sigma_{k=1}^{p} 4^k = (4^{p+1} - 4)/(4 - 1) \\ &= (4^{p+1} - 4)/3 \end{aligned}$$

- *Generality*: The chunk transfers to any two points on the grid connected by a path of length p. There are n^2 points on the grid that can act as a source. To calculate the number of destinations possible from a given source, consider Figure 14. The points reached from the source form squares (or euclidean circles), around the source. If the path length is p, then the squares are at a distance of 2, 4, 6, . . . , p (for an even p), and at a distance of 1, 3, 5, . . . , p (for an odd p). Let us call this distance the radius of the square.
  Let k be the radius of a square. Let x and y be the cartesian coordinates of a destination point on the square with the source as the origin. Then the number of possible destinations is obtained by enumerating the solutions of the following equation for integer values of x and y.

$$|x| + |y| = k$$

  There are $4{*}k$ solutions to this equation, since both x and y can vary between $-k$, . . . 0, . . . , k. Thus there are $4{*}k$ destinations on a square of radius k.
  If M is the total number of points that can be reached from a given source, then for an even p (the analysis is very similar for an odd p):

$$\begin{aligned} M &= 1 + 4 * 2 + 4 * 4 + \ldots\ldots + 4 * p \text{ [1 is for the source]} \\ &= 1 + 8\,(1 + 2 + \ldots + p/2) = 1 + p^2 + 2p \\ &= (p + 1)^2 \end{aligned}$$

  The total number of transfers

$$\begin{aligned} &= \text{sources} * \text{destinations per source} \\ &= n^2 * (p + 1)^2 \end{aligned}$$

- *Total cost*: The total cost is the product of the cost per chunk and the number of chunks. Since only one chunk is learned, the total cost is equal to the cost of that one chunk.

Unique-attribute representation

- *Cost*: As shown in Figure 13, the number of tokens in the chunk is linear in the length of the path (p).

- *Generality*: The chunk can transfer to any two points connected in a specific manner, e.g., up-right-up-right. Thus, for one given source, the chunk will transfer to one destination. There are n^2 points on the grid. Therefore the generality is n^2.
- *Total cost*: The total cost (without sharing) is the product of the cost (p) of one chunk and the total number ($(p + 1)^2$) of chunks learned. The total number of chunks learned in this representation is obtained from the discussion in Section 6.1.

Appendix III. Generality of chunks in the Eight-puzzle task

This analysis compares the generality of chunks in three different representations: no-transfer, multi-attributes, and unique-attributes. We assume that some simple search-control chunks are to be learned. Given two possible operators (see Figure 19 for examples of operators in the Eight-puzzle domain), the chunks give a preference to one operator over the other. The operators are evaluated as follows:

- If an operator moves a tile from out-of-place to in-place: + 1
- If an operator moves a tile from in-place to out-of-place: − 1
- If an operator moves a tile from out-of-place to out-of-place: 0

In-place and out-of-place are determined by explicit comparison of the positions of the tiles in the present state and the desired state. If the two operator evaluations result in the same value, then an *indifferent* preference is generated, i.e., the two operators are equally preferable (or unpreferable).

No-transfer chunks. A no-transfer chunk has the format shown in Figure 25. Note that this chunk does not have any variables. It first matches the entire present state, then the entire desired state, then the two operators, and then prefers one operator over the other.

```
    (Production::No-transfer-case-78200

        (Desired-state-position 11  ^tile  1)
        (Desired-state-position 12  ^tile  2)
        (Desired-state-position 13  ^tile  3)
                      .
                      .

        (Present-state-position 11  ^tile  1)
        (Present-state-position 12  ^tile  0)
        (Present-state-position 13  ^tile  2)
                      .
                      .

        (Operator-move-tile  OP1  ^tile  1)
        (Operator-move-tile  OP2  ^tile  2)
-->
        (Preference  Operator-move-tile  OP1  worse-than  Operator-move-tile OP2))
```

Figure 25. A no-transfer chunk in the Eight-puzzle domain.

For each possible present and desired state, we require one such chunk. There are 9! possible desired states in the Eight-puzzle. There are 9! possible present states. The total number of configurations of the present and desired states cannot be simply multiplied together, since depending on the position of the blank-tile, the number of two-operator combinations in each state is different. There are nine possible positions of the blank-tile. For each, the number of present-state and operator combinations need to be computed.

- If the blank-tile is in the center, four operators are available. The four operators can be combined in (4 choose 2 =) 6 ways. The rest of the tiles can be permuted in 8! ways—a total of 6*(8!) ways.
- If the blank-tile is in a corner, only two operators are available. There are four corner-positions for the blank-tile. For each of those, there are 8! permutations of the other tiles—a total of 4*(8!) ways.
- For the other positions of the blank-tile, three operators are available. We can combine them in three (3 choose 2) ways. There are four such positions of the blank-tile, and for each of those, there are 8! permutations of the other tiles—a total of 4*3*(8!) ways.

Combining all the above numbers, we get 9! (for the desired-state) * 8!*(4*3 + 4 + 6) (for the present state) = *7983360* ($\approx$ 8 million) no-transfer chunks to cover the entire space.

Multi-attributes. Consider a multi-attribute equivalent of the chunk in Figure 25. This chunk has the following features: it includes variables and uses the representation presented in Appendix I. Instead of matching the entire present state and desired state, it matches only the relevant tiles. Since it performs the match using multi-attributes and variables, it is completely general in terms of the operators being considered, the present state and the desired state. This implies that only a single chunk should be able to cover the entire space. However, since the two operators being considered in one search control chunk may still evaluate to different values, five different chunks would be required:

- One chunk each would be required for the following combinations of operator evaluations: (−1, −1), (−1, 0), (0, 0), (−1, 1), (0, 1)
- The combination of (1, 1) is not possible, as that would mean two tiles from the present state occupy the same cell in the desired state.

Therefore, five multi-attribute chunks suffice to cover the entire space covered by the $\approx$ 8 million no-transfer chunks.

Unique attributes. Consider a unique-attribute equivalent of the chunk in Figure 25. The analysis is similar to the analysis for the multi-attribute chunks. However, the operators in unique-attribute chunks have to be in specific directions *up*, *down*, *right*, *left*. There are six (4 choose 2) possible combinations of the four directions. Furthermore, for each possible combination of directions, there are 10 possible combinations of evaluations. For instance, one unique-attribute chunk will compare an operator for moving a tile *up* with an evaluation of −1, to an operator moving a tile *down* with an evaluation of 0. Therefore, (6*10 =) 60 unique-attribute chunks are needed to cover the entire space.

Acknowledgments

We thank John Laird, Steve Minton, David Steier, Don Cohen, Bob MacGregor, and members of the Soar group for many interesting discussions on expensive chunks. Jack Mostow and the three anonymous reviewers provided valuable feedback for improving the quality of this paper. We also thank Kathy Swedlow for technical editing.

This research was sponsored by the Defense Advanced Research Projects Agency (DoD) under contract numbers F33615-87-C-1499 and N00039-86C-0033 (via subcontract from the Knowledge Systems Laboratory, Stanford University), by the National Aeronautics and Space Administration under cooperative agreement number NCC 2-538, and by Encore Computer Corporation.

Notes

1. This article is an expanded version of the following two conference papers: (Tambe and Newell, 1988) and (Tambe and Rosenbloom, 1989).
2. The particular scheme for restricting expressiveness presented here is not the only one possible. Various other schemes are also possible (Tambe and Rosenbloom, 1990). However, this scheme remains one of the most interesting ones and deserves extended treatment.
3. Soar/PSM-E is implemented on the Encore multi-processor. For these measurements, the system was run in a uniprocessor mode.
4. Note that in all of the tasks above, where expensive chunks occurred, problem size was reduced to reduce the cost of the chunks. Without such a reduction, the problems would have been intractable after chunking. For instance, the N-queens task was converted to a 2-Queens task.
5. Since the completion of this investigation three other cases of expensive chunks were detected in (Washington and Rosenbloom, 1988), (Hsu, Prietula, and Steier, 1989), and (Reich, 1988). The analysis in this paper applies to those systems as well.
6. The Soar version decribed here and used in the experiments in this article is Soar 4.5 (Laird, et al., 1989).
7. Variations of about a factor of 2 have been seen in time/token. Therefore, a model built on tokens will not be extremely precise. However, the purpose here is to allow us to compare expensive and cheap chunks, independent of the machine implementation. Given the order of magnitude difference in the costs of those chunks, a token-based model seems to serve the purpose.
8. This assertion follows from a detailed analysis of the chunks learned in the tasks from Table 1 and an informal analysis of the productions and chunks in various other Soar tasks. This particular effect usually occurs because these productions/chunks test the current state in the problem space. After a single firing of the production, this state typically changes, removing previously accumulated k-search. Actually, even if the production does not fire, the k-search model manages to estimate the production's match cost. However, the k-search model may not work well if the production fires many different times without requiring any additional k-search.
9. We thank John Laird for this representation.
10. Having variables in the class and attribute fields is very rare in Soar. None of the tasks in this paper use such variabilized class and attribute fields. In fact, variabilization in these two fields is on the verge of elimination from Soar. The only consistent exception to the prebound variable in the identifier field is, obviously, the first condition in the production. This condition is the one that matches the current goal, and hence usually only a single WME—thus it does not change the main conclusion here about multiplicity in matching.
11. These multi-attributes should not be confused with the multi-attributes used in file indexing (Wiederhold, 1987).
12. Written in C, and running on a Vax 8800, which is about 6 MIPs.
13. The large speedup in Magic-square is partly because the original ordering required a large number of tokens, which cluttered up some of the hash tables used in this implementation, increasing the time per token.
14. Some other minor restrictions are also required: (1) Variable attributes must be prebound. (2) Soar's Rete match algorithm needs a small modification to handle conjunctive negations in Soar. The impact of these restrictions is expected to be very limited. See (Tambe and Rosenbloom, 1988) for details.

15. These and other subsequent measurements were done using a Common Lisp version of Soar running on a Vax 8800. Due to the differences in the implementation, and some small change in the Grid and Magic-square task sizes, the execution times of the multi-attributes presented here do not completely correspond with the numbers in Table 1. Also, the measurements here present the total run time as opposed to the match time in Table 1.
16. In addition to the unique- and multi-attributes, a representation that allows no generality whatsoever is also possible. For a comparative analysis of such a representation, see (Tambe and Rosenbloom, 1988).
17. Note that once Soar learns a chunk to solve a problem, it will not create more chunks to solve the same problem. In this particular case, once the unique-attribute system has learned a chunk covering a single path to a destination, it will not learn more chunks covering different paths for the same destination.
18. The large number of chunks (17) result from a variety of subgoals being chunked.
19. For this experiment, problems were generated by randomly walking back (a maximum of 12 steps) from a fixed goal-state. Both the number of steps and the move at each step were randomly generated using the microsecond clock of the Vax 8800.
20. Macros for writing productions may be another way to address this problem.
21. The result of the unique-attribute representation bounding the elaboration phase holds even if multiple goals are present in the goal hierarchy. However, the argument becomes somewhat more complex.

References

Brownston, L., Farrell, R., Kant, E. and Martin, N. (1985). *Programming expert systems in OPS5: An introduction to rule-based programming*. Reading, MA: Addison-Wesley.

Chalasani, P. and Altmann, E. (1989). Comparing the representations in Soar and Theo. Unpublished, School of Computer Science, Carnegie Mellon University.

Chase, M.P., Zweben, M., Piazza, R.L., Burger, J.D., Maglio P.P. and Hirsh, H. (1989). Approximating learned search control knowledge. *Proceedings of International Workshop on Machine Learning* (pp. 218–220).

Cohen, W. (1988). Generalizing number and learning from multiple examples in explanation-based learning. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 256–269).

Cohen, W., Mostow, J. and Borgida, A. (1988). Generalizing number in explanation-based learning. *Proceedings of the Spring Symposium on Explanation-Based Learning* (pp. 68–72).

DeJong, G.F. and Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning, 1*, 145–176.

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

Forgy, C.L. (1984). *The OPS83 Report* (Technical Report CMU-CS-84-133) Pittsburgh, PA: Carnegie Mellon University, Computer Science Department.

Garey, M.R. and Johnson, D.S. (1978). *Computers and intractability: A guide to the theory of NP-completeness*. San Francisco, CA: W.H. Freeman and Company.

Greiner, R. and Likuski, J. (1989). Incorporating redundant learned rules: A preliminary formal analysis of EBL. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 744–749).

Gupta, A. (1986). *Parallelism in production systems*. Doctoral dissertation, Computer Science Department, Carnegie Mellon University. Also a book, Morgan Kaufmann, (1987).

Gupta, A. and Tambe, M. (1988). Suitability of message passing computers for implementing production systems. *Proceedings of the Seventh Conference on Artificial Intelligence* (pp. 687–692).

Hsu, W., Prietula, M. and Steier, D. (1989). Merl-Soar: Scheduling within a general architecture for intelligence. *Proceedings of the Third International Conference on Expert Systems and the Leading Edge in Production and Operations Management* (pp. 467–481).

Iba, G.A. (1989). A heuristic approach to the discovery of macro-operators. *Machine Learning, 3*, 285–318.

Ishida, T. (1988). Optimizing rules in production system programs. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 699–704).

Keller, R. (1987). Defining operationality for explanation-based learning. *Proceedings of the Sixth National Conference on Artificial Intelligence* (pp. 482–487).

Laird, J.E., Swedlow, K.R., Altmann, E.M., Congdon, C.B. and Wiesmeyer, M. (1989). *Soar 4.5 user's manual.* School of Computer Science, Carnegie Mellon University and Department of Electrical Engineering and Computer Science, University of Michigan, June, 1989.

Laird, J.E., Newell, A. and Rosenbloom, P.S. (1987). Soar: An architecture for general intelligence. *Artificial Intelligence, 33* 1-64.

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

Levesque, H.J. and Brachman, R. J. (1985). A fundamental tradeoff in knowledge representation and reasoning. In Brachman, R.J. and Levesque, H.J. (Eds.), *Readings in Knowledge Representation.* Los Altos, CA: Morgan Kaufmann Publishers, Inc.

Markovitch, S. and Scott, P.D. (1988). The role of forgetting in learning. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 459–465).

Markovitch, S. and Scott, P.D. (1989). Information filters and their implementation in the SYLLOG system. *Proceedings of the Sixth International Workshop on Machine Learning* (pp. 404–407).

Markovitch, S. and Scott, P.D. (1989). Utilization filtering: a method for reducing the inherent harmfulness of deductively learned knowledge. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 738–743).

Minton, S. (1985). Selectively generalizing plans for problem-solving. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (pp. 596–599).

Minton, S. (1988). Quantitative results concerning the utility of explanation-based learning. *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 564–569).

Minton, S. (1988). *Learning Effective Search Control Knowledge: An explanation-based approach.* Doctoral dissertation, Computer Science Department, Carnegie Mellon University.

Minton, S., Carbonell, J.G., Knoblock, C.A., Kuokka, D.R., Etzioni, O. and Gil, Y. (1989). Explanation-based learning: A problem solving perspective. *Machine Learning,* Vol. *40* (pp. 63–118).

Miranker, D.P. (1987). Treat: A better match algorithm for AI production systems. *Proceedings of the Sixth National Conference on Artificial Intelligence* (pp. 42–47).

Mitchell, T.M., Allen, J., Chalasani, P., Cheng, J., Etzioni, O., Ringuette, M. and Schlimmer, J.C. (1989). Theo: A framework for self-improving systems. In VanLehn, K. (Ed.), *Architectures for Intelligence.* Hillsdale, NJ: Lawrence Erlbaum Associates.

Mitchell, T.M., Keller, R.M. and Kedar-Cabelli, S.T. (1986). Explanation-based generalization: A unifying view. *Machine Learning, 1,* 47–80.

Mooney, R. (1989). The effect of rule use on the utility of explanation-based learning. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 725–730).

Nayak, P., Gupta, A. and Rosenbloom, P. (1988). Comparison of the Rete and Treat production matchers for Soar (A summary). *Proceedings of the Seventh National Conference on Artificial Intelligence* (pp. 693–698).

Newell, A. (1990). *Unified theories of cognition.* Cambridge, MA: Harvard University Press. In press.

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

Newell, A., Rosenbloom, P.S. and Laird, J.E. (1990). Symbolic architectures for cognition. In M.I. Posner (Ed.), *Foundations of Cognitive Science.* Cambridge, MA: Bradford Books/MIT Press. In press.

Oflazer, K. (1987). *Partitioning in Parallel Processing of Production Systems.* Doctoral dissertation, Computer Science Department, Carnegie Mellon University.

Pereira, L.M. and Porto, A. (1982). Selective backtracking. In Clark, K.L. and Tarnlund, S.A. (Eds.), *Logic Programming.* NY: Academic Press.

Polk, T.A. and Newell, A. (1988). Modeling human syllogistic reasoning in Soar. *Proceedings of the Annual Conference of the Cognitive Science Society* (pp. 181-187).

Preiditis, A. and Mostow, J. (1987). PROLEARN: Towards a Prolog interpreter that learns. *Proceedings of the Sixth National Conference on Artificial Intelligence* (pp. 494–498).

Reich, Y. (1988). Learning plans as a weak method for design. Department of Civil Engineering, Carnegie Mellon University, Unpublished.

Rosenbloom, P.S. and Laird, J.E. (1986). Mapping explanation-based generalization onto Soar. *Proceedings of the Fifth National Conference on Artificial Intelligence* (pp. 561-567).

Rosenbloom, P.S. and Newell, A. (1986). The chunking of goal hierarchies: A generalized model of practice. In Laird, J.E., Rosenbloom, P.S., and Newell, A. (Eds.), *Universal Subgoaling and Chunking: The Automatic Generation and Learning of Goal Hierarchies.* Boston, MA: Kluwer Academic Publishers.

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

Scales, D.J. (1986). *Efficient matching algorithms for the Soar/Ops5 production system* (Technical Report KSL-86-47). Palo Alto, CA: Stanford University, Knowledge Systems Laboratory, Department of Computer Science.

Shavlik, J. (1989). Acquiring recursive concepts with explanation-based learning. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 688–693).

Shavlik, J.W. and DeJong, G.F. (1987). An explanation-based approach to generalizing number. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 236–238).

Shell, P. and Carbonell, J. (1989). Towards a general framework for composing disjunctive and iternative macro-operators. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 596–602).

Smith, D.E. and Genesereth, M.R. (1986). Ordering conjunctive queries. *Artificial Intelligence, 26,* 171–215.

Steier, D.M. (1986). Speeding up Soarware. School of Compuer Science, Carnegie Mellon University, Unpublished.

Steier, D.M. (1987). Cypress-Soar: A case study in search and learning in algorithm design. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (pp. 327–330).

Steier, D.M., Laird, J.E., Newell, A., Rosenbloom, P.S, Flynn, R.A., Golding, A., Polk, T.A., Shivers, O.G., Unruh, A. and Yost, G.R. (1987). Varieties of learning in Soar: 1987. *Proceedings of the Fourth International Workshop on Machine Learning* (pp. 300–311).

Tambe, M. (1988). Speculations on the computational effects of chunking. Computer Science Department, Carnegie Mellon University, Unpublished.

Tambe, M. and Newell, A. (1988). Some chunks are expensive. *Proceedings of the Fifth International Conference on Machine Learning* (pp. 451–458).

Tambe, M. and Rosenbloom, P. (1988). *Eliminating expensive chunks* (Technical Report CMU-CS-88-189). Pittsburgh, PA: Carnegie Mellon University, Computer Science Department.

Tambe, M. and Rosenbloom, P. (1989). Eliminating expensive chunks by restricting expressiveness. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 731–737).

Tambe, M. and Rosenbloom, P. (1990). A framework for investigating production system formulations with polynomially bounded match. *Proceedings of the Eighth National Conference on Artificial Intelligence.* (To appear.)

Tambe, M., Kalp, D., Gupta, A., Forgy, C.L., Milnes, B.G. and Newell, A. (1988). Soar/PSM-E: Investigating match parallelism in a learning production system. *Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming: Experience with applications, languages, and systems* (pp. 146–160).

Ullman, J.D. (1982). *Principles of database systems.* Rockville, MD: Computer Science Press.

Washington, R. and Rosenbloom, P.S. (1988). Applying problem solving and learning to diagnosis. Knowledge Systems Laboratory, Stanford University, December, 1988, Unpublished.

Wiederhold, G. (1987). *File organization for database design.* NY: McGraw-Hill.

Yost, G.R. and Altmann, E.M. (1989). TAQL 3.0: Soar task acquisition system user's manual. School of Computer Science, Carnegie Mellon University, December, 1989, Unpublished.

Yost, G.R. and Newell, A. (1989). A problem space approach to expert system specification. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence* (pp. 621–627).