Eliminating Expensive Chunks by Restricting Expressiveness

Milind Tambe
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
tambe@cs.cmu.edu

Paul Roscnbloom
Information Sciences Institute
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292
rosenbloom@ isi.edu

Abstract

Chunking, an experience based-learning mechanism, improves Soar's performance a great deal when viewed in terms of the number of subproblems required and the number of steps within a subproblem. This high-level view of the impact of chunking on performance is based on an deal computational model, which says that the time per step is constant. However, if the chunks created by chunking are *expensive,* then they consume a large amount of processing in the match, i.e, indexing the knowledge-base, distorting Soar*s constant time-per-stcp model. In these situations, the gain in number of steps does not reflect an improvement in performance; in fact there may be degradation in the total run time of the system. Such chunks form a major problem for the system, since absolutely 10 guarantees can be given about its behavior.

I "his article presents a solution to the problem of expensive chunks. The solution is based on the notion of restricting the expressiveness of Soar's representational language to guarantee that chunks formed will require only a limited amount of matching effort. We analyze the tradeoffs involved in restricting expressiveness and present some empirical evidence to support our analysis.

1. Introduction

The goal of the Soar project is to build a system capable of general intelligent behavior and autonomous existence [6]. Soar is based on formulating all symbolic goal-oriented behavior as search in problem spaces. The primitive acts of the system, called *decisions,* are those required to pursue this search: the selection of problem spaces, states, and operators, plus the application of operators to states to generate new states. The information necessary for the performance of these primitive acts can be provided in one of two ways: by the firing of productions, or by the recursive use of problem space search in subgoals. Both can result in adding new objects to the system's working memory and in adding new information about existing objects (encoded as attributes with values). Soar learns by converting subgoal-based search into productions that generate comparable results under similar conditions. The actions of the new productions are based on the ⌜esults of the subgoals. The conditions are based on those working memory elements (wmes) in parent goals upon which the results iepended. This *chunking* process is a form of explanation-based learning (EBL) [14].

Chunking improves Soar's performance a great deal when viewed in terms of the number of subproblems required and the number of steps within a subproblem f16]. However, despite the gain in lumber of steps, there may be degradation in the total run time of the system. In [18], we showed that this anomaly arises due to *expensive chunks,* i.e., learned productions that consume a large amount of processing in the match. At its worst, chunking can convert a subexponential problem space search into an exponential match problem — as shown in |18], matching expensive chunks is NP-hard.

A concern about degradation in performance due to learning has appeared widely in the EBL literature [4, 5, 9, 10, 21]. Various approaches have been used to deal with this degradation, most focusing on some form of cost-benefit analysis of the learned material. In this article we present a solution to the problem of expensive chunks that is based on restricting the expressiveness of the system's representation language so as to guarantee that all chunks will be cheap. In terms of a cost-benefit approach to learning, this amounts to guaranteeing that the cost of adding a production will always be close to zero. Thus, if the benefits are never less than zero, an explicit on-the-fly cost-benefit analysis can be avoided.

This article is organized as follows: Section 2 describes the problem of expensive chunks in more detail. Section 3 presents our solution to the expensive chunks problem. Section 4 takes an in depth look at the solution for a prototypical expensive chunks task: the Grid task. Section 5 provides further experimental results bearing on our solution. Section 6 presents a counter-point, a task in which our solution is at its worst: the Tree task. Finally, Section 7 summarizes the results and presents some issues for future work.

2. The Problem of Expensive Chunks

Intelligent systems, such as Soar, are based on symbolic architectures [13] mat partition the complete system into two independent 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, problem space search occurs in the cognitive domain. It is a symbolic process that can itself be controlled by further symbol processing. The production match, on the other hand, is part of the implementation domain. It is a fixed process mat runs to completion unaffected by the knowledge in the cognitive domain.

In concert with this distinction, it is possible to describe the affects of learning on task performance via two effects. 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 production actions that are executed, while the computational effect is the change in the time required per action. Table 2-1 (modified from [18]) shows the cognitive effect, the computational effect (computed as time-per-action before learning /time-pcr-action after learning), the total speedup (this is speedup in total match time, ignoring the other processing performed by the architecture) and the number of chunks learned for eight tasks implemented in Soar. These measurements were done on Soar/PSM-E [20], a system that uses a highly optimized implementation of the Rete production matcher.

As expected, chunking provides a large cognitive effect in all of the tasks. Learning causes the number of actions required to drop by a factor of between 5 and 14. For four of the tasks (Syllogisms,

[1]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. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the US Government or the Encore Computer Corporation.

	Cognitive effect	Computational effect	Total Speedup	Nbr of Chunks
8-puzzle	6.53	0.15	0.99	11
N-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	6.20	0.83	5.16	4
Waterjug	9.13	0.57	5.22	11
Farmer	11.09	0.45	5.04	14

Table 2-1: Effects of chunking on performance

Monkey, Waterjug, and Farmer) this cognitive effect is followed by a concomitant speedup in the total match time. However, for the other four tasks (8-Puzzle, N-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. This anomaly occurs because of the computational effect; the time per action for these four tasks increases by as much as a factor of 25 (for the Magic-Square). This computational effect does not come about because of the increased number of chunks — only five chunks are acquired in the Magic-Square — but instead from the presence of individually expensive chunks.

Expensive chunks are a problem for a number of reasons. Firstly, they cause the system to slow down with learning, a clearly undesirable effect. Secondly, they introduce NP-hard search into the implementation domain (the matcher) where it cannot be controlled by additional knowledge. Thirdly, they lead to gross violations of Soar's *ideal computational model,* that the time per step in problem solving should be constant.

To understand the origins of expensive chunks, it is useful to have a means of analyzing the production match that is independent of many of the variations in match algorithms and implementations. The *k-search model [18]* is one such approach that covers match algorithms that find all possible solutions, without the aid of heuristics. This includes widely used match algorithms such as Rete [2] and Treat [11]. The k-search model is based on the notion of tokens, i.e., partial instantiations of productions. Consider the production *length-2* shown in Figure 2-1-a. It contains three conditions and one action. In the figure, the use of ^ indicates an attribute, and the use of o indicates a variable. Figure 2-1-b shows the working memory of the production system, which describes the graph shown in Figure 2-1-c. On the creation of the working memory element *(current-position B*), the production *length-2* will match the working memory, generating tokens, such as: *(2; x = B,* z *= C), (2; x = B, z = D)* etc. The first number in the token indicates the number of conditions matched and the other elements indicate the bindings for the variables. Thus, tokens indicate what conditions have matched and under what variable bindings.

The tokens generated in the match can be represented in the form of a *match* tree, as shown in Figure 2-1-d (at each stage, only the additional variable bindings are shown). This tree represents the search conducted by the matcher, using tokens, to match the production. Since this search is done in the production system, i.e., in the knowledge base, it is called *k-search,* to distinguish it from problem space search. Measurements on Soar/PSM-E indicate that the time spent per token is approximately constant Therefore, for Soar productions, the *number of tokens* in the k-search tree is a reasonable estimate of the work done in performing match.

For a chunk with a k-search tree of depth *H* and a constant branching factor of F, the match cost will be $\sum_{k=1}^{H} F^k > F^H$ tokens. This is exponential in the depth of the tree (H), and if *F* and *H* are

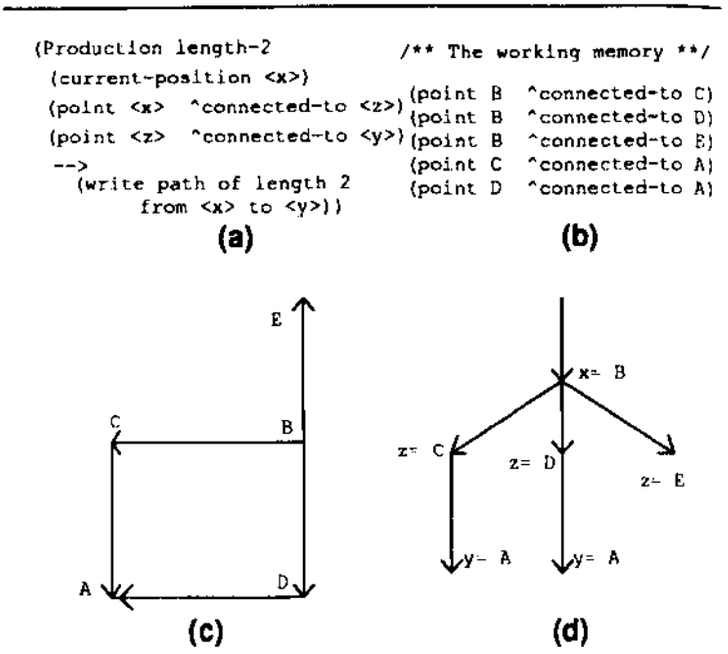


Figure 2-1: An example production system and its k-search tree.

large, the chunk will clearly be expensive. Let's look at both of these factors in a little more detail. (This analysis is a summary of the one presented in [18].) The depth of a production's k-search tree *(H)* is equivalent to the number of conditions in the production. For a chunk, this corresponds to the size of the *footprint;* that is, the number of wmes in the supergoals examined by the problem-solver in producing results of the corresponding subgoal. The branching factor 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 cross-condiuon variable tests. In Soar, cross-condition variable tests are used extensively to implement a *linked-access* restriction. This restriction says that before an object can be tested in a production, a path from a goal to that object (via attributes) must have already been tested. Given this restriction, there remain only two sources of branching factor in k-search trees: bad condition orderings, which can reduce the constraint provided by cross - condition variable tests; and attributes that have multiple values *(multi-attributes).*[2] Multi-attributes are used in representing open, unstructured sets in working memory. For example, in Figure 2-1-b, *connected-to* is a multi-attribute of the object B — points C, D, and E exist as an unstructured set of points connected to point B. Figure 2-1-d illustrates how the k-search tree branches out in matching points C, D and E connected to B.

3. Eliminating Expensive Chunks

Eliminating expensive chunks means that all of the chunks used by the system in problem solving are cheap chunks. An ideal solution would impose a fixed upper bound on the cost of each chunk, allowing the system to achieve a constant time per step. Even if a fixed bound is not possible, it would be preferable 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 and hence will only minimally distort the constant time-per-step model.

Two techniques that initially look like potential solutions can be

[2] Preferences — special control elements in Soar — can also contribute to the branching factor. However, their impact is much smaller than that of multi-attributes, and they will be ignored here. A discussion of the relationship of preferences to expensive chunks can be found in [19].

immediately 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 [18]. A better condition ordering can definitely reduce the amount of k-search (see [18], for example), but cannot completely eliminate it, or even make it non-exponential in general. The same is also unfortunately true for the other smart match algorithms. The other technique that looks like a potential solution is the use of massive parallelism. Given any amount of parallelism it is always possible to have an exponential match that will exceed the capacity of the machine. A third technique, that of hand rewriting of expensive chunks, is ruled out by the need for the system to run autonomously. This is one key way in which the problem of expensive chunks differs from the more general problem of expensive productions. When these three techniques are eliminated, four alternative strategies remain:

1. *Selective learning/for getting:* A cost-benefit analysis is performed before (or after) adding a learned rule into the system's knowledge base. The rule is added to the knowledge base only if the analysis proves to be positive [4, 5, 9, 10].
2. *selective Matching:* The matcher reasons about the expense of the learned rules and decides the best rule to be matched at any point in time. Thus the compiled rule always remains in the knowledge base.
3. *Modifying learned rules:* After learning a rule, its left-hand side is simplified to reduce its match cost. This may be accomplished by processes such as removing applicability conditions [21].
4. *Restricting expressiveness:* The system gives up some expressiveness of its language to guarantee that all learned rules are cheap. This tradeoff is similar to the tradeoff in the expressiveness of a representational language and its computational tractability [8].

In this article, we investigate the strategy of restricting expressiveness for eliminating expensive chunks. There are two obvious candidates for this restriction: restricting the size of the footprint, and restricting the use of multi-attributes. 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 be polynomial in the breadth of the k-search tree. However, if the bound is large the chunks will still not be cheap, and if the bound is small, extensive modifications will be required to the production system and perhaps also to the chunking mechanism. The other alternative — restricting the use of multi-attributes — is more promising. Multi-attributes control the branching factor of the k-search tree. If multi-attributes are eliminated completely, the branching factor of the k-search tree will be reduced to one. This will then limit the number of tokens in the k-search tree to the size of the footprint. Thus, *the cost of a chunk will be linear in the number of conditions.* This conforms to our definition of cheap chunks.

What are the implications of such a restriction? Recall that multi-attributes are used for representing open, unstructured sets in working memory. This allows accessing (or searching) all elements of the set by matching a single production; that is, by k-search rather than by problem space search. For instance, in Figure 2-1-d, all the points connected to B are obtained via k-search. But k-search in the presence of multi-attributes is combinatoric. When combinatorics occur in the k-search, then they cannot be controlled, potentially causing exponential slowdowns.

Thus, the restriction on multi-attributes implies that open, unstructured sets cannot be represented directly in working memory. All sets in working memory have to be structured *(trees, lists or some other task-specific structures).* We refer to the new restricted attributes as *unique-attributes.* The principle impact of going with unique-attributes is the removal of the combinatorial k-search from the matcher — all combinatorics now occur as search in problem spaces. There are some fixed overheads associated with the problem space search of the unique-attributes (selection of states, operators etc.). These overheads can cause the system to slow down by a constant factor. But, the gain in carrying out the search in the problem space lies in the ability to use control knowledge to terminate or control it. Moreover, chunking will gradually reduce and ultimately eliminate this search. Thus, the constant overheads of problem space search will also be eliminated from the unique-attributes.

A secondary impact of unique-attributes is that the chunks that are learned may be less general. This occurs because a multi-attribute condition can match any element of an open set, while a unique-attribute condition can match only one possible element. This issue is examined in detail in the following sections.

Table 3-1 presents the total run time, before and after chunking, with unique-attributes and multi-attributes, for the four expensive chunks tasks in Table 2-1. (Due to the differences in implementations, these results are not direcdy comparable to those in Table 2-1.) The parethesized numbers are the number of chunks learned. 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. We see that the time to complete the task without chunking in both representations is comparable. However, three out of the four tasks with multi-attributes require more time after learning than before it. With the unique-attribute representation, all four tasks speed up, by factors between 3 and 9. This is the case even when significantly more unique-attribute chunks are learned.

	Eight puzzle	N queens	Grid	Magic square
Multi-attribute before chunking	28.69	4.48	23.44	13.92
Multi-attribute after chunking	34.56 (11)	13.52 (3)	12.65 (17)	18.72 (5)
Unique-attribute before chunking	26.67	3.55	19.81	12.62
Unique-attribute after chunking	8.88 (86)	0.40 (3)	3.09 (142)	2.72 (5)

Table 3-1: The total run time in seconds before and after chunking with two different representations.

4. The Grid Task: A Detailed Example

Consider the example from the Grid task, shown in Figure 4-1-a. To simplify our analysis, we assume that this grid is *infinite*. The task is to go from point A, to point B, a path of length four. We will solve this task first using multi-attributes and then using unique-attributes.

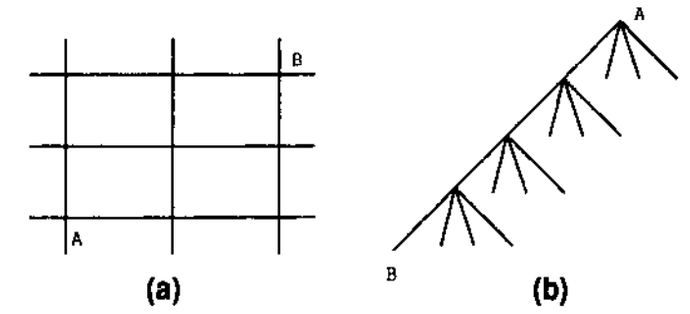


Figure 4-1: The Grid task.

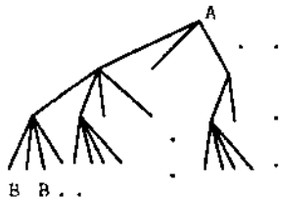
In the multi-attribute version, the grid is represented using *connected* as a multi-attribute of a *point* on the grid. Any point Y

adjacent to a point X on the grid is represented as: *(point X Connected Y).* Thus, this is an unstructured set of connections between a point and all its immediate neighbors. The problem space has only one operator: *move.* 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 4-1-b. This generates 16 tokens, four tokens for each step (even if the heuristics don't work well, the system will generate only four tokens per step).

The chunk formed in solving the task is shown in Figure 4-2-a. 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 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. Figure 4-2-b shows the k-search tree formed in matching the chunk. Each condition multiplies the number of tokens by four, which is the number of points connected to any given point. Since there are four conditions in the chunk (for a path of length four), the total number of tokens is 340 (= 4 + 16 + 64 + 256) tokens. Comparing this with the four tokens per step in the original problem solving, we see mat the chunk is expensive. The 340 tokens correspond to all possible paths originating from point A that have a length of four.

```
(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>))
```

(a)



(b)

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

(c)

(d)

Figure 4-2: The Grid task with multi-attributes.

In the unique-attribute version, each location points to its four adjacent locations using specific unique-attributes: *up, down, left* and *right.* Instead of one operator *move,* there are four distinct 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 4-1-b. The chunk formed with this representation is shown in Figure 4-2-c. It says that if the goal is to move to a point <d> from the point <x> and if the connection between the two points is through the specific relation described *(up-right-up-right),* then choose the appropriate operator: *move-up.* The k-search tree formed is shown in Figure 4-2-d. There are only four tokens formed in this case — 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.

Table 4-1 summarizes the cost and generality of the two representations. The generality is measured in terms of the number of transfers in an *nxn* grid. The length of the path traversed in the grid is assumed to be/?. Comparing the multi-attributes and unique-attributes, we see that a single multi-attribute chunk has a level of generality that is $(p+1)^2$ times more than the generality achieved by a single unique-attribute chunk. Thus to achieve the same performance, 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 the multi-attribute system; i.e., the multi-attributes have produced an expensive chunk.

	Cost in tokens per chunk	Generality in transfers per chunk	# chunks for same generality	Cost in tokens with all chunks
Multi	$(4^{p+1}-4)/3$	$n^2*(p+1)^2$	1	$(4^{p+1}-4)/3$
Unique	p	n^2	$(p+1)^2$	$(p+1)^2*p$

Table 4-1: The cost and generality of various representations for the Grid task with a fixed p.

To understand why the unique-attribute version has an advantage, consider a problem with a path length $p = 4$. There are only a polynomial number of destinations on the grid that can be reached from a source: $(p+1)^2 - 25$. However, there are an exponential number of paths from the source to these destinations: $4^P = 256$. When given the goal of reaching one particular position, the expensive chunk tries to find all possible paths of length four. It discovers all 256 paths to all of the 25 positions; an excessive amount of k-search, since only one path to one position is required. This k-search of all paths to each of the positions gives rise to the exponential factor. On the other hand, the chunks acquired by the unique-attribute version learn only one path to each of those positions. This avoids the useless computation of finding all paths to each position. Even after learning all the 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). Thus, the multi-attribute representation uses k-search to gain generality. However, in the grid example, much of the generality is essentially superfluous. This excessive generality is a typical characteristic of expensive chunks[18]. A unique-attribute system is able to avoid such excessive k-search, which ultimately delivers only superfluous generality.

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, 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 [7].
2. *decomposition:* If a task is decomposed into smaller subtaslcs, then chunking the smaller subtasks independently provides anotner source of generality.

5. Experimental Analysis

Using two tasks from Table 2-1 — Grid and 8-Puzzle — this section experimentally demonstrates how restricting multi-attributes eliminates expensive chunks and excessive k-search. To this end, each task was run independently with multi-attributes and with unique-attributes. For each representation and task, the system was first run without chunking. The system was then allowed to chunk on the problem, after which it was then run on the same problem; that is, after having chunked on the problem. A sequence of such experiments was performed with the unique-attributes representation to accumulate a set of chunks yielding the same level of generality as achieved with the multi-attribute representation.

Figure 5-1 shows the computational effect — the change in time per action — with the addition of chunks for the two tasks (the details of the representations used can be found in [19]). The first graph in Figure 5-1 is from the Grid task. The multi-attribute representation learns 17 chunks (the large number of chunks — 17 — is caused by the variety of subgoals that are being chunked) and causes a computational effect of about .11; that is the time per action has gone up by about a factor of 9. The unique-attribute version requires learning on more problems to reach the same level of generality (each star (*) represents one problem). In the process, it accumulates 142 chunks, but the computational effect is much more limited (about .42). The graph for the 8-Puzzle can be interpreted in a similar manner.

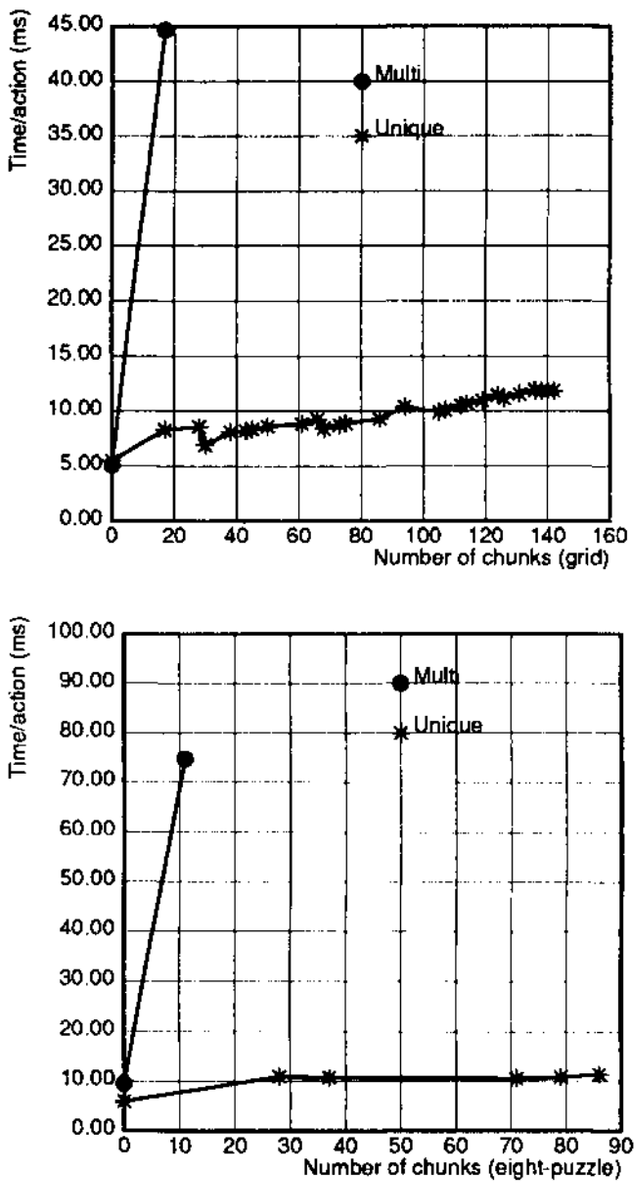


Figure 5-1: Time/action for the Grid and 8-Puzzle tasks.

From the two graphs, we see that though the unique-attribute version may require more chunks to gain the same generality, it posits a big gain by avoiding excessive k-search (graphs for tokens per action present a similar picture). The two graphs also show quite clearly the ability of the unique-attribute version to stay relatively close to the ideal computational model of constant time per step, and the inability of the multi-attribute version to do so. The reason that the unique-attributes version isn't completely consonant with the ideal computational model is that, though the chunks are individually cheap, each one does add something to the total match cost. We refer to this as the *average growth effect,* the distortion in Soar's computational model due to the addition of a large number of chunks. One potential solution to this problem is parallelism. Recent work has shown that chunking increases the concurrency, or available parallelism, in the system [17]. We expect that with future research in parallel production systems [3, 20], it will be possible to convert the increase in concurrency into *real parallelism,* allowing Soar to preserve its constant time per step model (if the number of chunks remains bounded (see Section 6)).

Table 3-1 showed how the unique-attribute version is more efficient after reaching the same level of generality as the multi-attribute version, but it didn't show the finer-grained behavior of what happens during the learning process. Specifically, it didn'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 8-Puzzle domain were run under both versions. Both versions started off with zero 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.

The first graph in Figure 5-2 shows the number of decisions required by the two versions over the sequence of problems. Decisions arc typically used in Soar to measure the amount of search carried out by the system. The curves show that initially both systems take a large number of decision cycles to reach the goal. However, this pattern changes quickly as the increased generality provided by the multi-attribute chunks reduces by a greater amount the number of decisions required for the problems. As more problems are solved, and more unique-attribute chunks are acquired, the difference in number of decisions gets increasingly small. The second graph in Figure 5-2 compares the amount of total time required by the two versions of the 8-Puzzle task. In contrast to the picture presented for decision cycles, here the performance of the unique-attribute version completely dominates the performance of the multi-attribute version.

The first graph in Figure 5-3 shows the time per decision for the two systems. The point corresponding to the zeroth problem corresponds to 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. The second graph in Figure 5-3 shows the cumulative times for the two systems on the 20 problems. The unique attribute system coasistently outperforms the multi-attribute system. Very similar results were obtained for the grid task (see [19].)

6. The Tree Task: A Counterpoint

In the Grid task there was a trade-off between expressiveness and efficiency, but the amount of efficiency gained far outweighed the amount of expressiveness lost. However, this is not always the case. In the worst case there is a one-one trade-off between the two factors. A good example of this is the Tree task. The Tree task is just like the Grid task except that the structure to be searched is a tree, and the path to be found is always from the root to one of the leaves. In this task, a single multi-attribute chunk, learned for one branch of the tree, can transfer to all other branches of the tree. A unique-attribute chunk, on the other hand, remains specific to a particular branch of the tree. Table 6-1 presents the cost and the generality of the chunks learned in this task, for a tree of depth three and a branching factor of two.

One observation that can be made from this table is that the cost of matching all eight unique-attribute chunks (one chunk per path) is equal to the cost of matching the one multi-attribute chunk. Thus,

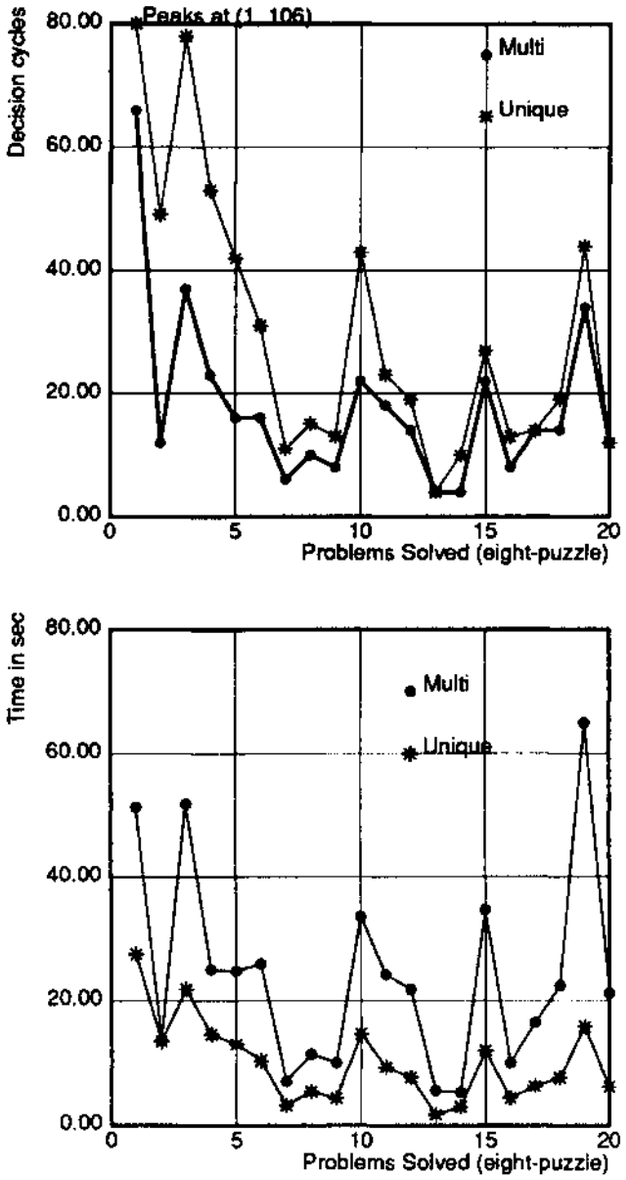


Figure 5-2: Decision cycles and time for the 8-Puzzle task.

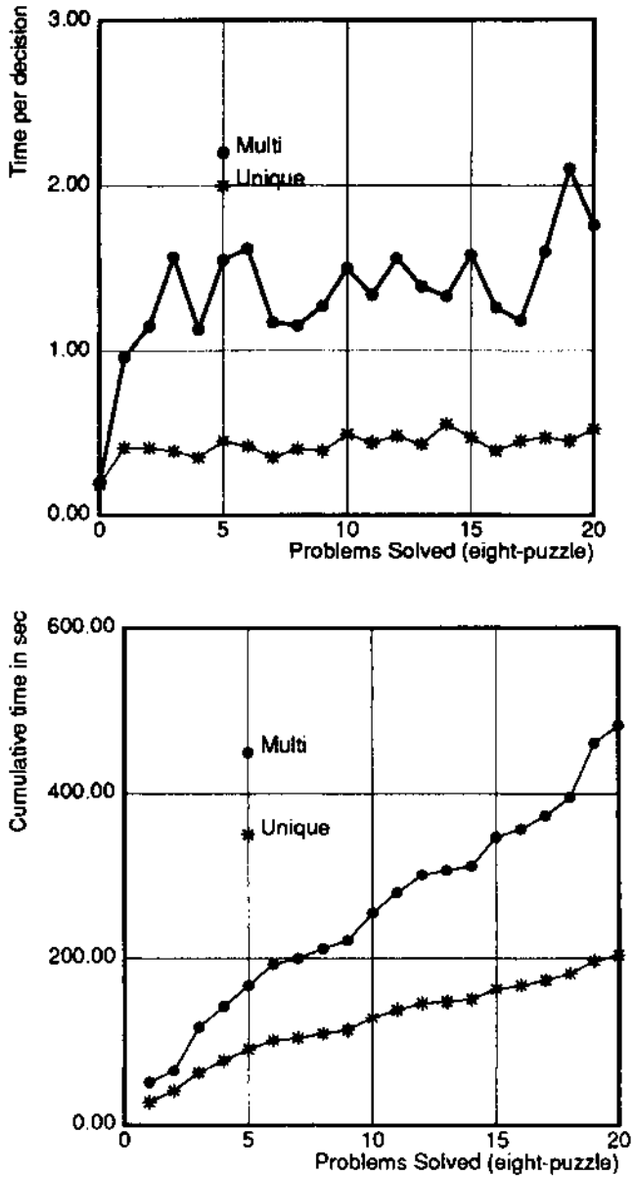


Figure 5-3: Time/decision and cumulative time for the 8-Puzzle task.

there is no excessive k-search involved in the multi-attribute chunk. A second observation is that the lower generality of the unique-attribute chunks demands an exponential number of chunks to cover the level of generality of one multi-attribute chunk. The obvious question that this 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 protection. 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 experienced. The match cost always increases gradually (at worst), and remains bounded by the number of branches that have been experienced. At worst the number of branches that have been experienced is equal to the number in the tree, but in many domaias only a small portion of the entire exponential space is ever experienced.

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, then there is a

	Cost in tokens per chunk	Generality in transfers per chunk	# chunks for same generality	Cost in tokens with all chunks
Multi	8	8	1	8
Unique	3	1	8	8

Table 6-1: The cost and generality of various representations for the tree task.

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).

How often is this worst case likely to arise? Observe that the large (exponential in the depth) tree structure must be present in working memory for performing this task. An exponential amount of time must be spent in generating such an exponential structure. It is unlikely that the problem solver will be able to generate very large structures in its lifetime. On the contrary, in the earlier grid

example, the size of the structure is limited, but the matching of the structure requires exponential time (due to the connectivity). This provides some evidence that the worst case shown here is unlikely to occur; instead, it is more likely for the grid world situations to occur. In fact, none of the tasks from Table 2-1 require a one-one expressiveness-efficiency tradeoff.

Thus, in the unlikely worst case (the tree search), an expressiveness-efficiency tradeoff exists. However, in the general case, multi-attribute chunks generate excessive k-search. We cannot get rid of this excessive k-search with multi-attributes, since the match problem is inherently intractable. There is no tradeoff involved in this excessive search, the unique-attribute approach simply gets rid of it.

7. Summary and Future Work

Expensive chunks are caused by three factors: multi-attributes, big footprints and condition ordering. Multi-attributes allow combinatorial searches to occur in the match. These searches can result in exponential slowdowns. By eliminating multi-attributes, i.e., by restricting the expressiveness of the production system, we were able to limit the cost of a chunk to be linear in the number of its conditions. The principal impact of the unique-attribute representation is the removal of combinatories from the matcher (when combined with the linked-access restriction). The combinatorics now occur in the problem space, where they can be controlled using control knowledge. We provided some empirical evidence to show that our solution not only guarantees cheap chunks, but also gets rid of excessive (and expensive) generality. From a cost-benefit point of view, our solution may be construed as guaranteeing a close to zero cost per chunk.

The restriction on expressiveness need not translate into a difficulty in performing tasks in Soar. Though set operations cannot be performed easily at the production-system-language level, it is possible to perform the tasks at the problem-solving level, using states and operators. We have implemented a set of operators which can perform most of the common set operations, and expect these to effectively replace the functionality provided by multi-attributes.

The approach presented in this article is related to the operationality-generality tradeoff discussed in the EBL literature — some expressiveness is sacrificed to form cheap chunks. However, the system is also able to avoid excessive k-search, which docs not involve a tradeoff. In addition, other sources of generality available to the system arc not involved in the tradeoff.

Recently, Chalasani and Altmann [1] provided some more evidence in favor of unique-attributes. They pointed out that the knowledge representation adopted by Theo, a frame-based architecture for problem-solving and learning [12J, corresponds to the unique-attribute representation. Theo\s knowledge access language is restricted so that it works within the unique-attributes framework. Multi-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).

A number of issues remain to be addressed. An important issue is the usability and utility of unique-attributes in more complex task domains. Toward this end, we have recently converted RI-Soar [15], part of an expert system for computer configuration, into the unique-attribute representation. RI-Soar uses only four different multi-attribute sets. Hence, the conversion turned out to be relatively easy, requiring only two man days. Furthermore, RI-Soar does not form expensive chunks; thus, there was only about 5-10% change in decisions and run time due to the conversion. Additional large-scale tasks will also need to be converted for a full evaluation. A second issue is whether the unique-attributes restriction is the most appropriate one. Though it clearly removes expensive chunks, there are several related restrictions, some more restrictive and some less, which are also worth investigating. We hope that this will allow us to gain a better understanding of the interaction between learning, representation and efficiency.

Acknowledgements

We thank John Laird, Allen Newell, Steve Minton, David Steier, Don Cohen, Prasad Chalasani, Bob MacGregor and members of the Soar group for many interesting discussions on expensive chunks. We also thank Kathy Swedlow for technical editing.

References

[1] Chalasani, P., Altmann, A. Comparing the representations in Soar and Theo. Unpublished.

[21 Forgy, C. L. "Rete: A fast algorithm for the many pattern/many object pattern match problem". *Artificial Intelligence 19,* 1 (1982), 17-37.

[31 Gupta, A. *Parallelism in Production Systems.* Ph.D. Th., Computer Science Department, Carnegie Mellon University, March 1986.

[41 Iba, G.A. A Heuristic approach to the Discovery of Macro-Operators. Tech. Rept. TR88-506.1, GTE Computer and Intelligent Systems Laboratory, January, 1988.

[5] Keller, R. Defining operational ity for explanation-based learning. Proceedings of the National Conference on Artificial Intelligence, August, 1987, pp. 482-487.

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

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

[81 Levesque, H. J. and Brachman, R. J. A fundamental tradeoff in knowledge representation and reasoning. In *Readings in Knowledge Representation,* Brachman, R. J. and Levesque, H. J., Eds., Morgan Kaufmann Publishers, Inc., Los Altos, California, 1985, pp. 42-69.

[9] Minton, S. Selectively generalizing plans for problem-solving. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, August, 1985, pp. 596-599.

[10] Minton, S. *Learning Effective Search Control Knowledge: An explanation-based approach.* Ph.D. Th., Computer Science Department, Carnegie Mellon University, March 1988.

[11] Miranker, D. P. Treat: A better match algorithm for AI production systems. Proceedings of the National Conference on Artificial Intelligence, August, 1987, pp. 42-47.

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

[13] Newell, A., Rosenbloom, P. S., & Laird, J. E. Symbolic architectures for cognition. In *Foundations of Cognitive Science,* M. I. Posner, Ed., Bradford Books/MIT Press, Cambridge, MA, 1989, ch. 3. In press.

[14] Rosenbloom, P. S. and Laird, J. E. Mapping explanation-based generalization onto Soar. Proceedings of the National Conference on Artificial Intelligence, August, 1986.

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

[16] 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. Varieties of learning in Soar: 1987. Proceedings of the Eourth International Workshop on Machine Learning, June, 1987, pp. 300-311.

[17] Tambe, M. Speculations on the Computational Effects of Chunking. Unpublished.

[18] Tambe, M. and Newell, A. Some chunks are expensive. Proceedings of the Fifth International Conference on Machine Learning, June, 1988, pp. 451-458.

[19] Tambe, M. and Rosenbloom, P. Eliminating Expensive Chunks. Tech. Rept. CMU-CS-88-189, Computer Science Department, Carnegie Mellon University, November, 1988.

[20] Tambe, M., Kalp, D., Gupta, A., Forgy, C.L., Milnes, B.G., and Newell, A. 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, July, 1988, pp. 146-160.

[21] Zweben, M. and Chase, M. Improving operationality with approximate heuristics. Proceedings of the Spring Symposium on Explanation-Based Learning, March, 1988, pp. 100-107.