Matching 100,000 Learned Rules

Robert B. Doorenbos

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
Robert.Doorenbos@CS.CMU.EDU

Abstract

This paper examines several systems which learn a large number of rules (productions), including one which learns 113,938 rules — the largest number ever learned by an AI system, and the largest number in any production system in existence. It is important to match these rules efficiently, in order to avoid the machine learning *utility problem*. Moreover, examination of such large systems reveals new phenomena and calls into question some common assumptions based on previous observations of smaller systems. We first show that the Rete and Treat match algorithms do not scale well with the number of rules in our systems, in part because the number of rules affected by a change to working memory increases with the total number of rules in these systems. We also show that the sharing of nodes in the beta part of the Rete network becomes more and more important as the number of rules increases. Finally, we describe and evaluate a new optimization for Rete which improves its scalability and allows two of our systems to learn over 100,000 rules without significant performance degradation.[1]

1. Introduction

The goal of this research is to support large learned production systems; i.e., systems that learn a large number of rules. Examination of such systems reveals new phenomena and calls into question some common assumptions based on previous observations of smaller systems. In large systems it is crucial that we match the rules efficiently; otherwise the systems will be very slow. In particular, we don't want the match cost to increase significantly as new rules are learned. Such an increase is one cause of the *utility problem* in machine learning (Minton, 1988) — if the learned rules slow down the matcher, the net effect of learning can be to slow down the whole system, rather than speed it up. For example, learned rules may reduce the number of basic steps a system takes to solve problems (e.g., by pruning the search space), but the slowdown in the matcher increases the time per step, and this can outweigh the reduction in the number of steps. This has been observed in several machine learning systems (Minton, 1988; Etzioni, 1990; Tambe, Newell, & Rosenbloom, 1990; Cohen, 1990; Gratch & DeJong, 1992).

This paper examines several systems which learn a large number of rules, including one which learns 113,938 rules — the largest number ever learned by an AI system, and the largest number in any production system in existence. Recent work on large production systems has investigated their integration with databases (Sellis, Lin, & Raschid, 1988; Acharya & Tambe, 1992; Miranker et al., 1990). Much of this work is aimed at scaling up production systems to have large working memories, but only a relatively small number of rules. Scalability along the dimension of the number of rules, on the other hand, has been largely neglected. This dimension is of interest for machine learning systems, as noted above; it is also of interest for another class of production systems: those used for cognitive models. In such systems, the productions model human long-term memory. Since the capacity of human long-term memory is vast, production systems used for cognitive models may require a very large number of rules.

(Doorenbos, Tambe, & Newell, 1992) examined various aspects of a single system, Dispatcher-Soar, which learned 10,000 rules; no increase in match cost was observed. The current paper focuses entirely on matching, and studies three other systems in addition to Dispatcher-Soar; the four systems learn 35,000-100,000 rules. We begin by showing that the best currently available match algorithms, Rete (Forgy, 1982) and Treat (Miranker, 1990), do not scale well with the number of rules. Section 2 shows that using the standard Rete algorithm, all these systems suffer a substantial increase in match cost. This was previously overlooked in Dispatcher-Soar due to the smaller number of rules learned and the way match cost was measured.

Previous studies of smaller production systems suggest that Rete and Treat *should* scale well with the number of rules, because only a few rules are affected by changes to working memory, no matter how many rules are in the system. Section 3 shows that this does not hold in any of the four systems studied here. Since the work done by the

[1]The research was sponsored by the Avionics Laboratory, Wright Research and Development Center Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597, and by the National Science Foundation under a graduate fellowship award. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. government.

Treat algorithm is at least linear in the number of affected productions, we conclude that Treat would not scale well with the number of rules in these systems. We suggest reasons for this difference in the number of affected productions, and note its possible implications for parallel production systems.

In Section 4, we examine the sharing of nodes in the beta part of the Rete network and show that it becomes increasingly important as the number of rules increases. This sharing has previously been considered relatively unimportant, and some match algorithms have been designed without incorporating it. Our results here demonstrate that for certain classes of large learning systems, efficient match algorithms must incorporate sharing.

Finally, we show that the scalability of the Rete algorithm can be improved by applying a new optimization, *right unlinking*, which eliminates one source of slowdown in the match as the number of rules increases. Section 5 describes and evaluates this optimization, which can improve the scalability of Rete and reduce the match cost by a factor of ~40. In Section 6, we suggest ways to reduce or eliminate other sources as well.

The results in this paper concern several large learning systems, each implemented using Soar (Laird, Newell, & Rosenbloom, 1987; Rosenbloom et al., 1991). Soar provides a useful vehicle for this research because in addition to providing a mechanism for learning new rules (*chunking*), it incorporates one of the best existing match algorithms (Rete). Soar is an integrated problem-solving and learning system based on formulating every task as search in *problem spaces*. Each step in this search — the selection of problem spaces, states and operators, plus the immediate application of operators to states to generate new states — is a *decision cycle*. The knowledge necessary to execute a decision cycle is obtained from Soar's knowledge base, implemented as a production system. If this knowledge is insufficient to reach a decision, Soar makes use of recursive problem-space search in subgoals to obtain more knowledge. Soar learns by converting this subgoal-based search into *chunks*, productions that immediately produce comparable results under analogous conditions (Laird, Rosenbloom, & Newell, 1986).

Four large learning systems are examined in this paper. The first system, Dispatcher-Soar (Doorenbos, Tambe, & Newell, 1992), is a message dispatcher for a large organization; it makes queries to an external database containing information about the people in the organization. It uses twenty problem spaces, begins with ~1,800 rules, and learns ~114,000 more rules. The second system, Assembler-Soar (Mertz, 1992) is a cognitive model of a person assembling printed circuit boards. It is smaller than Dispatcher-Soar, using six problem spaces, starting with ~300 rules and learning ~35,000 more. Neither of these systems was designed with these experiments in mind. In addition to these two "natural" systems, two very simple "artificial" systems, Memory1 and Memory2, were built for these experiments. They each use two problem spaces (the minimum necessary for learning in Soar) and are based on the memorization technique described in (Rosenbloom & Aasman, 1990); they differ in the structure of the objects being memorized, with the learned rules in Memory1 more closely resembling some of those learned in Dispatcher-Soar, and those in Memory2 resembling Assembler-Soar. The two systems learned ~102,000 rules and ~60,000 rules, respectively. For each system, a problem-generator was used to create a set of problem instances in the system's domain. The system was then allowed to solve the sequence of problems, learning new rules as it went along.

2. Performance of Standard Rete

Before presenting performance results, we briefly review the Rete algorithm. As illustrated in Figure 1, Rete uses a dataflow network to represent the conditions of the productions. The network has two parts. The alpha part performs the constant tests on working memory elements; its output is stored in *alpha memories* (AM), each of which holds the current set of working memory elements passing all the constant tests of an individual condition. The beta part of the network contains *join nodes*, which perform the tests for consistent variable bindings between conditions, and *beta memories*, which store partial instantiations of productions (sometimes called *tokens*). When working memory changes, the network is updated as follows: the changes are sent through the alpha network and the appropriate alpha memories are updated. These updates are then propagated over to the attached join nodes (*activating* those nodes). If any new partial instantiations are created, they are propagated down the beta part of the network (activating other nodes). Whenever the propagation reaches the bottom of the network, it indicates that a production's conditions are completely matched.

The figure is drawn so as to emphasize the *sharing* of nodes between productions. This is because our focus here is on matching a large number of productions, not just a few, and in this case the sharing becomes important, as we demonstrate in Section 4. When two or more productions have a common prefix, they both use the same network nodes. Due to sharing, the beta part of the network shown in Figure 1 forms a tree. (In general, it would form a forest, but by adding a dummy node to the top of a forest, we obtain a tree.)

The figure also omits the details of the alpha part of the network (except for the alpha memories). In this paper we focus only on the beta part of the network. Previous studies have shown that the beta part accounts for most of the match cost (Gupta, 1987), and this holds for the systems examined here as well.

Figure 2 shows the performance of the basic Rete algorithm on our systems. For each system, it plots the average match cost per decision cycle as a function of the

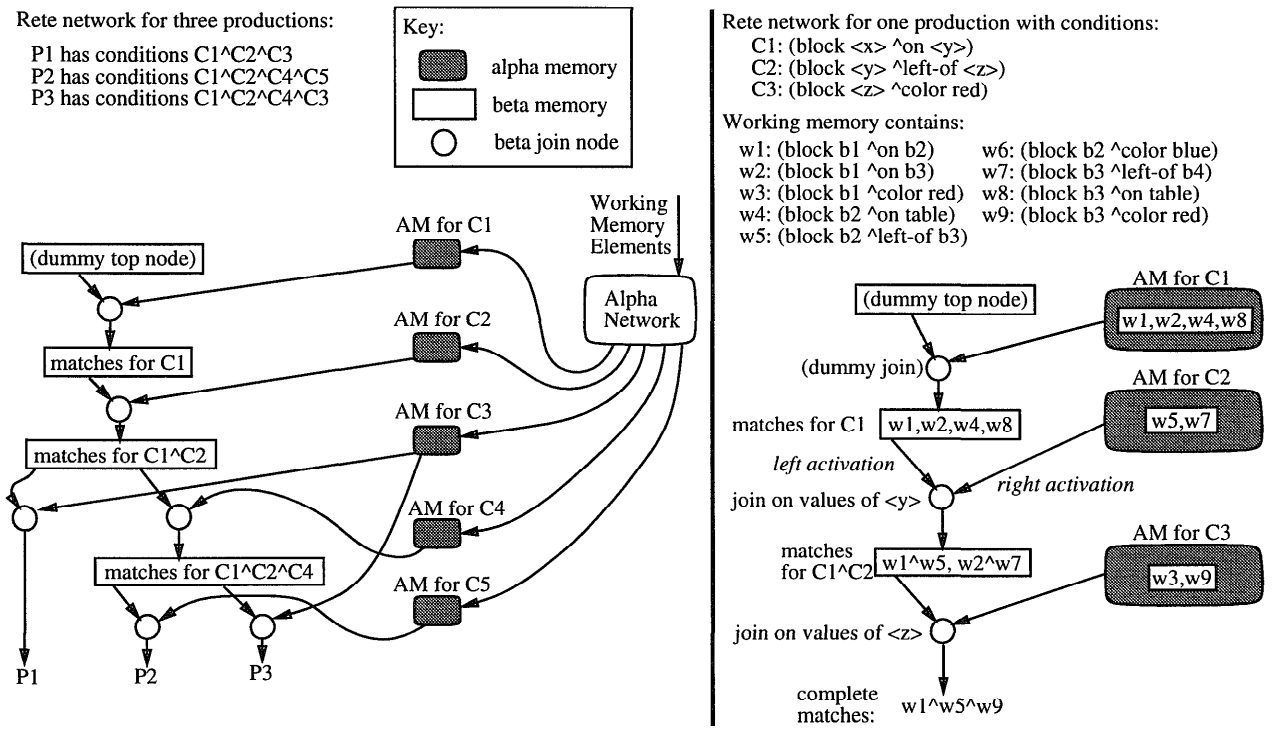

Figure 1: Rete network for several productions (left) and instantiated network for one production (right).

number of rules in the system.[2] It clearly demonstrates that as more and more rules are learned, the match cost increases significantly in all four systems. Thus, the standard Rete algorithm does not scale well with the number of learned rules.

3. Affect Set Size

The data shown in Figure 2 may come as a surprise to readers familiar with research on parallelism in production systems. This research has suggested that the match cost of a production system is limited, *independent* of the number of rules. This stems from several studies of OPS5 (Forgy, 1981) systems in which it was observed that only a few productions (20-30 on the average) were affected by a change to working memory (Oflazer, 1984; Gupta, 1987). A production is *affected* if the new working memory element matches (the constant tests of) one of its conditions. This small *affect set size* was observed in systems ranging from ~100 to ~1000 productions. Thus, the match effort is limited to those few productions, regardless of the number of productions in the whole system.

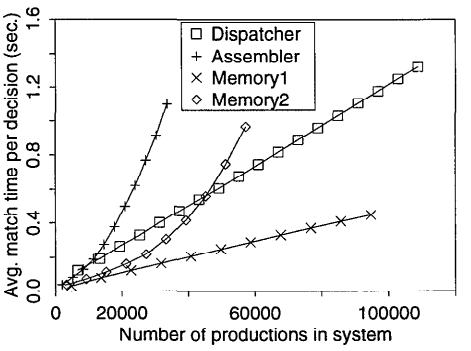

Figure 2: Match time with standard Rete.

This result does not hold for the systems we have studied. Figure 3 shows a lower bound[3] on the number of productions affected per match cycle (recognize-act cycle) for the four systems, plotted as a function of the number of productions. Each point on the figure is the mean taken over 100,000 match cycles. (Recall that the

[2]The *decision cycle* is the natural unit of measurement in Soar; each decision cycle is a sequence of a few match cycles (recognize-act cycles). The numbers reported here are based on Soar version 6.0, implemented in C, running on a DECstation 5000/200. The Rete implementation uses hashed alpha and beta memories (Gupta, 1987); the alpha part of the network is implemented using extensive hashing so that it runs in approximately constant time per working memory change.

[3]The graph shows that the lower bound is linearly increasing. An upper bound on the affect set size is the total number of productions, which of course is also linearly increasing. Measuring the *exact* affect set size is computationally expensive, but since both the lower bound and upper bound are linearly increasing, one can assume the actual mean affect set size is linearly increasing also — if it weren't, it would eventually either drop below the (linearly increasing) lower bound, or else rise above the (linearly increasing) upper bound.

number of productions is gradually increasing as the system learns. The exact affect set size varies greatly from one cycle to the next.) In each system, the average affect set size increases fairly linearly with the number of productions. Moreover, each system grows to where the average affect set size is ~10,000 productions — considerably more than the total number of productions any of them started with.

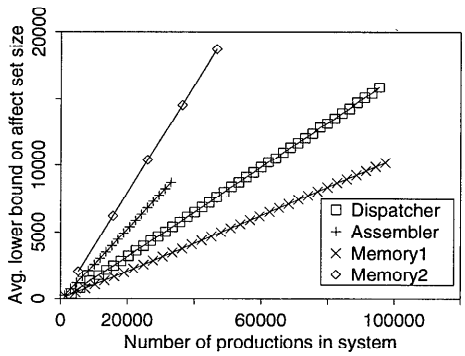

Figure 3: Affect set size.

Why does the affect set size increase in these systems but not in the aforementioned OPS5 systems? (Gupta, 1987) speculates: "The number of rules associated with any specific object-type or any situation is expected to be small (McDermott, 1983). Since most working-memory elements describe aspects of only a single object or situation, then clearly most working-memory elements cannot be of interest to more than a few of the rules." While this reasoning may hold for many OPS5 systems, it does not hold for systems that extensively use problem space search. In such systems, a few working memory elements indicate the active goal, state (or search node), and operator; these working memory elements are likely to be of interest to many rules. In the Soar systems studied here, many rules have the same first few conditions, which test aspects of the current goal, problem space, state, or operator. This same property appears to hold in Prodigy, another search-oriented problem-solving and learning system, for the rules learned by Prodigy/EBL (Minton, 1988). Whenever a working memory element matching one of these first few conditions changes, a large number of rules are affected. More generally, if a system uses a few general working memory elements to indicate the current problem-solving context, then they may be of interest to a large number of rules, so the system may have large affect sets.

(Gupta, 1987) also speculates that the limited affect set size observed in OPS5 programs may be due to particular characteristics of human programmers; but these need not be shared by machine learning programs. For example, people often hierarchically decompose problems into smaller subproblems, then write a few rules to solve each lowest-level subproblem. Consequently, the few working memory elements relevant to a given subproblem affect only those few rules used to solve it. However, if we add a knowledge compilation mechanism to such a program, it may generate rules that act as "macros," solving many subproblems at once. This would tend to increase the number of rules in the system affected by those working memory elements: they would now affect both the original rules and the new macro-rules.

More work needs to be done to better understand the causes of a large affect set size. However, the above arguments suggest that this phenomenon is likely to arise in a large class of systems, not just these particular Soar systems.

Note that a limited affect set size is considered one of the main reasons that parallel implementations of production systems yield only a limited speedup (Gupta, 1987). The results here suggest that parallelism might give greater speedups for these systems. However, if sequential algorithms can be optimized to perform well in spite of the large number of affected productions, then speedup from parallelism will remain limited. So the implications of these results for parallelism are unclear.

4. The Importance of Sharing

Given the increase in the number of productions affected by changes to working memory, how can we avoid a slowdown in the matcher? One partial solution can already be found in the existing Rete algorithm. When two or more productions have the same first few conditions, the same parts of the Rete network are used to match those conditions. By *sharing* network nodes among productions in this way, Rete avoids the duplication of match effort across those productions.

In our systems, sharing becomes increasingly important as more rules are learned. Figure 4 shows the factor by which sharing reduces the number of tokens (partial instantiations) generated by the matcher. The y-axis is obtained by taking the number of tokens that would have been generated if sharing were disabled, and dividing it by the number that actually were generated with sharing. The figure displays this ratio as a function of the number of rules in each of the systems. (As mentioned before, we focus on the beta portion of the match — the figure shows the result of sharing in the beta memories and join nodes only, not the alpha part of the network. CPU time measurements would be preferable to token counts, but it would have taken months to run the systems with sharing disabled.) The figure shows that in the Dispatcher and Memory1 systems, sharing accounts for a tremendous reduction in the number of tokens; at the end of their runs, sharing reduces the number of tokens by two and three orders of magnitude, respectively. In the Assembler and Memory2 systems, sharing is not as effective, reducing the number of tokens by factors of 6 and 8, respectively. (In Figure 4, their points are very close to the horizontal axis.)

Why is sharing so important in Dispatcher and Memory1? As mentioned above, in each of these systems, many of the learned rules have their first few

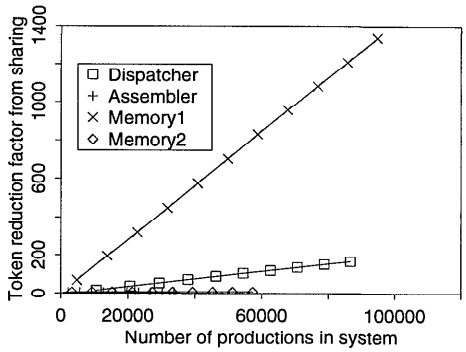

Figure 4: Reduction in tokens due to sharing.

conditions in common. Thus, newly learned rules often share existing parts of the Rete network. In particular, nodes near the top of the network tend to become shared by more and more productions, while nodes near the bottom of the network tend to be unshared (used by only one production) — recall that the beta portion of the network forms a tree. The sharing near the top is crucial in Dispatcher and Memory1 because nodes near the top are activated more often (on average) than nodes near the bottom. Thus, the token counts are dominated by the tokens generated near the top of the network, where sharing increases with the number of rules. In the Assembler and Memory2 systems, however, there is a significant amount of activity in lower parts of the network where nodes are unshared. (The difference is because the lower part of the network forms an effective discrimination tree in one case but not the other.) So sharing is not as effective in Assembler and Memory2. In Section 6, we give ideas for reducing the activity in lower parts of the network. If they prove effective, the activity near the top of the network would again dominate, and sharing would be increasingly important in the Assembler and Memory2 systems as well.

Interestingly, work in production systems has often ignored sharing. Previous measurements on smaller production systems have found sharing of beta nodes to produce only very limited speedup (Gupta, 1987; Miranker, 1990). This is probably due to the limited affect set size in those systems. An important example of the consequences of ignoring sharing can be found in the Treat algorithm. Treat does not incorporate sharing in the beta part of the match, so on each working memory change, it must iterate over all the affected rules. Thus, Treat requires time at least linear in the affect set size. So it would not scale well for the systems examined here — like Rete as shown in Figure 2, it too would slow down at least linearly in the number of rules.

Moreover, work on machine learning systems has also often failed to incorporate sharing into the match process. For example, the match algorithm used by Prodigy (Minton, 1988) treats each rule independently of all the others. As more and more rules are learned, the match cost increases, leading to the utility problem in Prodigy. Prodigy's approach to this problem is to discard many of the learned rules in order to avoid the match cost. The results above suggest another approach: incorporate sharing (and perhaps other optimizations) into the matcher so as to alleviate the increase in match cost.

5. Right Unlinking

While sharing is effective near the top of the Rete network, it is, of course, uncommon near the bottom, since new productions' conditions match those of existing productions only up to a certain point. (In these systems, it is rare for two productions to have *exactly* the same conditions.) Consequently, as more and more rules are learned, the amount of work done in lower parts of the network increases. This causes an overall slowdown in Rete, as shown in Figure 2. What can be done to alleviate this increase? The match cost (in the beta part of the network) can be divided into three components (see Figure 1): (1) activations of join nodes from their associated alpha memories (henceforth called *right activations*), (2) activations of join nodes from their associated beta memories (henceforth called *left activations*), and (3) activations of beta memory nodes. This section presents an optimization for the Rete algorithm that reduces (1), which turns out to account for almost all of the Rete slowdown in the Dispatcher and Memory1 systems. Some ways to reduce (2) and (3) are discussed in Section 6.

Our optimization is based on the following observation: on a right activation of a join node (due to the addition of a working memory element to its associated alpha memory), if its beta memory is empty then no work need be done. The new working memory element cannot match any items in the beta memory, because there aren't any items there. So if we know in advance that the beta memory is empty, we can skip the right activation of that join node. We refer to right activations of join nodes with empty beta memories as *null right activations*.

We incorporate *right unlinking* into the Rete algorithm as follows: add a counter to every beta memory to indicate how many items it contains; update this counter whenever an item is added to or deleted from the memory. If the counter goes from 1 to 0, *unlink each child join node from its associated alpha memory*. If the counter goes from 0 to 1, *relink each child join node to its associated alpha memory*. On each alpha memory there is a list of associated join nodes; the unlinking is done by splicing entries into and out of this list. So while a join node is unlinked from its alpha memory, it never gets activated by the alpha memory. Note that since the only activations we are skipping are null activations — which would not yield a match anyway — this optimization does not affect the set of complete production matches that will be found.

Figure 5 shows the results of adding right unlinking to Rete. Like Figure 2, it plots the average match time per decision cycle for each of the systems as a function of the number of rules. Note that the scale on the vertical axis is

different; all four systems run faster with right unlinking. Figure 6 shows the speedup factors obtained from right unlinking in the systems. (This is the ratio of Figure 2 and Figure 5.)

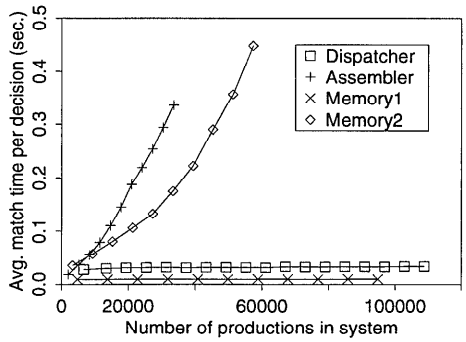

Figure 5: Match cost with right unlinking.

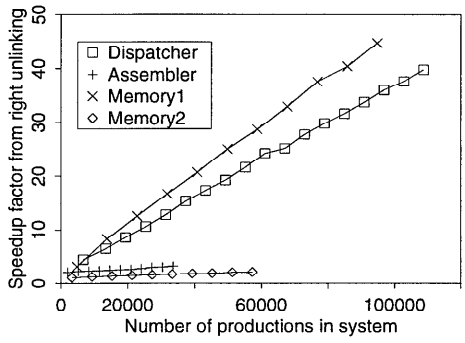

Figure 6: Speedup factors from right unlinking.

Why is this optimization effective? At first glance it seems like a triviality, since it merely avoids null right activations, and a null right activation takes only a handful of CPU cycles. What makes right unlinking so important is that *the number of null right activations per working memory change can grow linearly in the number of rules* — and as the number of rules becomes very large, this can become the dominant factor in the overall match cost in Rete. We first explain why the number of right activations can grow linearly, then explain why almost all of these are null right activations.

Recall from Section 3 that the average affect set size increases with the number of rules. For each affected production, there is some join node that gets right activated. Of course it may be that all the affected productions share this same join node, so that there is only one right activation. But as the beta part of the network forms a tree, sharing is only effective near the top; whenever a new working memory element matches the *later* conditions in many productions, many different join nodes are activated. As an extreme example, consider the case where the last condition is the same in every one of n productions. A single alpha memory will be used for this last condition; but if earlier conditions of the productions differ, the productions cannot share the same join node for the last condition. Thus, the alpha memory will have n associated join nodes, and a new match for this last condition will result in n right activations. Right unlinking is essentially a way to reduce this potentially large fan-out from alpha memories. Although reordering the conditions would avoid the problem in this particular example, in many cases no such reordering exists.

Now, as more and more rules are learned, the number of join nodes in the lower part of the network increases greatly, since sharing is not very common there. However, at any given time, almost all of them have empty beta memories. This is because usually there is some join node higher up that has no matches — some earlier condition in the rule fails. To see why, consider a rule with conditions $C_1, C_2, \ldots, C_k$, and suppose each C_i has probability p_i of having a match. For the very last join node to have a non-empty beta memory, all the earlier conditions must match; this happens with the relatively low probability $p_1 p_2 \cdots p_{k-1}$. Since most of the lower nodes have empty beta memories at a given time, most right activations of them are in fact null right activations.

In the Dispatcher and Memory1 systems, right unlinking yields a speedup factor of 40-50 when the number of rules is ~100,000. The factor increases with the number of rules. Comparing Figures 2 and 5, we see that right unlinking eliminates almost all of the slowdown in the Dispatcher and Memory1 systems. Thus, the slowdown in the standard Rete for those systems is almost entirely due to increasing null right activations. In the Assembler and Memory2 systems, however, there are additional sources of slowdown in the standard Rete: left and beta memory activations are also increasing. Right unlinking eliminates just one of the sources, so the speedup factors it yields are lower (2-3).

Note that the match cost measurements here are done directly in terms of CPU time. The token or comparison counts commonly used in studies of match algorithms would not reflect the increasing match cost in Rete or Treat, since a null right activation requires no comparisons and generates no new tokens. This prevented (Doorenbos, Tambe, & Newell, 1992) from noticing any increase in match cost in an earlier, shorter run of Dispatcher-Soar. The machine- and implementation-dependent nature of CPU time comparisons is avoided here by running the same implementation (modulo the right unlinking optimization) on the same machine.

6. Conclusions and Future Work

We have shown that the Rete and Treat match algorithms do not scale well with the number of rules in our systems, at least in part because the affect set size increases with the total number of rules. Thus, it is important to reduce the amount of work done by the

matcher when the affect set size is large. Both the right unlinking optimization and the sharing of beta nodes can be viewed as members of a family of optimizations that do this. Right unlinking avoids the work for right activations associated with many of the affected rules which cannot match. The sharing of beta nodes allows the matcher to do work once for large groups of affected rules, rather than do work repeatedly for each affected rule.

Another possible optimization in this family is *left unlinking*: whenever a join node's associated alpha memory is empty, the node is unlinked from its associated beta memory. This is just the opposite of right unlinking; while right unlinking reduces the number of right activations by reducing the fan-out from alpha memories, left unlinking would reduce the number of left activations by reducing the fan-out from beta memories. Unfortunately, left and right unlinking cannot simply be combined in the same system: if a node were ever unlinked from both its alpha and beta memories, it would be completely cut off from the rest of the network and would never be activated again, even when it should be.

Another optimization in this family is incorporated in Treat. Treat maintains a *rule-active* flag on each production, indicating whether all of its alpha memories are non-empty. If any alpha memory is empty, the rule cannot match, so Treat does not perform any of the joins for that rule. This essentially reduces the number of left activations and beta memory activations. But since Treat must at least check this flag for each affected rule, the number of right activations remains essentially the same. Right and left activations and beta memory activations together account for the entire beta phase of the match, so this suggests that the slowdown observed in the Assembler and Memory2 systems might be eliminated by some hybrid of Rete (with right unlinking) and Treat, or by adding to Treat an optimization analogous to right unlinking. Of course, much further work is needed to develop and evaluate such match optimizations.

7. Acknowledgements

Thanks to Anurag Acharya, Jill Fain Lehman, Dave McKeown, Paul Rosenbloom, Milind Tambe, and Manuela Veloso for many helpful discussions and comments on drafts of this paper, to the anonymous reviewers for helpful suggestions, and to Joe Mertz for providing the Assembler-Soar system.

8. References

Acharya, A., and Tambe, M. 1992. *Collection-oriented Match: Scaling Up the Data in Production Systems*. Technical Report CMU-CS-92-218, School of Computer Science, Carnegie Mellon University.

Cohen, W. W. 1990. Learning approximate control rules of high utility. *Proceedings of the Sixth International Conference on Machine Learning*, 268-276 .

Doorenbos, R., Tambe, M., and Newell, A. 1992. Learning 10,000 chunks: What's it like out there? *Proceedings of the Tenth National Conference on Artificial Intelligence*, 830-836 .

Etzioni, O. 1990. *A structural theory of search control.* Ph.D. diss., School of Computer Science, Carnegie Mellon University.

Forgy, C. L. 1981. *OPS5 User's Manual.* Technical Report CMU-CS-81-135, Computer Science Department, Carnegie Mellon University.

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

Gratch, J. and DeJong, G. 1992. COMPOSER: A probabilistic solution to the utility problem in speed-up learning. *Proceedings of the Tenth National Conference on Artificial Intelligence*, 235-240 .

Gupta, A. 1987. *Parallelism in Production Systems.* Los Altos, California: Morgan Kaufmann.

Laird, J.E., Newell, A., and Rosenbloom, P.S. 1987. Soar: An architecture for general intelligence. *Artificial Intelligence 33*(1), 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*(1), 11-46.

McDermott, J. 1983. Extracting knowledge from expert systems. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, 100-107 .

Mertz, J. 1992. Deliberate learning from instruction in Assembler-Soar. Proceedings of the Eleventh Soar Workshop.

Minton, S. 1988. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* Ph.D. diss., Computer Science Department, Carnegie Mellon University.

Miranker, D. P. 1990. *TREAT: A New and Efficient Match Algorithm for AI Production Systems.* San Mateo, California: Morgan Kaufmann.

Miranker, D. P., Brant, D. A., Lofaso, B., and Gadbois, D. 1990. On the performance of lazy matching in production systems. *Proceedings of the Eigth National Conference on Artificial Intelligence*, 685-692 .

Oflazer, K. 1984. Partitioning in parallel processing of production systems. *Proceedings of the IEEE International Conference on Parallel Processing*, 92-100 .

Rosenbloom, P.S. and Aasman J. 1990. Knowledge level and inductive uses of chunking (EBL). *Proceedings of the National Conference on Artificial Intelligence*, 821-827 .

Rosenbloom, P. S., Laird, J. E., Newell, A., and McCarl, R. 1991. A preliminary analysis of the Soar architecture as a basis for general intelligence. *Artificial Intelligence 47*(1-3), 289-325.

Sellis, T., Lin, C-C., and Raschid, L. 1988. Implementing large production systems in a DBMS environment: Concepts and algorithms. *Proceedings of the ACM-SIGMOD International Conference on the Management of Data*, 404-412 .

Tambe, M., Newell, A., and Rosenbloom, P. S. 1990. The problem of expensive chunks and its solution by restricting expressiveness. *Machine Learning 5*(3), 299-348.