Combining Left and Right Unlinking for Matching a Large Number of Learned Rules

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

Abstract

In systems which learn a large number of rules (productions), it is important to match the rules efficiently, in order to avoid the machine learning *utility problem*. So we need match algorithms that scale well with the number of productions in the system. (Doorenbos 1993) introduced *right unlinking* as a way to improve the scalability of the Rete match algorithm. This paper introduces a symmetric optimization, *left unlinking*, and demonstrates that it makes Rete scale well on an even larger class of systems. Unfortunately, when left and right unlinking are combined in the same system, they can interfere with each other. We give a particular way to combine them which we prove minimizes this interference, and analyze the worst-case remaining interference. Finally, we present empirical results showing that the interference is very small in practice, and that the combination of left and right unlinking allows five of our seven testbed systems to learn over 100,000 rules without incurring a significant increase in match cost.[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. This is important because if AI is to achieve its long-term goals, AI systems (including rule-based systems) must be able to use vast amounts of knowledge. 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 if the slowdown in the matcher increases the time per step, then this can outweigh the reduction in the number of steps. This has been observed in several machine learning systems (Minton 1988; Etzioni 1990a; Tambe, Newell, & Rosenbloom 1990; Cohen 1990; Gratch & DeJong 1992).

To avoid this slowdown, previous research on the utility problem from match cost has taken three general approaches. One approach is simply to reduce the number of rules in the system's knowledge base, by being selective about when to learn or which rules or types of rules to learn, or by forgetting previously learned rules if they slow down the matcher enough to cause an overall system slowdown (Minton 1988; Etzioni 1990b; Holder 1992; Gratch & DeJong 1992; Greiner & Jurisica 1992; Markovitch & Scott 1993). Unfortunately, this approach is inadequate for the long-term goals of AI because, given the current state of match technology, it precludes learning a vast amount of knowledge. Moreover, it is intuitively desirable to have AI systems that take advantage of every opportunity for learning, rather than forgoing certain opportunities.

The second general approach is to reduce the match cost of individual rules, taken one at a time. Many techniques have been developed for this (Tambe, Newell, & Rosenbloom 1990; Minton 1988; Etzioni 1990a; Pérez & Etzioni 1992; Chase *et al.* 1989; Cohen 1990; Kim & Rosenbloom 1993). This prevents just a handful of *expensive rules* from slowing the matcher down to a crawl; thus, the system can learn more rules before an overall slowdown results. Unfortunately, an overall slowdown can still result when a large number of individually cheap rules exact a high match cost.

The third approach, and the one taken by this work, complements the second by reducing the aggregate match cost of a large number of rules, without regard to

[1]This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. Views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of Wright Laboratory or the United States Government.

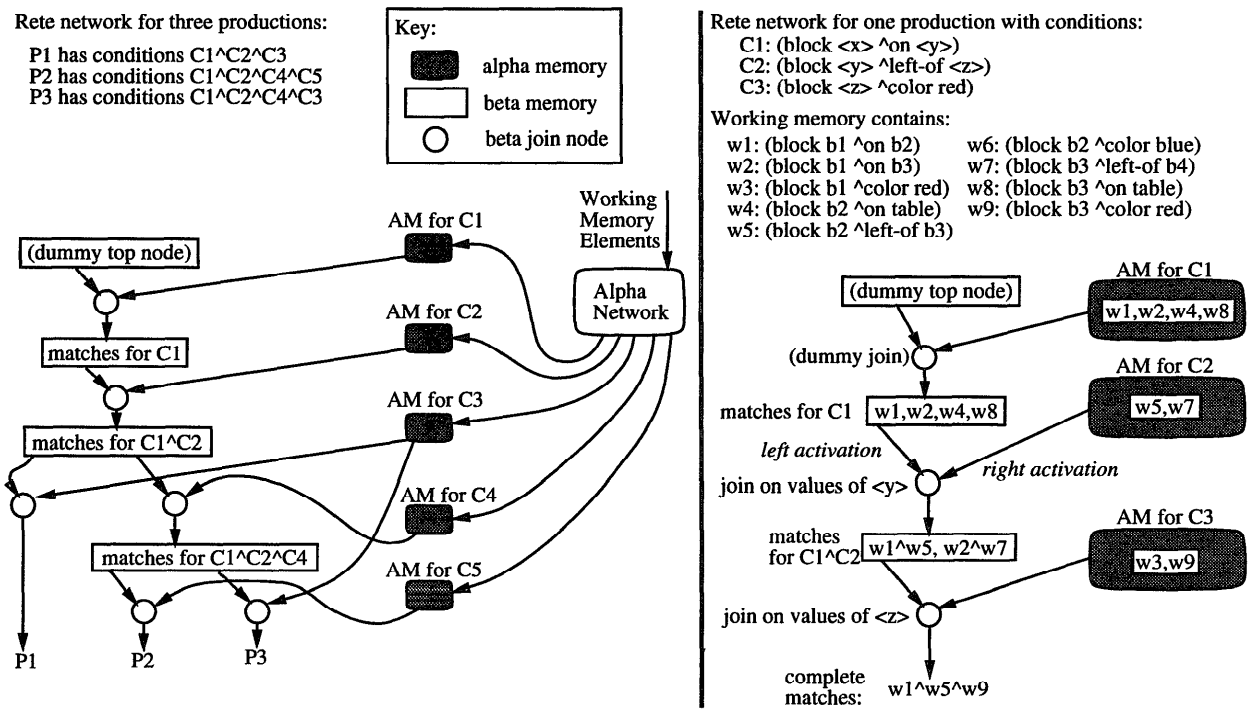

Figure 1: Network used by Rete for several productions (left) and instantiated network for one production (right).

the cost of each individual rule. As (Doorenbos 1993) showed, the use of sophisticated match algorithms can sometimes reduce the matcher slowdown due to learning a large number of rules to the point where it is unproblematic. (Doorenbos 1993) introduced the idea of *right unlinking* in the Rete match algorithm (Forgy 1982), and showed that for at least one "natural" system (not designed just for match algorithm performance), Dispatcher, right unlinking eliminated virtually all the matcher slowdown associated with learning 100,000 rules. However, for another natural system, Assembler, the matcher was still slowing down to a crawl as the system learned 35,000 rules.

In this paper we build on the idea of right unlinking. We begin by reviewing in Section 2 the basic Rete algorithm and right unlinking. Section 3 examines the cause of the slowdown in the Assembler system, and shows how it can be avoided by adding another improvement to Rete: *left unlinking*. Left unlinking is essentially symmetric to right unlinking. Unfortunately, these optimizations are not completely orthogonal: when combined in the same system, they can interfere with each other. In Section 4 we give a particular way to combine them which we prove minimizes this interference. The worst-case remaining interference is analyzed in Section 5. Finally, Section 6 presents empirical results, showing that in contrast to the worst case, the interference is very small in practice. The evaluation is done with respect to a set of seven systems — including the aforementioned Dispatcher and Assembler — implemented in Soar (Laird, Newell, & Rosenbloom 1987; Rosenbloom *et al.* 1991), an architecture which learns new rules through *chunking* (Laird, Rosenbloom, & Newell 1986). The combination of both left and right unlinking allows five out of the seven systems to learn over 100,000 rules without incurring a significant increase in match cost.

2 Background

We begin by briefly reviewing 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 alpha network is commonly implemented using a simple discrimination network and/or hash tables, and thus is very efficient, running in approximately constant time per change to working memory. Previous studies have shown that the beta part of the network accounts for most of the match cost (Gupta 1987). The beta part 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.

An important feature of Rete is its *sharing* of nodes between productions. When two or more productions have a common condition, Rete uses a single alpha memory for it, rather than creating a duplicate memory for each production. Furthermore, when two or more productions have the same first few conditions, the same parts of the network are used to match those conditions, avoiding duplicating match effort across those productions.

Although Rete is one of the best standard match algorithms available, (Doorenbos 1993) observed that neither it nor its major alternative — Treat (Miranker 1990) — scales well with the number of rules in the Dispatcher and Assembler systems: both Rete and Treat slow down linearly in the number of rules. However, (Doorenbos 1993) described an optimization for Rete, *right unlinking*, which eliminated this linear slowdown in Dispatcher (but not Assembler).

To understand what right unlinking is, consider the activation of a join node from its associated alpha memory (henceforth called a *right activation*) — this happens whenever a working memory element is added to or removed from its alpha memory. Right unlinking is based on the observation that if the join node's beta memory is empty, then no work need be done: the 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 modify the Rete algorithm to incorporate *right unlinking* as follows. On each alpha memory there is a list of associated join nodes. Whenever a join node's beta memory becomes empty, we splice that join node out of the list on its associated alpha memory. When the beta memory later becomes nonempty again, we splice the join node back into the list. Now 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.

As (Doorenbos 1993) showed, in the Dispatcher system, the slowdown in the standard Rete algorithm is almost entirely due to a linear increase in the number of null right activations. So right unlinking is very effective in avoiding the slowdown in Dispatcher. However, in the Assembler system, it yields only a constant speedup factor of about three, because null right activations are not the only major source of slowdown there.

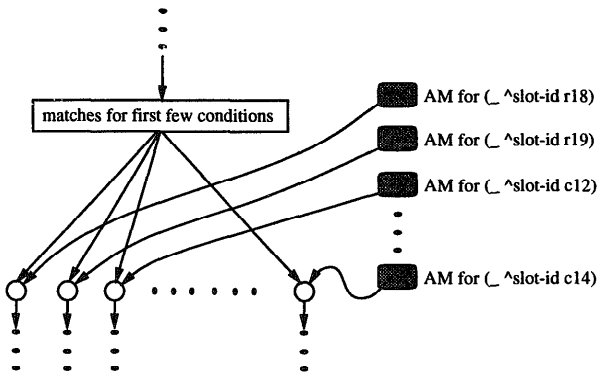

Figure 2: Part of the Rete network for Assembler.

3 Left Unlinking

For the Assembler system, there appears to be a second major source of slowdown: a significant linear increase in the number of null left activations. This system is a cognitive model of a person assembling printed circuit boards (e.g., inserting resistors into the appropriate slots). Most of the rules it learns are specific to the particular slot on the board being dealt with at the moment. The first few conditions in these rules are always the same, but the next condition is different in each rule. As illustrated in Figure 2, this leads to a large fan-out from one beta memory. The first few conditions in all the rules share the same nodes, but at this point, sharing is no longer possible because each rule tests for a *different* slot. As the system deals with more and more slots, more and more rules are learned, and the fan-out increases linearly in the total number of rules.

Now, whenever all of the first few conditions of these rules are true, the dataflow in the Rete network reaches this beta memory and a token is stored there. This token is then propagated to all the memory's child join nodes, *left activating* those nodes. Since the number of such nodes is increasing linearly in the number of rules, the work done here is also linearly increasing. However, most of this work is wasted. Since the system is only focusing its attention on a few slots at a time, most of the join nodes have empty alpha memories. Their activations are therefore *null left activations*, and no new matches result from them. Although each individual null left activation takes very little time to execute, the number of null left activations (per change to working memory) is linearly increasing, and so this can grow to dominate the overall match cost.

To avoid doing all this fruitless work, we can incorporate *left unlinking* into the Rete algorithm. Left un-

linking is symmetric to right unlinking: whereas with right unlinking, a join node is spliced out of its alpha memory's list of successors whenever its beta memory is empty, with left unlinking, a join node is spliced out of its beta memory's list of successors whenever its alpha memory is empty. Thus, in Figure 2, most of the join nodes would be unlinked from the beta memory, so they would *not* be activated whenever the first few conditions in the rules are true. As with right unlinking, the only activations we are skipping are null activations — which would not yield any matches anyway — so this optimization leaves the correctness of the Rete algorithm intact.[2]

Left unlinking is expected to be useful in many systems in addition to Assembler. The large fan-out from beta memories could arise in any system where the domain has some feature with a large number of possible values, and the learned rules are specific to particular values of that feature. For instance, in a robot domain, if the appropriate action to be taken by the robot depends on the exact current room temperature, it might learn a set of rules where each one checks for a different current temperature reading. In a system with a simple episodic memory, learned rules implementing that memory might all contain different timestamps in their conditions. In cases like these, learned rules will often share nodes in the Rete network for their *early* conditions, up to but not including the conditions testing the feature in question. If this feature can have only one value at a time, then most of the rules will fail to match at this condition, and left unlinking will avoid a large number of null left activations. In addition, we will see in Section 6 that left unlinking can often be beneficial even in systems where the fan-out isn't especially large and null left activations don't dominate the overall match cost.

4 Combining Left & Right Unlinking

Since right unlinking avoids all null right activations, and left unlinking avoids all null left activations, we would like to combine both in the same system and avoid all null activations entirely. Unfortunately, this is not possible, because the two optimizations can interfere with each other. The problem arises when a join node's alpha and beta memories are *both* empty. Left unlinking dictates that the node be unlinked from its beta memory. Right unlinking dictates that the node be unlinked from its alpha memory. If we do both, then the join node will be completely cut off from the rest of the network and will never be activated again, even when it should be. The correctness of the match algorithm would be lost. *To ensure correctness, when a join node's memories are both empty, we can use either left unlinking or right unlinking, but not both.* But which one? If we use left but not right unlinking in this situation, then we can still suffer null right activations. If we use right but not left unlinking, then we can still suffer null left activations in this situation. Thus, no scheme for combining left and right unlinking can avoid *all* null activations.

If both memories are empty, which one should the join node be unlinked from? A number of possible heuristics come to mind. We might left unlink nodes whose beta memories have sufficiently large fan-out (as in Figure 2). Or we might do a trial run of the system in which we record how many null left and right activations each node incurs; then on later runs, we would unlink from the side that incurred more null activations in the trial run.

Remarkably, it turns out that there is a simple scheme for combining left and right unlinking which is not only straightforward to implement, but also provably optimal in minimizing the residual number of null activations.

Definition: *In the* **first-empty-dominates** *scheme for combining left and right unlinking, a join node J with alpha memory A and beta memory B is unlinked as follows. (1) If A is empty but B is nonempty, it is linked to A and unlinked from B. (2) If B is empty but A is nonempty, it is linked to B and unlinked from A. (3) If A and B are both empty, it is (i.e., remains) linked to whichever memory became empty earlier, and unlinked from the other memory.*

To see how this works and how it falls naturally out of a straightforward implementation, consider a join node that starts with its alpha and beta memories both nonempty (so it is linked to both). Now suppose the alpha memory becomes empty. We unlink the join node from its beta memory (i.e., left unlink it). As long as the alpha memory remains empty, the join node remains unlinked from the beta memory — and hence, never gets activated from the beta memory: it never hears about any changes to the beta memory. Even if the beta memory becomes empty, the join node doesn't get informed of this, so nothing changes — it remains left unlinked — and the empty alpha memory essentially "dominates" the empty beta memory because the alpha memory became empty first. The join node remains unlinked from its beta memory until the alpha memory becomes nonempty again.

The definition of first-empty-dominates ignores the possibility that a join node could *start* with both its memories empty. When a rule is learned and added to the Rete net, some of its join nodes may have both memories empty. In this case, we can pick one side by any convenient method. (In the current implementation, the node is right unlinked.) The worst that can happen is that we pay a one-time initialization cost of one null activation for each join node; this cost is negligible in the long run. Once one of the memories

[2] We ignore here the case of negative conditions, which test for the *absence* of particular items in working memory. Nodes for negative conditions cannot be left unlinked without destroying the correctness of the algorithm. Fortunately, they are typically much less common than positive conditions.(Gupta 1987)

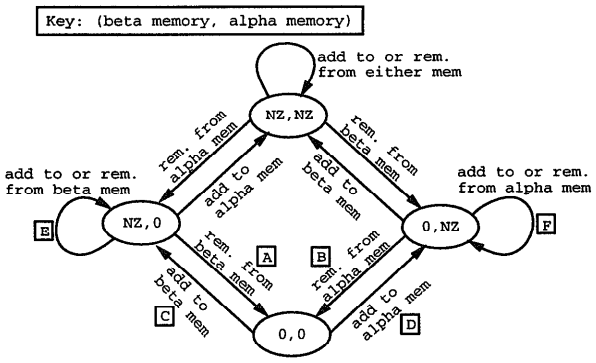

Figure 3: Possible states of a join node and its alpha and beta memories.

becomes nonempty, we can use first-empty-dominates.

It turns out that first-empty-dominates is the optimal scheme for combining left and right unlinking: except for the possible one-activation initialization cost, it minimizes the number of null activations. Thus, this simple scheme yields the minimal interference between left and right unlinking. This result is formalized as follows:

Theorem (Optimality of First-Empty-Dominates): *Any scheme for combining left and right unlinking must incur at least as many null activations of each join node as first-empty-dominates incurs, ignoring the possible one-activation initialization cost.*

Proof: For any given join node, Figure 3 shows the four states its alpha and beta memories can be in: the number of items in each memory can be 0 or nonzero (NZ). The figure also shows all the possible state transitions that can occur on changes to the alpha and beta memories. All the transitions into and out of (NZ,NZ) are *non-null* activations. Unlinking never avoids non-null activations, so the join node will incur one activation on each of these transitions no matter what unlinking scheme we use.

The remaining transitions (labeled A–F) are null activations if no unlinking is done; but the join node will not be activated on these if it is unlinked from the appropriate memory. Under first-empty-dominates, the join node is always unlinked from its beta memory when in state (NZ,0). This means it will not incur a null activation on transition A or E. Similarly, it is always unlinked from its alpha memory when in state (0,NZ), so it will not incur a null activation on transition B or F. This leaves just C and D to consider. In state (0,0), the join node is unlinked from its beta memory if its alpha memory became empty *before* its beta memory did — i.e., if it arrived at (0,0) via transition A — and unlinked from its alpha memory otherwise — i.e., if it arrived via B. (This ignores the case where the node *starts* at (0,0).) This means a null activation is incurred by first-empty-dominates only when D follows A or when C follows B.

Now, in *any* scheme for combining left and right unlinking, the join node must incur at least one null activation when taking transition A and then D — the reason is as follows. The join node cannot start out unlinked from both sides: as noted above, this would destroy the correctness of the algorithm. If it starts out linked to its beta memory, it incurs a null activation on transition A. On the other hand, if it starts out linked to its alpha memory, it incurs a null activation on transition D. (The link cannot be "switched" after transition A but before D — that would require executing a piece of code just for this one join node, which logically constitutes an activation of the node.) So in any case, it incurs at least one null activation.

A symmetric argument shows that in *any* unlinking scheme, at least one null activation must be incurred when taking transition B and then C. Since these are the only causes of null activations in first-empty-dominates, and it incurs only a single null activation on each one, it follows that *any* scheme must incur at least as many null activations. Q.E.D.

5 Worst-Case Analysis

How bad could the interference between left and right unlinking be? It would be nice if the residual number of null activations per change to working memory were bounded, but unfortunately it is not — other than being bounded by n, the total number of join nodes in the network (a very large bound). However, null activations don't occur all by themselves — they are *triggered* by changes to alpha and beta memories. If it takes a lot of triggering activity to cause a lot of null activations, then null activations won't dominate the overall match cost. The question to ask is: "By what factor is the matcher slowed down by null activations?" To answer this, we'll look at

$$\nu \stackrel{\text{def}}{=} \frac{\#\text{ of activations of all nodes}}{\#\text{ of act'ns of all nodes, } \textit{except} \text{ null join act'ns}} = 1 + \frac{\#\text{ of null join node act'ns}}{\#\text{ act'ns of all nodes, } \textit{except} \text{ null join act'ns}}.$$

Note that ν is actually a pessimistic answer to this question, because it assumes all activations have the same cost, when in fact null activations take less time to execute than other activations.

Without any unlinking, or with left or right unlinking but not both, it is easy to show that ν is $O(n)$ in the worst case, and in fact this often arises in practice, as shown in (Doorenbos 1993) and in Section 6 below. However, the first-empty-dominates combination of left and right unlinking reduces this theoretical worst case to $O(\sqrt{n})$.

Before we get to the worst-case analysis, two assumptions must be made. First, we will ignore the possible initial null activation of each join node which starts out with its alpha and beta memories both empty — of course, this initialization cost is negligible in the long run. Second, we will assume that no

two join nodes use both the same alpha memory *and* the same beta memory. This is the normal situation in practice.[3]

Given these assumptions, it turns out that with the first-empty-dominates combination of left and right unlinking, $\nu \leq 1 + \frac{1}{2}\sqrt{n}$. To see how the worst case can arise, consider a system with k alpha memories, each initially with one item in it; k beta memories, each initially empty; and $n = k^2$ join nodes, one for each pair of alpha and beta memories. Now suppose that in each alpha memory, the one item in it is removed; following this, one item is first added to and then removed from every beta memory; and finally, a new item is added to each alpha memory. This sequence of $4k$ activations of alpha and beta memories causes every join node to undergo transitions B, C, A, and D (see Figure 3), incurring two null activations, for a total of $2k^2$ null activations. Hence $\nu = 1 + \frac{2k^2}{4k} = 1 + \frac{k}{2} = 1 + \frac{1}{2}\sqrt{n}$. That this value of ν is actually the worst one possible is proved in the extended version of this paper (Doorenbos 1994).

6 Empirical Results

In Section 3, we introduced left unlinking as a way to avoid the matcher slowdown associated with increasing fan-outs from beta memories in systems like Assembler. After presenting in Section 4 the optimal way to combine left and right unlinking so as to minimize the interference between them, we analyzed in Section 5 how much interference there can be in the worst case. This section examines what happens in practice — how much interference is there in practice, and how well does combined left and right unlinking avoid matcher slowdown?

To answer these questions, we ran experiments using a set of seven Soar systems, including the aforementioned two. Assembler (Mertz 1992) is a cognitive model of a person assembling printed circuit boards. Dispatcher (Doorenbos, Tambe, & Newell 1992) is a message dispatcher for a large organization and uses an external organizational database. Merle (Prietula *et al.* 1993) schedules tasks for an automobile windshield factory. Radar (Papageorgiou & Carley 1993) learns to classify aircraft radar images (specified as simple feature vectors) as friendly, neutral, or hostile. SCA (Miller & Laird 1991) is a Soar system that performs traditional concept acquisition. Two versions of SCA were used: SCA-Fixed always focuses its attention on the same features of each training example, whereas SCA-Rand focuses on a different randomly chosen set of features on each training example. This leads to much better sharing in the Rete network for SCA-Fixed than SCA-Rand, and consequent matcher performance differences. Finally, Sched (Nerb, Krems, & Ritter 1993) is a computational model of skill acquisition in job-shop scheduling. These seven systems provide a good test suite because they use a variety of problem-solving methods in a variety of domains, and none was designed for these experiments. For each system, a problem generator was used to create a set of problem instances; the system was then allowed to solve the sequence of problems, learning new rules as it went along. Each system learned at least 100,000 rules.

Table 1 shows, for each system, the number of null and non-null join node activations per working memory change, averaged over the course of the whole run. For null activations, four different numbers are given, corresponding to four different match algorithms: the basic Rete algorithm without any unlinking, Rete with left but not right unlinking, Rete with right but not left unlinking, and Rete with the first-empty-dominates combination of left and right unlinking. The table shows that without any unlinking, or with left unlinking only, the matcher is essentially swamped by null activations in all the systems. With right unlinking but no left unlinking, there are still a large number of null (left) activations in both Assembler and Radar, and a fair number in Sched. Finally, with left and right unlinking combined, the number of null activations is very small in all the systems. Thus, the interference between left and right unlinking turns out to be insignificant in practice, at least for this diverse set of systems.

Returning to our second question — How well does combined left and right unlinking avoid matcher slowdown? Figure 4 shows the match cost, in CPU time per change to working memory, for each of the systems.[4] The match cost is plotted as a function of the total number of rules in the system. The four lines on each graph correspond to the four match algorithms described above. (Note that the vertical axes on the graphs have different scales — the match cost varies across systems.) The figure shows that without right unlinking, all the systems suffer a major linear slowdown as the system learns more and more rules. The addition of left unlinking (in combination with right unlinking) enables both Assembler and Radar to avoid a significant linear slowdown as the number of rules increases, and Sched to avoid a slight linear slowdown. This is because these three systems have increasing fan-outs from beta memories, as discussed in Section 3.

[3] It is possible for two or more join nodes to use the same alpha and the same beta memories — if there are productions whose conditions test exactly the same constants in the same places, but have different inter-condition variable binding consistency checks — but this is not very common in practice. Even in theory, the number of join nodes using the same pair of memories can be bounded *independent* of the number of productions in the system. If this bound is c, the worst-case bound becomes $1 + \frac{c}{2}\sqrt{n}$.

[4] Times are for Soar version 6.0.6 (modulo changes to the matcher) on a DFCstation 5000/200. Many of the runs without unlinking became so slow that time limitations forced them to be stopped at much less than 100,000 rules.

System	# of rules	Join node activations per change to working memory:				
			Null, when using this type of unlinking:			
		Non-null	None	Left only	Right only	Both
Assembler	105,308	15.4	2214.8	1711.5	503.3	0.11
Dispatcher	115,962	19.6	1248.4	1234.3	14.0	0.16
Merle	102,048	22.8	7561.7	7548.3	13.4	0.28
Radar	105,385	9.8	1570.7	1482.8	87.9	0.12
SCA-Fixed	108,799	7.1	2302.9	2301.7	1.2	0.20
SCA-Rand	106,853	13.6	2338.2	2333.4	4.8	2.29
Sched	117,386	21.7	4020.1	3976.0	44.1	0.22

Table 1: Average number of join node activations per change to working memory on each system, with different versions of the matcher.

Finally, the addition of left unlinking to right unlinking reduces the match cost slightly (7–15%) in most of the other systems.[5] Thus, not only is the addition of left unlinking to right unlinking crucial in systems where fan-outs from beta memories grow large, but it can be helpful in other systems, too.

7 Conclusions and Future Work

Although right unlinking avoids matcher slowdown due to increasing null right activations, it is insufficient for many systems which learn rules specific to one value of a domain feature with many possible values. In such systems, we can avoid the increase in null left activations by using left unlinking. Left and right unlinking have the potential to interfere with each other — significantly in the worst case — but the first-empty-dominates combination minimizes this interference and makes it very small in practice.

This paper dealt with networks containing just *binary* joins — every join node has exactly two input memories. Unlinking can be generalized to k-ary joins: if any one of the k input memories is empty, the node can be unlinked from each of the $k-1$ others (Barrett 1993). Finding extensions of the optimality theorem of Section 4 and the worst-case analysis of Section 5 for k-ary joins remains a topic for future work.

Although the remaining number of null activations is insignificant, this still leaves two other potential causes of matcher slowdown: increasing non-null activations (this appears to be the main source of the remaining ~2.8-fold slowdown in SCA-Rand), and increasing time per activation (this appears to be the main source of the remaining ~2.5-fold slowdown in Merle). The magnitude of the slowdown from these effects is much smaller than that from null activations in the unmodified Rete algorithm, but it is still significant. Determining under what circumstances these effects occur and finding ways to avoid them are therefore important areas for future work.

[5] Adding left unlinking can also slow the matcher down by a small factor, since it adds some overhead to each node activation. If this small factor is not outweighed by a large factor reduction in the number of activations, a slowdown can result. This happens in some of the systems when left unlinking is used *alone*: when (null) right activations are numerous, avoiding null left activations only reduces the total number of activations by a very small factor. This also happens in SCA-Fixed, where adding left unlinking to right unlinking increases the match cost 4%.

8 Acknowledgements

Thanks to Jill Fain Lehman and Paul Rosenbloom for many helpful discussions and comments on drafts of this paper, to the anonymous reviewers and Bill Kennedy for helpful suggestions, and to Josef Krems, Joe Mertz, Craig Miller, Josef Nerb, Constantine Papageorgiou, and David Steier for providing testbed systems.

References

Barrett, T. 1993. Private communication.

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

Cohen, W. W. 1990. Learning approximate control rules of high utility. In *Proceedings of the Seventh 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? In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 830–836.

Doorenbos, R. B. 1993. Matching 100,000 learned rules. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 290–296.

Doorenbos, R. B. 1994. Combining left and right unlinking for matching a large number of learned rules (extended version). Technical Report CMU-CS-94-132, School of Computer Science, Carnegie Mellon University.

Etzioni, O. 1990a. *A Structural Theory of Search Control.* Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University.

Etzioni, O. 1990b. Why PRODIGY/EBL works. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, 916–922.

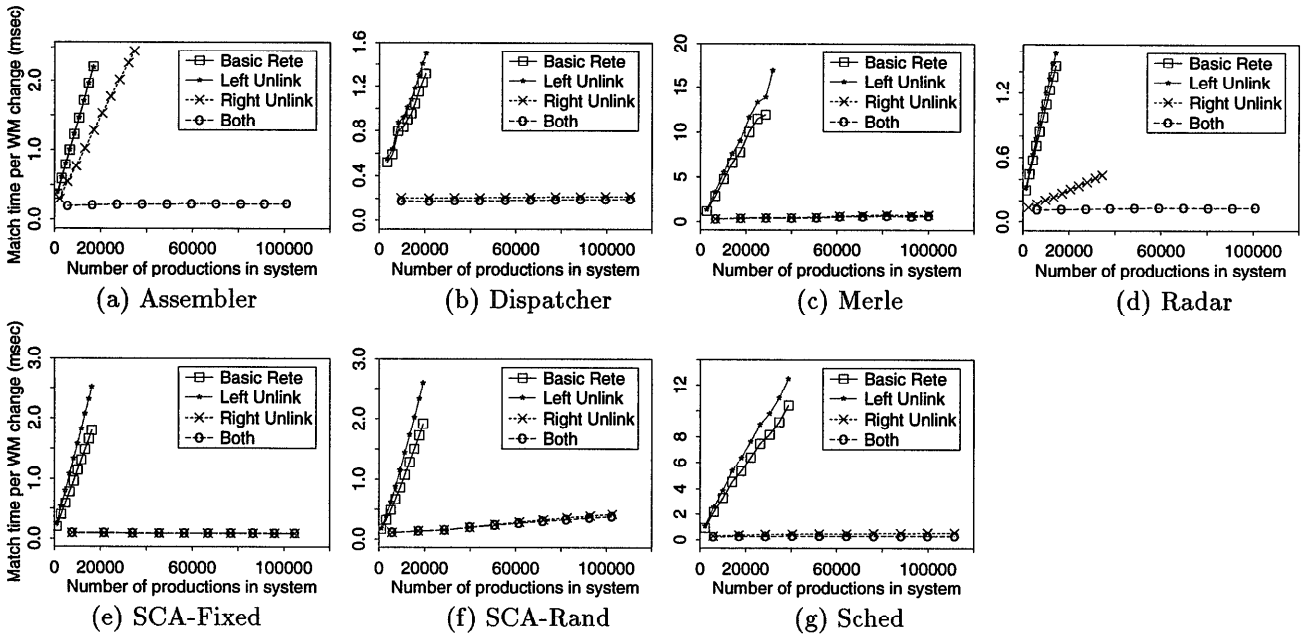

Figure 4: Performance of the matcher on each of the systems.

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. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 235–240.

Greiner, R., and Jurisica, I. 1992. A statistical approach to solving the EBL utility problem. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 241–248.

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

Holder, L. B. 1992. Empirical analysis of the general utility problem in machine learning. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 249–254.

Kim, J., and Rosenbloom, P. S. 1993. Constraining learning with search control. In *Proceedings of the Tenth International Conference on Machine Learning*, 174–181.

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.

Markovitch, S., and Scott, P. D. 1993. Information filtering: Selection mechanisms in learning systems. *Machine Learning* 10(2):113–151.

Mertz, J. 1992. Deliberate learning from instruction in Assembler-Soar. In *Proceedings of the Eleventh Soar Workshop*, 88–90. School of Computer Science, Carnegie Mellon University.

Miller, C. S., and Laird, J. E. 1991. A constraint-motivated model of concept formation. In *Proceedings of the Thirteenth Annual Conference of the Cognitive Science Society.*

Minton, S. 1988. *Learning Effective Search Control Knowledge: An Explanation-Based Approach.* Boston, MA: Kluwer Academic Publishers.

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

Nerb, J.; Krems, J. F.; and Ritter, F. E. 1993. Rule learning and the power law: a computational model and empirical results. In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society.*

Papageorgiou, C. P., and Carley, K. 1993. A cognitive model of decision making: Chunking and the radar task. Technical Report CMU-CS-93-228, School of Computer Science, Carnegie Mellon University.

Pérez, M. A., and Etzioni, O. 1992. DYNAMIC: A new role for training problems in EBL. In *Proceedings of the Ninth International Conference on Machine Learning*, 367–372.

Prietula, M. J.; Hsu, W.-L.; Steier, D.; and Newell, A. 1993. Applying an architecture for general intelligence to reduce scheduling effort. *ORSA Journal on Computing* 5(3):304–320.

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.

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.