Comparison of the Rete and Treat Production Matchers for Soar (A Summary)*

Pandurang Nayak
Knowledge Systems Lab.
Stanford Univ.
Stanford, CA 94305

Anoop Gupta
Computer Systems Lab.
Stanford Univ.
Stanford, CA 94305

Paul Rosenbloom
Information Sciences Inst.
Univ. of Southern California
Marina del Rey, CA 90292

Abstract

RETE and TREAT are two well known algorithms used for performing match in production systems (rule-based systems). In this paper, we compare the performance of these two algorithms in the context of Soar programs. Using the number of tokens processed by each algorithm as the performance metric, we show that the RETE algorithm performs better than the TREAT algorithm in most cases. Our results are different than the ones shown by Miranker for OPS5. The main reasons for this difference are related to the following: (i) fraction of times no joins need to be done; (ii) the long chain effect; (iii) matching of static structures; and (iv) handling of combinatorial joins. These reasons go beyond Soar in their applicability, and are relevant to other OPS5-based production systems that share some of Soar's properties. We also discuss several implementation issues for the two algorithms.

1 Introduction

Soar is a cognitive architecture that provides the foundations for building systems that exhibit general intelligent behavior [Laird *et al.*, 1987]. Soar uses an OPS5-like production system [Brownston *et al.*, 1985] to encode its knowledge base and it provides a vision of how future expert systems may be constructed. It has been exercised on many different tasks, including some of the classic AI toy tasks such as the Eight Puzzle, and the Missionaries and Cannibals problem, as well as on large tasks such as the R1 computer configuration task [Rosenbloom *et al.*, 1985], the Neomycin medical diagnosis task [Washington and Rosenbloom], and the Cypress algorithm design task [Steier, 1987]. It exhibits a wide range of problem-solving mechanisms and has a general mechanism for learning.

RETE and TREAT are prominent algorithms that have been designed to perform match in production systems. The RETE algorithm [Forgy, 1982] was proposed by Forgy and is currently used in almost all implementations of OPS5-like production systems. The TREAT algorithm [Miranker, 1984] has been proposed more recently by Miranker. In his recent study [Miranker, 1987], Miranker presents empirical evidence, based on five OPS5 programs, that seem to show that the TREAT match algorithm can outperform the RETE match algorithm, often by more than fifty percent.

This paper describes our experiments with the TREAT match algorithm for Soar, and the results of comparing its performance with the currently used RETE algorithm [Scales, 1986]. Our experiments show very different results than Miranker's. They show that for Soar, RETE outperforms TREAT in most cases. The main reasons for this difference are related to the following: (i) fraction of times no joins need to be done; (ii) the long chain effect; (iii) matching of static structures; and (iv) handling of combinatorial joins. These reasons go beyond Soar in their applicability, and are relevant to other OPS5-based production systems that share some of Soar's properties. We also discuss several implementation issues for the two algorithms.

The paper is organized as follows. The next section presents background material on Soar, RETE, and TREAT. Section 3 describes the condition ordering algorithms used in the various match algorithms. Section 4 presents the results and the discussion. The conclusions are in section 5.

2 Background

2.1 Soar

Soar is an architecture for a system that is to be capable of general intelligence. Soar is based on the *Problem Space Hypothesis*, which states that all symbolic goal-oriented behavior can be cast as a search in a problem space. A problem space consists of a set of states and a set of operators to move amongst these states. Every goal in Soar is formulated as a search in some appropriate problem space. Goals may have subgoals, leading to a hierarchy of goals.

Long-term knowledge in Soar is stored in OPS5-like productions. The current status of the problem-solving is stored in the *working memory*. The working memory consists of the *context stack*, the *augmentations*, and the *preferences*. The context stack contains the hierarchy of goals, with each goal having slots for the associated problem-space, state, and operator. Problem solving is driven by selecting objects for the slots in the context. Augmentations specify values for an attribute of an object. Preferences encode statements about selection of objects for the slots in the context stack.

To better understand the match algorithms, we now describe the working memory elements and the productions in some detail [Laird, 1986]. Augmentations of objects have four fields: (i) the *class* of the object, (ii) its (unique) *identifier*, (iii) the name of some *attribute*, and (iv) a *value* for that attribute. Preferences have nine fields, as against only four for augmentations. The meanings of these fields are not very relevant to the paper and will not be discussed here.

A production in Soar is a condition-action rule. The condition part of a production is made up of *condition elements*. Each field of a condition element specifies tests on the corresponding fields of a working memory element. A working memory element *matches* a condition element if it satisfies all

*This research was sponsored by the Hughes Aircraft Company, by Digital Equipment Corporation, and by the Defense Advanced Research Projects Agency (DOD) under contracts N00039-86-C-0133 and MDA903-83-C-0335. Computer facilities were partially provided by NIH grant RR-00785 to Sumex-Aim. 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 Hughes Aircraft Company, Digital Equipment Corporation, the Defense Advanced Research Projects Agency, the US Government, or the National Institutes of Health.

the tests specified in that condition element. For example, the condition element

(goal ↑id <g> ↑attr state ↑value <s>)

specifies that the class of a matching working memory element should be **goal**, and its attribute field should be **state**. The variables <g> and <s> are bound to the identifier and value fields of the working memory element, respectively.

We now define the notion of a *production instantiation*. A production instantiation is a list of working memory elements such that each condition element in the production is matched by a working memory element, with the added restriction that the variable bindings induced by the working memory elements must be consistent.[1] For example, consider the following production, named **joe-production**, which consists of of three condition elements and one action:[2]

```
(p joe-production
   (goal  ↑id <g> ↑attr state ↑value <s>)
   (state ↑id <s> ↑attr hole  ↑value <x>)
   (state ↑id <s> ↑attr peg   ↑value <x>)
   -->
   (make state ↑id <s> ↑attr fits? ↑value yes))
```

The list of working memory elements

```
((goal  ↑id g1 ↑attr state ↑value s2)
 (state ↑id s2 ↑attr hole  ↑value square)
 (state ↑id s2 ↑attr peg   ↑value square))
```

is an instantiation of the production since each condition element has a matching working memory element and the variables have consistent bindings (<g> is consistently bound to **g1**, <s> to **s2**, and <x> to **square**).

A production may have more than one instantiation, corresponding to the different sets of working memory elements matching its condition elements in a consistent way. Each instantiation of a production causes it to *fire*. Firing a production instantiation is the process of executing its action part with variables being replaced by the bindings induced by the instantiation. Usually, the action of Soar productions is to add new working memory elements, though other kinds of actions are available for tracing and interfacing.

2.2 Database formalism for the matcher

The matcher of a production system must keep track of the production instantiations based on the contents of working memory. Efficient match algorithms are very important since the matcher often dominates all other computations and determines the speed of execution. To facilitate the discussion of the match algorithms, it is convenient to use terminology borrowed from database theory.

With each condition element of a production, we associate a *relation* (in the database sense). The attributes of the relation are the field names of the condition element. The tuples of the relation are the working memory elements that match the condition element. Using this formalism, it is easy to see that the set of all instantiations of a production with condition elements $C_1, \ldots, C_n$, and associated relations $R_1, \ldots, R_n$, respectively, is the same as the join of the relations $R_1, \ldots, R_n$ (written either as $R_{1\ldots n}$ or $R_1 \bowtie \ldots \bowtie R_n$). The join tests correspond exactly to the tests that need to be done to check that the working memory elements satisfying the condition elements induce consistent variable bindings.

[1] Soar also allows negated condition elements. While the results take into account the effect of negated condition elements, for brevity we shall not talk about them in this paper. The interested reader is referred to [Nayak *et al.*], a longer version of this paper.

[2] Productions are actually input to Soar in a more compact form.

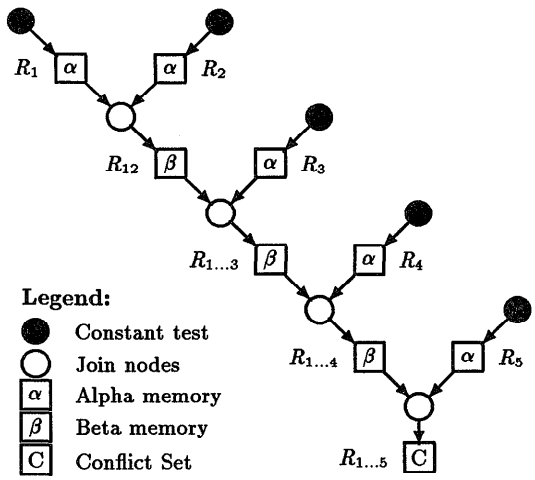


Figure 1: A RETE network.

Different algorithms may be used to solve the production match problem. In this paper we discuss the relative merits of two of them—the RETE match algorithm, and the TREAT match algorithm. Only some of the details are provided here. More complete descriptions can be found in [Nayak *et al.*].

2.3 The RETE matcher for Soar

The RETE matcher [Forgy, 1982] is the most commonly used matcher for OPS5-like systems (Soar being one of them). For a production with condition elements $C_1, \ldots, C_n$, and associated relations $R_1, \ldots, R_n$, respectively, the RETE matcher creates a data flow graph like the one shown in Figure 1 (where $n = 5$). The tuples of the relations $R_1, \ldots, R_n$ are stored in *alpha memories*, the tuples of the relations $R_{12}, \ldots, R_{1\ldots n-1}$ (called *intermediate relations*) are stored in *beta memories*, and the tuples of the relation $R_{1\ldots n}$ are stored in the *conflict set*. Each alpha memory has an associated *constant test* node that contains the tests necessary to decide whether or not a working memory element matches, in isolation, the condition element corresponding to that alpha memory. *Join nodes* contain the tests necessary to join two input relations and produce an output relation. The network specifies the order in which the joins are done.

New working memory elements that satisfy the tests in a constant test node are added to the relation in the corresponding alpha memory. Any change to a relation (stored either in an alpha or a beta memory) is propagated down the network by joining the changed part of the relation with the opposite relation (the relation that it shares the join node with), possibly adding tuples to the output relation. This propagation stops when either the conflict set is updated, or when no new tuples result from a join. Removal of working memory elements is handled in an analogous fashion. The only difference is that the addition of tuples to the relations is replaced by the removal of tuples.

2.4 The TREAT matcher for Soar

TREAT stores the relations $R_1, \ldots, R_n$ in alpha memories (with associated constant test nodes as above) and the relation

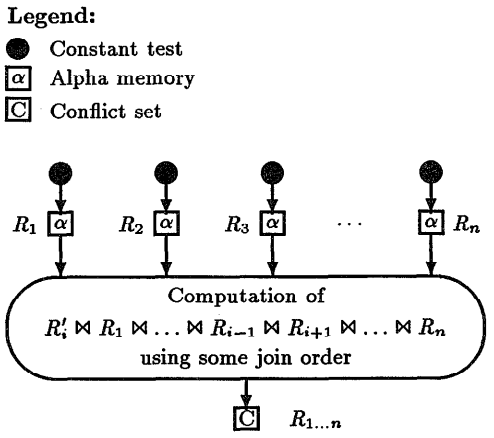


Figure 2: A TREAT network.

$R_{1\ldots n}$ in the conflict set. However, unlike RETE, the intermediate relations are not stored, but parts of them are recomputed as and when required (see Figure 2).

As in RETE, working memory elements that satisfy a constant test are added to the relation in the corresponding alpha memory. Suppose that a working memory element is added to a relation R_i. This working memory element is called the *seed*. Clearly, any new instantiation of the production resulting from this addition must contain the seed. If we let R'_i represent a relation with the same attributes as R_i, but having only one tuple—the seed working memory element, then it is easy to see that the relation corresponding to the new instantiations of the production is exactly the relation $R'_i \bowtie R_1 \bowtie \ldots \bowtie R_{i-1} \bowtie R_{i+1} \bowtie \ldots \bowtie R_n$. TREAT computes precisely this relation by joining one relation at a time, starting with the seed relation, R'_i. The tuples of this relation are added to the conflict set, $R_{1\ldots n}$, to complete the update. One thing to note is that since the join operation is commutative, the relations may be joined in any order.

The removal of a working memory element from a relation R_i results in the removal of those tuples of $R_{1\ldots n}$ that contain the removed working memory element (recall that the tuples of $R_{1\ldots n}$ are sets of working memory elements).

3 Issues in ordering

The order in which relations are joined in either the RETE or the TREAT algorithm is very important for good performance. A bad join order can generate large intermediate relations, making the matcher very inefficient. Since we wanted to compare the RETE and TREAT matchers, factoring out the effect of poor join orders was very important. In generating good join orders, various heuristics (especially domain dependent ones) are very important.

The RETE ordering algorithm for Soar orders the condition elements statically. The interested reader is referred to [Scales, 1986] for the exact details of the algorithm. The key idea in the RETE ordering algorithm is to join condition elements that are *linked* to the already joined condition elements. A condition element is said to be linked to the already joined condition elements if the variable in the identifier field of the condition element is bound in the already joined condition elements (the identifier field of a condition element is always a variable). Since the identifier of an object is unique, only the augmentations of that object can match. Furthermore, most condition elements in Soar have constant attribute fields, with one value per attribute. Thus the sizes of intermediate relations do not increase on joining linked condition elements.

Table 1: Program information.

Feature	EP	M&C	R1-Soar	NM
Num. of prods.	71	78	334	419
Avg. CEs/prod.	8.8	8.9	9.9	8.8
Num. of prod. firings	796	1617	3732	2173
Total WMEs added	1715	6609	7810	4554
Total WMEs removed	1313	5620	6458	4336

TREAT can create the join orders either at run-time—each time a sequence of joins needs to be done (called dynamic ordering)—or at the time the production is loaded into the system (called static ordering).

While various dynamic ordering algorithms were tried (for example, joining the smallest relation first, or joining the relation with the largest variable overlap first—see [Nayak *et al.*] for details), the following algorithm (again using the notion of being linked) proved to be the best. Starting with the seed condition element, at each stage a condition element that is linked to the previously joined condition elements is joined. If two or more condition elements are linked, then the one with the smallest corresponding relation is picked.[3]

Various static ordering schemes were also tried [Nayak *et al.*]. Creating a RETE-style join order starting with each condition element of each production makes loading productions very slow. The use of Miranker's *seed ordering* (in which the join order for each condition element is essentially the RETE join order with the seed condition element promoted to the top of the order [Miranker, 1987]) also proved inadequate. An extension of seed ordering proved to be the best. Instead of promoting a single condition element, a variable number of condition elements are promoted. The first unpromoted condition element in the RETE order is required to be linked to the promoted condition elements. Each promoted condition element is required to be linked to the previously promoted condition elements. Also if no condition element exists that is linked to the already promoted condition elements, then the promotion stops and the original RETE order is used. This is a cross between seed ordering and generating pure RETE-style orders, and works quite well.

4 Results and Discussion

Experimental results for the three production matchers for Soar described above (RETE, TREAT with dynamic ordering, and TREAT with static ordering) are now presented. Miranker uses the number of comparisons required to compute variable bindings as the metric in his studies. This metric is dependent on the actual implementations of the algorithms. For example, the use of hashing in the alpha memories can cut down the number of comparisons by as much as a factor of 10 [Gupta, 1987]. Similarly, using execution time alone is not a good idea due to the differing levels of optimization of the various matchers.

The metric that we have chosen is the total number of *token changes*. A token is merely a synonym for a tuple, and we use it to be consistent with other literature on this subject. The number of token changes is the sum of the number of tokens generated and the number of tokens removed from the various relations. In TREAT, this includes the tokens generated in the recomputation of intermediate relations. This metric is clearly independent of details of implementation, and depends only on the match algorithm and the ordering algorithm. Given

[3]The complete algorithm also takes into account some special cases and is described in [Nayak *et al.*]

Table 2: Total token changes.

Algorithm	EP	M&C	R1-Soar	NM
Rete	28,293	53,982	125,151	79,905
Treat (Dynamic Order)	31,692	158,432	136,456	134,472
Treat (Static Order)	27,390	138,093	292,046	181,164

the amount of experimentation we have done with the ordering algorithms, we are fairly sure that close to optimal orders are being generated. This means that the number of token changes will in fact reflect the differences in the basic match algorithms.

The experiments were run on four different Soar programs—the Eight Puzzle (EP), the Missionaries and Cannibals (M&C), R1-Soar, and Neomycin-Soar (NM). EP and M&C are the standard toy tasks. R1-Soar is an implementation of a subset of the expert system R1 in Soar [Rosenbloom *et al.*, 1985]. R1 configures computer systems for DEC. Neomycin-Soar is an implementation of a portion of the expert system Neomycin in Soar [Washington and Rosenbloom]. Neomycin diagnoses infectious diseases like meningitis. For each program, Table 1 shows the number of productions, the average number of condition elements per production, the total number of production firings, and the number of working memory elements added and removed during the run. It is interesting to note that there are, on the average, about 9 condition elements per production compared to an average of a little over 3 in OPS5 programs [Gupta, 1987].

4.1 Experimental Results

The total number of token changes for each of these programs using the three different match algorithms is displayed in Table 2. Thus for Neomycin-Soar, the largest system, RETE generated 79,905 token changes, TREAT with dynamic ordering generated 134,472 token changes, and TREAT with static ordering generated 181,164 token changes. It is clear from the results that the RETE matcher outperforms both the TREAT matchers in almost all the cases (EP is an exception in which all the matchers perform approximately equally well).

4.2 Discussion

One of the advantages of TREAT is that no joins need to be done when a working memory element is removed from a condition element. Even when working memory elements are added to a condition element, no joins need to be done if there is a condition element in the corresponding production that has no working memory elements satisfying it. RETE has neither of these advantages. Joins need to be done when working memory elements are removed to keep the intermediate relations up to date. When a working memory element is added to a condition element, joins may still need to be done even if a condition element in the corresponding production has no working memory elements satisfying it. This is again needed to keep the intermediate relations up to date.

However, RETE has an advantage that TREAT does not have. Consider a production with condition elements C_1, C_2, C_3, C_4, and C_5 with associated relations $R_1, \ldots, R_5$ respectively. Suppose that the RETE ordering of the condition elements is as above. Further suppose that all the condition elements have some working memory elements satisfying them, but the relation $R_1 \bowtie R_2$ is null. If we have a new working memory element satisfying C_3, then clearly RETE will do no work since the first step is to join the seed working memory element with the relation $R_1 \bowtie R_2$, which is null. However, TREAT will have to do some joins before discovering that no production instantiation can be found. Thus RETE does no joins if the opposite memory is empty.

Table 3: Percentage of time that no joins need to be done.

Algorithm	EP	M&C	R1-Soar	NM
Rete	79%	66%	81%	76%
Treat	69%	60%	83%	81%

Table 4: Average length of chains.

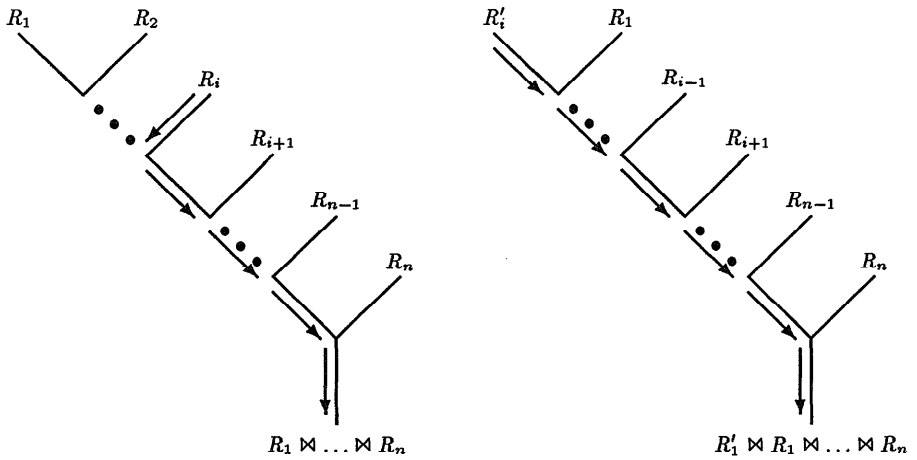
Algorithm	EP	M&C	R1-Soar	NM
Rete	1.87	1.21	1.26	1.30
Treat (dynamic)	2.31	2.32	1.78	1.84
Treat (static)	2.05	2.14	1.98	1.98

Table 3 shows the percentage of time that no joins need to be done using RETE and TREAT (clearly the number for both TREAT matchers is the same). Thus in R1-Soar, RETE does no joins 81% of the time (because of empty opposite memories), and TREAT does no joins 83% of the time (because either a working memory element was being removed or because there was a condition element with no working memory elements satisfying it). The numbers are reasonably close which means that the competing advantages offset each other pretty well.

Since all three matchers need to do joins for about the same percentage of times, the difference in the number of tokens generated points to the fact that the TREAT matchers generate more tokens when they do joins, than does the RETE matcher. To understand the reasons for this, we introduce the notion of *long chains*.

Adding or deleting a working memory element from a condition element can result in a sequence of joins being performed until either there are no more relations to join or a null relation results. The more the number of relations that need to be joined, the more is the number of tokens generated. This is called the *long chain* effect. As Table 1 shows, the average number of condition elements in a Soar production is fairly large. This means that the long chain effect is probably quite significant. Table 4 tabulates the average length of the chain of joins in the three systems, given that at least one join needs to be done. It is evident from the table that the TREAT matchers seem to generate longer chains on the average than does the RETE matcher. This helps to explain the reasons for a larger number of tokens being generated in the TREAT matchers than in the RETE matcher. The reasons for the longer chains in TREAT are discussed next.

Firstly, every time a production fires, TREAT joins in each condition element to the seed working memory element. Given the number of condition elements in a typical Soar production, this is clearly a long chain. In RETE however, all the condition elements need not be joined every time a production fires. This is because some of the condition elements (the ones before the condition element that satisfies the seed working memory element) would have already been joined in previous cycles. Thus production firings lead to longer chains in TREAT than in RETE. Figure 3 shows this pictorially. Figure 3a) shows a



(a) A long chain in RETE

(b) A long chain in TREAT

Figure 3: Long chains when a production fires.

typical long chain in RETE resulting from a production firing. Figure 3b) shows a long chain in TREAT resulting from a production firing (the seed condition element has been promoted to the top of the order in the diagram). Note how the TREAT long chain is longer than the RETE long chain.

Even when the production does not fire, TREAT will tend to have longer chains before the null relation results than RETE. To see this, suppose that n condition elements have to be joined before the null relation results. In TREAT all n will have to be joined. In RETE however, only some of these n condition elements will have to be joined, since the ones before the seed would have already been joined in previous cycles.

The second major reason for the RETE matcher performing better than the TREAT matchers can be traced back to the dichotomy of saving intermediate states versus recomputing them.

Consider a production that fires only when the current state of problem solving is equal to some desired state. Such a production would have condition elements that match the current state and condition elements that match the desired state. Since the desired state is usually fixed for a given run of the program, it is possible for RETE to match it just once when it is initially set up in working memory. TREAT, however, must rematch it (or at least a part of it) every time a change is made to the current state. Thus any time that a production needs to match a static structure (like the desired state), TREAT is more expensive.

The matching of a certain class of dynamic structures also favours RETE. This is the class of *monotonic* structures. We say that a structure is monotonic if it is built in working memory without any component working memory element being removed. The only time component working memory elements are removed is when the whole structure is removed from working memory. Matching of such structures using the RETE algorithm guarantees that each join is computed once when the structure is built and once when the structure is removed. However, since TREAT does not save the results of intermediate joins, it may have to recompute some of the joins each time a working memory element was added to the structure. This could happen when none of the alpha memories in the production is empty (for example, if a similar structure had already been matched). This can become quite expensive, making RETE perform better than TREAT.

Monotonic structures are quite common in Soar. This is due to various special features of Soar (for example, Soar productions cannot remove elements from working memory). Any production system in which monotonic structures are common (for whatever reason) would tend to exhibit the above phenomenon.

The third reason for the difference in performance has to do with the effect of *combinatorial joins*. A combinatorial join is one which results in more tuples than are in either of the input relations. A combinatorial join may occur either when one joins an unlinked condition element or when one joins a linked condition element with multiple values for the specified attribute. A combinatorial join penalizes both RETE and TREAT. However, since TREAT prefers to recompute joins, it would recompute the combinatorial joins as well. The recomputation of combinatorial joins is clearly quite expensive. RETE, on the other hand, would save the result of a combinatorial join, thus saving a significant amount of work.

An associated shortcoming of TREAT is the relatively high frequency of unlinked condition elements in the join orders. While it is the norm to write productions with condition elements that can be ordered such that, starting with a goal condition element, each condition element can be linked to the previously joined condition elements (as is done in RETE), it is not always possible to write productions with condition elements that are linked to previously joined condition elements when starting with an arbitrary condition element in the production (as is required in TREAT). This leads to TREAT join orders having more unlinked condition elements than RETE orders, leading to more combinatorial joins in TREAT.

4.3 Implementation Issues

The discussion so far has been purely in terms of token changes. We now present some results that show that TREAT has some other practical shortcomings.

Table 5 shows the time per token change in RETE and TREAT with dynamic ordering. The time per token is clearly smaller for RETE. While part of the difference can be attributed to inefficiencies in our encoding of TREAT, the difference seems large enough that it seems highly unlikely that more optimization would make TREAT with dynamic ordering perform better than RETE. Part of the reason for TREAT's poor performance in this case is due to the time spent in dynamically creating the join orders. Table 6 shows the fraction of time that TREAT spends in dynamically ordering the joins. This table clearly shows that a significant fraction of the time

Table 5: Match time per token change (in ms).

Algorithm	EP	M&C	R1-Soar	NM
Rete	1.77	2.77	3.11	2.98
Treat (dynamic)	4.94	8.06	4.82	5.72

Table 6: Fraction of total match time spent in dynamic ordering in TREAT.

EP	M&C	R1-Soar	NM
41.29%	21.10%	25.96%	22.7%

is spent in ordering, a cost completely absent in RETE.

Static ordering in TREAT can be made as efficient as RETE. However, for each production, TREAT must create a complete join order for each condition element, while RETE needs to create only one. Given that there is an average of 9 condition elements per production, loading productions in TREAT can take an average of 9 times longer than in RETE, making it quite impractical.

5 Conclusions

We have presented empirical evidence, based on four Soar programs, showing that in most cases RETE is a better match algorithm than TREAT. This seems to contradict the results that Miranker reported recently [Miranker, 1987]. Part of this difference may be because of the metric used by Miranker. Counting the number of comparisons is dependent on the implementation (for example, the use of hashing can decrease the number of comparisons 10-fold), and may not actually reflect the intrinsic differences between the match algorithms. We have also identified specific features of Soar programs that contribute to this difference.

One of the major advantages of TREAT—not having to do any joins when either there is a condition element with no matching working memory elements, or when working memory elements are removed—is completely offset by RETE not having to do any joins when the opposite memory is empty. Since the fraction of times that joins are done in RETE and TREAT is about the same, the longer chains of joins generated in TREAT lead to a larger number of tokens being generated.

The fundamental difference between RETE and TREAT—saving intermediate relations versus recomputing them—is the second main reason for TREAT's poorer performance. The matching of static structures involves recomputation of joins in TREAT, while RETE is able to match such structures just once. RETE is also better at matching monotonic structures.

Finally, we noted that combinatorial joins tend to make TREAT perform worse than RETE. In addition, TREAT tends to have more combinatorial joins than does RETE.

We also showed that both dynamic and static ordering in TREAT do not perform as well as RETE in practice. The actual process of ordering the condition elements dynamically takes a significant amount of time, making the system run very slowly. Static ordering is not good either because the time to load in productions can go up approximately 9 fold (i.e. by the average number of condition elements in a production).

While RETE seems to be better than TREAT in most cases, there are some situations under which TREAT is comparable to RETE and may even be better. This leads to the possibility of having hybrid match algorithms (TRETE?) which allow some productions to be matched using the RETE algorithm and some to be matched using the TREAT algorithm. An even closer enmeshing of the two algorithms is possible in which parts of one production are matched using the RETE algorithm while other parts are matched using the TREAT algorithm. Deciding which parts of the productions should be matched by which algorithm is not an easy task, and is a possible direction for future research.

References

[Brownston *et al.*, 1985] Lee Brownston, Robert Farrel, Elaine Kant, and Nancy Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, 1985.

[Forgy, 1982] Charles L. Forgy. Rete : A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, September 1982.

[Gupta, 1987] Anoop Gupta. *Parallelism in Production Systems.* Morgan Kaufmann Publishers, Inc., 1987.

[Laird, 1986] John E. Laird. *Soar User's Manual (Version 4).* Technical Report ISL-15, Xerox Palo Alto Research Center, 1986.

[Laird *et al.*, 1987] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar : An architecture for general intelligence. *Artificial Intelligence*, 33:1–64, September 1987.

[Miranker, 1984] Daniel P. Miranker. Performance estimates for the DADO machine : A comparison of TREAT and Rete. In *Fifth Generation Computer Systems*, ICOT, Tokyo, 1984.

[Miranker, 1987] Daniel P. Miranker. TREAT : A better match algorithm for AI production systems. In *AAAI-87 Proceedings*, American Association for Artificial Intelligence, July 1987.

[Nayak *et al.*] Pandurang Nayak, Anoop Gupta, and Paul S. Rosenbloom. Comparison of the Rete and Treat production matchers for Soar. In preparation.

[Rosenbloom *et al.*, 1985] Paul S. Rosenbloom, John E. Laird, John McDermott, Allen Newell, and Edmund Orciuch. R1-Soar : An experiment in knowledge-intensive programming in a problem-solving architecture. *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, :561–569, 1985.

[Scales, 1986] Daniel J. Scales. *Efficient Matching Algorithms for the Soar/OPS5 Production System.* Technical Report KSL 86-47, Knowledge Systems Laboratory, June 1986.

[Steier, 1987] David Steier. CYPRESS-Soar : A case study in search and learning in algorithm design. In *IJCAI-87 Proceedings*, International Joint Conference on Artificial Intelligence, 1987.

[Washington and Rosenbloom] Richard M. Washington and Paul S. Rosenbloom. Neomycin-Soar : Applying search and learning to diagnosis. In preparation.