Proc. of the 1992 IEEE
Int. Conf. on Tools with AI
Arlington, VA, Nov. 1992

An Efficient Algorithm for Production Systems with Linear-Time Match

Milind Tambe, Dirk Kalp
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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

Abstract

Combinatorial match in production systems (rule-based systems) is problematical in several areas of production system application: real-time performance, learning new productions for performance improvement, modeling human cognition, and parallelization. The unique-attribute representation in production systems is a promising approach to eliminate match combinatorics. Earlier investigations have focused on the ability of unique-attributes to alleviate the problems caused by combinatorial match [33]. This paper reports on an additional benefit of unique-attributes: a specialized match algorithm called Uni-Rete. Uni-Rete is a specialization of the widely used Rete match algorithm for unique-attributes. The paper presents performance results for Uni-Rete, which indicate over 10-fold speedup with respect to Rete. It also discusses the implications of Uni-Rete for non-unique-attribute systems.[1]

1. Introduction

Production (rule-based) systems are used extensively in the field of AI [13, 18, 25, 35]. In production systems, computation proceeds by repeated execution of *recognize-act* cycles. In the recognize phase, productions (condition-action rules) in the system are matched with a set of data-items, called working memory. In the act phase, one or more of the matched productions are fired, possibly changing the working memory, and causing the system to execute the next recognize-act cycle.

This paper focuses on a key performance bottleneck in production systems: production match. Production match is combinatoric (NP-hard), and it has proven to be a problem in several areas of production-system application. In real-time production systems, combinatorial match hinders the achievement of guaranteed response times [13, 28]. In systems that learn new productions, combinatorial match can cause a slowdown with learning [19, 33]. Combinatorial match is also problematical for modeling human cognition [25] and parallelization [1, 28].

In [30, 33], the *unique-attribute* representation was introduced to eliminate combinatorics from production match. *With unique-attributes, match becomes linear in the number of conditions.* The unique-attribute approach appears to be a promising one to alleviate the problems caused by combinatorial match without sacrificing production-system functionality. To date, the unique-attribute representation has been applied successfully to a variety of tasks, including some complex tasks with 500 or more productions [16, 17, 28].

The linear match bound in unique-attributes can provide an *additional* benefit: specialized implementation techniques that yield additional speedups. These techniques include: specialized match algorithms, specialized uni-processor hardware, and low-overhead parallelism. This paper investigates the first technique — it reports on a match algorithm for unique-attributes called *Uni-Rete*. Uni-Rete is a specialization of the widely used Rete match algorithm [7]. In tasks done in Soar [14], *Uni-Rete has shown over 10-fold speedup over Rete.*

While the speedups from Uni-Rete are presented within the context of unique-attribute Soar systems, Uni-Rete has wider implications. First, Uni-Rete is not limited to Soar; it can be used in all unique-attribute production systems. Second, Uni-Rete can be combined with Rete, and this combination can provide performance improvement in systems that are only partially based on unique-attributes. Third, Uni-Rete illustrates a possible method for reducing the memory management overhead in match algorithms. Finally, the basic premise in Uni-Rete — exploiting the representation scheme employed by building a more specialized implementation — may be applied in other systems to obtain better performance. These issues are discussed in further detail in Section 7.

This paper is organized as follows: Sections 2 and 3 provide background information about Rete and the unique-attribute representation. Since Uni-Rete is a specialization of Rete for unique-attributes, it is necessary to understand both Rete and unique-attributes to understand Uni-Rete. Section 4 introduces Uni-Rete. Section 5 describes a more optimized version of Uni-Rete. Section 6 presents the results from the Uni-Rete implementation. Section 7 discusses related work and implications of Uni-Rete for other systems. Finally, Section 8 summarizes the results and describes issues for future work.

2. The Rete Match Algorithm

The Rete match algorithm [7] is based on two main optimizations: *sharing* and *state-saving*. Sharing common parts of condition elements (also called conditions or CEs) in a single production or across productions reduces the number of tests required to do match. State-saving accumulates partially completed matches from previous recognize-act cycles for use in future cycles. Thus, if a new working-memory element (*wme*) is added in a new cycle, only the new wme is matched; older wmes

[1]This 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. 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.

0-8186-2905-3/92 $03.00 © 1992 IEEE

from previous cycles are not re-matched.

We will use the simple production system shown in Figure 2-1-a to illustrate Rete's operation. It shows a production P1, which is to be matched with the wmes W1,..,W6. Production P1 has three conditions and one action. The symbols in the conditions are either constants, e.g., STATE, that test if identical constants appear in identical positions in the wmes, or variables (enclosed in <>) that bind to symbols appearing in identical positions in the wmes. Production match involves finding *all* possible instantiations of the production with the wmes. Here an instantiation refers to a collection of wmes that provide consistent bindings for the variables in the production. In Figure 2-1-a, wmes W1, W4 and W6 form an instantiation, since they provide a consistent binding B1 for the variable <B>.

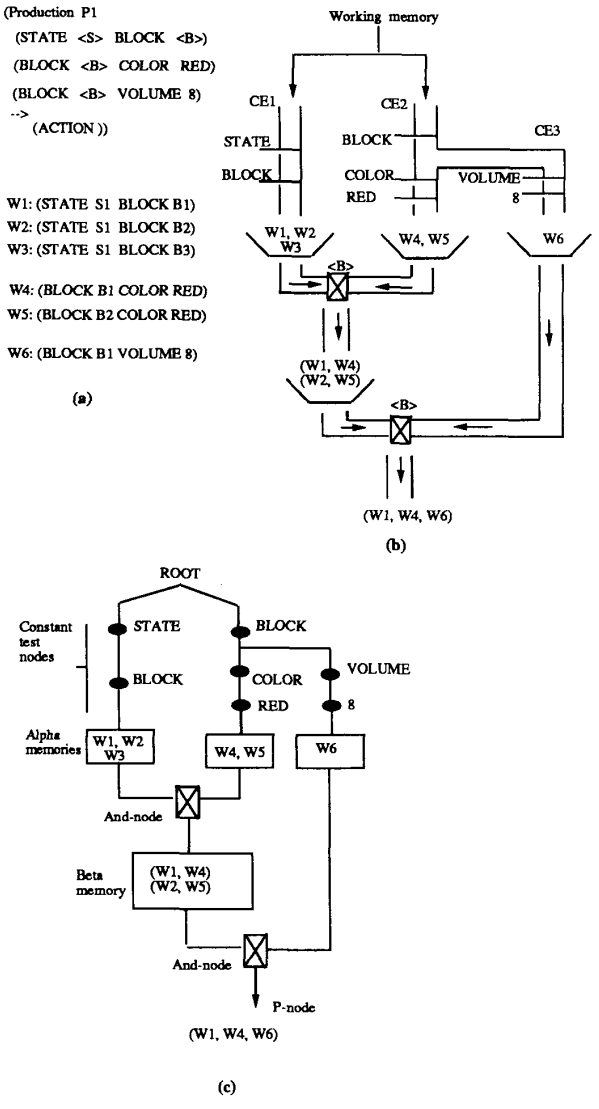

Figure 2-1: A simple production system and its Rete network.

Rete's operation in matching production P1 with the wmes W1,..,W6 can be understood using the analogy of water-flow through pipes. As shown in the upper part of Figure 2-1-b, each condition of production P1 can be considered as a pipe, ending in a bucket. The wmes flow through these pipes. Each pipe has filters associated with it, which correspond to the constant tests in the condition and allow only particular wmes to pass through. For example, the filters in the first pipe (corresponding to the first condition) check if STATE and BLOCK appear in the first and third fields of a wme respectively. Due to these filters, only W1, W2, and W3 pass through the first pipe and appear in the bucket for the first pipe. Note that since the filter BLOCK for the second and third pipes tests an identical field of the wmes, this filter is shared between the two pipes, illustrating the sharing optimization in Rete in a simple form.

Next, the small boxes with Xs inside them and <B> written over them check for consistency between wmes. Consider the box that joins the buckets for the first two conditions. Since <B> appears in the fourth field of the first condition and the second field of the second condition, this box tests the fourth field of wmes W1, W2 and W3 against the second field of wmes W4 and W5. Two pairs of wmes — (W1,W4) and (W2,W5) — pass this consistency test and are stored in the following bucket. Now the second small box checks the bindings for <B> of each of the two pairs against W6. The combination of W1,W4 and W6 is finally found to be successful and forms the instantiation of this production.

Note that the buckets in Figure 2-1-b realize state-saving by storing partial results of the match. If a new wme, say W7, is added in a new recognize-act cycle, then only W7 will be matched; Rete will avoid re-matching W1,...,W6 in this later cycle. The penalty for state-saving is that if a wme, say W4, is deleted then it is re-matched following a course identical to its addition. During this process, W4 and all combinations containing W4, such as (W1, W4), are deleted from the buckets. However, this penalty for deletion is usually quite small compared to the benefit of state-saving [8, 9]. (See also [20, 24].)

In an actual Rete implementation, the pipes discussed above form a dataflow network as shown in Figure 2-1-c. Wmes travel down from the ROOT node. The filters that test for constants are called constant test nodes. The buckets that store individual wmes are called alpha memories. The wmes stored in alpha memories are called *right tokens*. Combinations of wmes, e.g., (W1, W4), are called *left tokens* (just *tokens* if the context is clear) and are stored in beta memories. And-nodes perform consistency-checks, while P-nodes add and delete instantiations.

In Rete, *combinatorics in production match translates into combinatorial numbers of tokens in beta memories* [29]. In the worst case, there can be $O(W^C)$ number of tokens generated in matching a single production, where W is the number of wmes in the system and C is the number of conditions in a single production. Since a beta memory can be a host to such a large number of tokens (which cannot be predicted at compile time), Rete uses dynamic structures such as linked lists to store the tokens. More optimized versions of Rete, such as the one used in this paper, often rely on hash tables [9, 10]. Despite this, it is estimated that beta memories consume a majority of the time in match [9].

3. The Unique-attribute Representation

Combinatorics arise in production match due to the ambiguity about which wmes can actually match a production's condition. In a production with multiple conditions, a cascading effect of such ambiguity leads to an exponential match effort. This cascading ambiguity is reflected in the formation of a large number of left tokens while matching a single condition — each left token encodes a particular partial result of match. The unique-attribute representation eliminates this ambiguity. With unique-attributes, each condition leads to the formation of at most one left token, thus eliminating combinatorics from production match.

We will demonstrate the unique-attribute representation using the standard triple-based *(object attribute value)* representation. Optionally, a class field describing the class of the object may be present, as in: *(class object attribute value)*. This is the representation used in Figure 2-1, where wmes are of the form: *(STATE S1 BLOCK B1)*, and also in our experiments with Uni-Rete. In this representation, *unique-attributes* refers to allowing only a unique value given fixed class, object and attribute fields. Thus, the wmes in Figure 3-1-a do not adhere to the unique-attribute constraint. This is because B1, B2 and B3 are three different values given fixed class, object and attribute fields, i.e., STATE, S1, and BLOCK respectively. The presence of multiple values given the three fixed fields is what gives rise to the ambiguity in the match. With unique-attributes, the same wmes would be represented either as in 3-1-b or some alternate form, so as to allow only a unique value given fixed class, object and attribute fields.

(a)	(b)
(STATE S1 BLOCK B1)	(STATE S1 BLOCK B1)
(STATE S1 BLOCK B2)	(BLOCK B1 LEFT B2)
(STATE S1 BLOCK B3)	(BLOCK B1 RIGHT B3)

Figure 3-1: (a) Unrestricted representation (b) Unique-attributes.

Given this constraint on the wmes and some related constraints on the conditions matching them, unique-attributes guarantee that the match cost is linear in the number of conditions. That is, beta memories can contain a maximum of one token, and not a combinatorial number of them as in an unrestricted system (i.e., a system without the unique-attribute restriction).

Note that unique-attributes achieve much more than a simple redistribution of the match of one unrestricted production among many unique-attribute productions. In many situations, especially those involving expensive learned rules, the unrestricted representation causes a large (even exponential) amount of unnecessary match effort. Unique-attributes get rid of such unnecessary match effort, and thus provide a big performance improvement.

To date the unique-attribute representation has been applied successfully to a variety of tasks encoded in Soar, including some complex tasks with 500 or more productions [3, 16, 17, 28]. In these tasks, unique-attributes have improved problem-solving performance and eliminated expensive learned rules; thus, allowing researchers to develop large production systems without getting bogged down in efficiency problems. However, further speedups, as with Uni-Rete, are still useful.

For its efficiency gains, the unique-attribute representation does tradeoff some expressiveness. In particular, with unique-attributes, it becomes difficult to encode unstructured sets in working memory. For instance, in Figure 3-1-a, blocks B1, B2 and B3 are an unstructured set of blocks in state S1. With unique-attributes, all sets in working memory have to be structured, e.g., as in Figure 3-1-b, or using some simpler structure such as a list etc. Since this set structure impacts problem-solving efficiency, additional encoding effort may be required.

While unique-attributes are investigated in Soar in detail, they are also relevant for other systems. The central idea in unique-attributes is the elimination of cascading ambiguity in the match, which should be applicable to production systems such as Prodigy [19], OPS5 [5] and others. In fact, unique-attributes have already been applied, in a more generalized form, in production system languages such as **K** [4] and PAMELA [2]. Unique-attributes also appear relevant to frame-based systems. For instance, Chalasani and Altmann [6] have pointed out that the knowledge representation scheme adopted by Theo, an integrated frame- and rule-based architecture for problem-solving [23], corresponds to the unique-attribute representation. Thus, algorithms such as Uni-Rete, developed for unique-attributes, may be useful in speeding up a variety of systems. Further details on this and related issues appear in [28, 31, 33].

4. Introducing Uni-Rete

Figure 4-1 illustrates the implication of unique-attributes for Rete. Each beta memory contains only a single token. However, despite this single token, Rete goes through the process of creating and storing it as usual, and incurs a large memory management overhead. Note that the alpha memory for the first condition also contains only a single wme.

Uni-Rete exploits this bound of a single token per beta memory by storing tokens implicitly and reducing the token memory management overhead. In particular, only a small part of a token is stored in a given beta memory, the rest being implicit from the preceding beta memories. Thus, instead of storing tokens as shown in Figure 4-1, they can be stored as shown in Figure 4-2. Consider the beta memory containing the token (W1, W4) in Figure 4-1. In the corresponding beta memory in Figure 4-2, only W4 is stored. Since the preceding alpha memory contains only a single wme (W1), the token (W1, W4) is implicit from the two memories. Similarly, instead of storing the entire token (W1, W4, W6) as in Figure 4-1, only W6 is stored in Figure 4-2, the remaining portion of that token being implicit from the preceding memories. Thus, tokens can be stored implicitly, by storing only a single wme. The space for this single wme can be allocated in advance, thus avoiding the memory management overhead of Rete. This single location per wme actually stores a pointer to a wme and will be referred to as a wme-pointer location. In Figure 4-2, the locations storing wmes W1, W4, W6 and W8 are such wme-pointer locations.

Note that this implicit storage cannot be used if the beta memories contain multiple tokens, as this will create ambiguity about which wmes are actually supposed to form a token. Thus, the technique of implicit storage *cannot* be used in an unrestricted system.

Figure 4-3 describes the Uni-Rete algorithm using the two working-memory operations: add and delete. We will explain the steps in the algorithm using the example in Figure 4-2. Suppose the wmes W6 and W8 are not yet present in Figure 4-2. Now suppose wme W6 is added. In step 1 of the addition

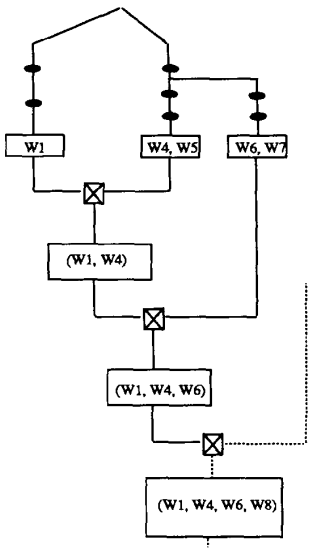

Figure 4-1: Rete with Unique-attributes.

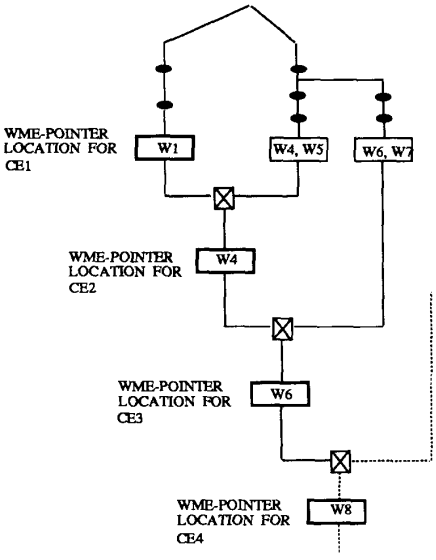

Figure 4-2: Uni-Rete.

algorithm, W6 passes the constant tests and is stored in the alpha memory for the third condition (CE3) as shown in Figure 4-2. In step 2, we check the wme-pointer location for the previous condition (CE2). This location is non-null (it contains W4) and hence we proceed to step 3. For step 3, suppose some variable <Y> appears in CE2 and CE3. Then we check the consistency of the bindings for <Y> between W4 and W6. With a successful consistency test, we proceed to step 4. In step 4, we store a pointer to W6 in the wme-pointer location for CE3, i.e., CE3 is now successfully matched. In step 5, the alpha memory for CE4 is tested for a wme that is consistent with wmes in the preceding wme-pointer locations. If this test succeeds, we assign a wme-pointer to the wme-pointer location for CE4. We continue to apply step 5 until either the consistency tests for some following CE fail or the consistency tests for the last CE succeed. In the latter case, the production is instantiated in step 6.

Now, suppose wme W6 is deleted. In step 1 of the deletion algorithm, it passes the constant tests and is removed from the alpha memory of CE3. In step 2, we check if the wme-pointer location of CE3 points to W6 and find that it does. This implies that, earlier, W6 had successfully matched CE3. In step 3, we nullify the wme-pointer location of CE3, thus removing the match to W6. The match for all the conditions following CE3 depend on the match for CE3, i.e., if CE3 no longer matches a wme, then the conditions (CE4,...,CEN) also cannot match. Therefore, in step 4, we enter null into the wme-pointer locations of CE4 through CEN. Finally, in step 5, if the production was earlier instantiated, then its instantiation is deleted.

---

* *Comments: Let W_k be the new wme. Let the production system*
* *contain a single production P with conditions $C_1, C_2, ..C_k, ..C_N$,*
* *such that W_k passes the constant tests of C_k. For a condition*
* *C_k: alpha-memory(C_k) indicates its alpha memory,*
* *wme-pointer(C_k) indicates its wme-pointer location, tests(T_k)*
* *indicate its consistency tests. Instantiation(P) refers to the*
* *instantiation of P.*

Uni-Rete: Procedure for addition of a wme

1. Perform constant tests and store W_k in alpha-memory(C_k);
2. If (wme-pointer(C_{k-1}) = NULL) then return;
3. If tests(T_k) do not succeed using W_k then return;
4. wme-pointer(C_k) := pointer to W_k;
5. While ((tests(T_{k+1}) succeed for some wme W_j from alpha-memory(C_{k+1})) and (k < N))
   wme-pointer(C_{k+1}) := pointer to W_j;
   k := k+1;
6. if (k = N) then instantiate production P;

Uni-Rete: Procedure for deletion of a wme

1. Perform constant tests and delete W_k from alpha-memory(C_k);
2. If (wme-pointer(C_k) <> pointer to W_k) then return;
3. wme-pointer(C_k) := NULL;
4. for (j = k+1 to N) wme-pointer(C_j) := NULL;
5. if P is instantiated, remove instantiation(P);

Figure 4-3: Uni-Rete: procedures for adding/deleting a wme.

The extension of Uni-Rete to handle negated conditions is straightforward: If a wme matching a negated condition is added to the system, then it follows the normal procedure for adding wmes, except that steps 5 and 6 are replaced by steps 4 and 5 for deleting wmes — since a successfully matched negated condition does not allow the following conditions to match. If a wme matching a negated condition is deleted, then it follows the normal procedure for deleting wmes except that steps 4 and 5 are replaced by steps 5 and 6 for adding wmes — since a non-matching negated condition allows the following conditions to match. The Uni-Rete algorithm is described in further detail in [34].

As noted earlier, the absence of left tokens is the major

optimization in Uni-Rete that allows it to outperform Rete. There are three savings due to this optimization during the addition of a wme. First, since the space for the wme-pointer location is allocated in advance, Rete's memory allocation/de-allocation costs are avoided. Second, since only a single wme-pointer is stored in a wme-pointer location, Uni-Rete avoids the cost of copying pointers to all the previous wmes into the current token. For instance, in Figure 4-1, Rete copies pointers to W1 and W4 from the previous token when it is about to create a token containing (W1,W4,W6). In contrast, as shown in Figure 4-2, Uni-Rete does not require such copying. Third, Uni-Rete avoids the cost of hashing left tokens.

The absence of left tokens allows Uni-Rete to perform more efficiently during the deletion of a wme as well. Specifically, in Rete, deletion of a wme implies following a course symmetrical to its addition, and removing tokens containing the deleted wme. (While wme deletion in Rete can be optimized by maintaining some extra pointers, for a hashed version of Rete, this optimization would either provide small speedups (~10%) or cause a slowdown [2].) In Uni-Rete, this cost is avoided — the system simply zeros out all the successor locations. There are other minor optimizations in Uni-Rete as well, e.g., the instantiation processing is simplified since there is only a single instantiation, the implementation of negated conditions is also simplified, etc.

Overall, while the processing of left tokens is heavily optimized in Uni-Rete, the processing of right tokens is not optimized. Therefore, the speedup of Uni-Rete over Rete depends on the proportion of right tokens in the input. Note that Uni-Rete is a specialization of Rete, and it has identical state-saving and sharing. Thus, there are no losses or gains in Uni-Rete with respect to those optimizations.

5. Bilinear Organization of the Uni-Rete Network

This paper has, so far, focused on a linear version of Uni-Rete. That is, each condition joins the network at the end of a sequence of *all* the previous conditions in a production. Alternatively, it is possible to organize the network in a bilinear fashion, where a new condition may join the network at the end of some particular sequence of previous conditions, not necessarily all of them. Figure 5-1 contrasts a linear version of Uni-Rete with a bilinear version. In the bilinear network, conditions CE1-CE4 are organized as in a linear network. But conditions CE5 and CE6 join the network at the end of condition CE2. Thus, there are two independent chains of conditions in the bilinear network: (CE1, CE2, CE3, CE4) and (CE1, CE2, CE5, CE6). These two chains are joined together using a *super-p-node*. The super-p-node performs any consistency tests that were missed due to the bilinearization. For instance, in Figure 5-1, CE4 and CE6 appear in different branches of the bilinear network. Therefore, a super-p-node is required to perform any consistency checks between CE4 and CE6. Additionally, the super-p-node also performs the function of the p-node in Rete. Except for the super-p-node, bilinear Uni-Rete is identical to linear Uni-Rete.

Figure 5-2 shows the principle advantage of bilinear networks: an increase in the amount of sharing. Increased sharing implies fewer wme-pointer locations; which implies faster execution time. The figure shows two productions, P1 and P2. All of the conditions in P1 appear in P2, but not in the same sequence. In linear Uni-Rete, sharing is entirely dependent on a common initial subsequence of conditions. The subsequence (CE1, CE2) in P1 also appears in P2 — hence the

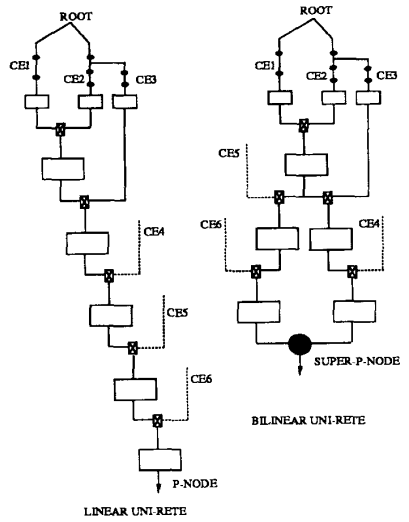

Figure 5-1: Bilinear Uni-Rete.

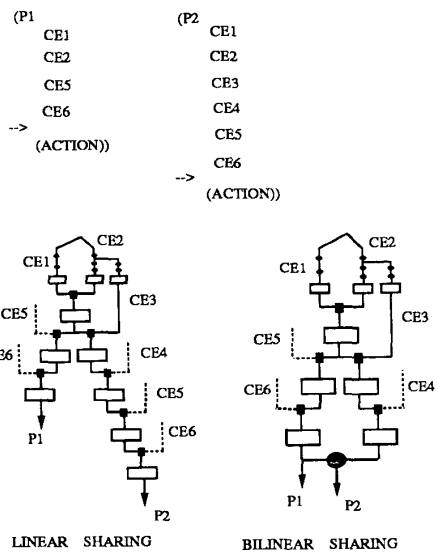

Figure 5-2: (b) Increased sharing with bilinearization.

linear Uni-Rete shares the two conditions across P1 and P2. However, the subsequence of CE5 and CE6 in P1 cannot be shared with CE3 and CE4 in P2. In bilinear Uni-Rete, sharing is not entirely dependent on a common sequence of conditions. In particular, production P2 can be organized as a bilinear network, with two chains of conditions: (CE1, CE2, CE3, CE4) and (CE1, CE2, CE5, CE6). This bilinear network shares one of its branches (CE1, CE2, CE5, CE6) with P1. With this sharing, there are fewer wme-pointer locations in the bilinear Uni-Rete network than in the linear network.

Thus, bilinear networks can lead to improved sharing. Unfortunately, while increased bilinearization leads to increased sharing, it also leads to increased consistency checks in the

super-p-nodes, thus increasing their overhead. This is particularly problematical since our current super-p-node implementation is suboptimal — it does not save partial consistency-check results for use in future cycles. Thus, a program for bilinearization needs to balance the conflicting requirements of increasing sharing and reducing super-p-node overheads. We have written a simple heuristic program called *bilin* that attempts to achieve this balance [34]. The output of *bilin* is used for the experiments in this paper.

Note that, in our experiments, the bilinear optimization was not applied to Rete, since a fair comparison requires that an optimization be applied to Rete only if it improves Rete's performance in the general case (improving Rete's performance for unique-attributes leads us to Uni-Rete). However, in the general case, bilinearization can actually degrade Rete's performance, since (i) it actually increases Rete's match activity [9], (ii) its implementation is quite complex, and could introduce a much larger overhead [15], and (iii) the benefits of sharing in an unrestricted system may not be as much as in a unique-attribute system [34].

6. Experimental Results

This section compares the performance of Rete and Uni-Rete using the Soar system. Soar is an integrated problem-solving and learning system that uses a production system for its knowledge base [14]. Soar is used for our comparison, as it is the only system in which unique-attributes have been investigated in any detail. Since integrating our C-based Rete and Uni-Rete implementations in the LISP-based Soar system is quite difficult [32], we obtained detailed traces of Soar's wme changes and used them as input for Uni-Rete and Rete. Given this input, the match activity in the C-based Rete and Uni-Rete matchers is identical to Soar's LISP-based matcher.

Figure 6-1 presents the benchmarks used. The first column provides the names of the benchmarks and the number of productions in each benchmark. The second column shows the ratio of right (alpha memory) tokens to left (beta memory) tokens. As discussed at the end of Section 4, this ratio is important in determining speedups. The third, fourth and fifth columns indicate the number of wme-pointer locations without sharing, with sharing in linear Uni-Rete, and with sharing in bilinear Uni-Rete. The numbers in parentheses are the sharing factors, i.e., ratios of unshared wme-pointer locations to shared wme-pointer locations. In the linear case, the average sharing factor is 1.6, close to the average sharing factor observed in other Rete systems [9]. In the bilinear case, the average sharing factor is 3.8, i.e., a 2.3-fold increase in sharing over the linear case.

Note that all the benchmarks are encoded in the unique-attribute representation. Thus, in our comparison, both Uni-Rete and Rete are run with identical unique-attribute-based benchmarks. Among the benchmarks, MAX is an artificial set of productions used to illustrate Uni-Rete's ability to achieve a large speedup over Rete. In particular, MAX has a very large number of left tokens compared to right tokens. *The speedup in MAX is 15.5 fold* (with linear Uni-Rete). The other benchmarks are described in [28].

Figure 6-2 shows the speedups from linear and bilinear Uni-Rete. The x-axis indicates the different benchmarks. The y-axis shows the total match time in seconds. The match times for three systems are shown: Rete, linear Uni-Rete and bilinear Uni-Rete. The numbers in parentheses are the speedups. In the linear case, the speedups are seen to be determined by right/left token ratio.

TASK NAME [NUMBER PRODS]	RATIO RIGHT/LEFT TOKENS	NO SHARING	LINEAR SHARING	BILINEAR SHARING
MAX [20]	~0	360	360	--
EIGHT1 [52]	0.14	726	489 *(1.48)*	206 *(3.52)*
EIGHT2 [62]	0.10	960	634 *(1.51)*	225 *(4.26)*
GRID [60]	0.14	1219	592 *(2.05)*	297 *(3.11)*
BLOCKS [181]	0.14	2284	1356 *(1.68)*	391 *(5.84)*
MFS • [59]	0.46	358	255 *(1.40)*	174 *(2.05)*

Figure 6-1: The benchmarks for comparing Rete and Uni-Rete.

For instance, the high speedup of 8.1 is achieved in EIGHT-2, which has a low right to left token ratio of 0.10. In fact, the correlation co-efficient between the token ratio and the speedups is -0.95, and regression analysis yields the following equation:

Speedup with linear Uni-Rete = (-11.6 * token-ratio) + 8.43

The increase in speedups from the linear to the bilinear case is also correlated with the increase in sharing from the linear to the bilinear case (a correlation coefficient of 0.91). For instance, BLOCKS shows the maximum increase in sharing (from 1.68 to 5.84 — a factor of 3.47) and a maximum increase in speedup (from 6.2 to 13.8 — a factor of 2.2). Regression analysis in this case yields the following:

Increase in speedup with bilinear Uni-Rete =
0.49*increase-in-sharing + 0.27

Overall, the speedup shown by BLOCKS is the maximum: 13.8 fold. The overall speedups in EIGHT1 and EIGHT2 are also quite substantial (9.8 and 11.2 fold respectively). Due to its high right to left token ratio and low increase in sharing, MFS* shows the lowest speedup.

7. Discussion

While the speedups from Uni-Rete are impressive, an important concern about this work is its applicability — is it restricted only to unique-attribute systems? This question is addressed in detail in this section along with a discussion of related work.

Uni-Rete can be used in production systems that are not strictly unique-attribute based. In particular, Uni-Rete is a form of Rete, and can be easily combined with Rete. This allows Uni-Rete to be used in systems that are only partially based on unique-attributes. A simple form of the Uni-Rete and Rete combination is to use Uni-Rete for unique-attribute productions, and Rete for other productions. A more complex form of combination is to use Uni-Rete within a production for its unique-attribute conditions, and Rete for other conditions. Such a combination requires that all unique-attribute conditions be placed before others, which is a useful heuristic for condition ordering [12]. In all such combinations, Uni-Rete and Rete would be able to share the constant tests and alpha memories. (Note that, as discussed in Section 3, this Uni-Rete and Rete combination would be applicable in other systems besides Soar.)

Another contribution of Uni-Rete is that it highlights the performance gains possible by reducing the dynamic memory management overhead associated with tokens. Algorithms like Rete are based on the assumption that the number of tokens

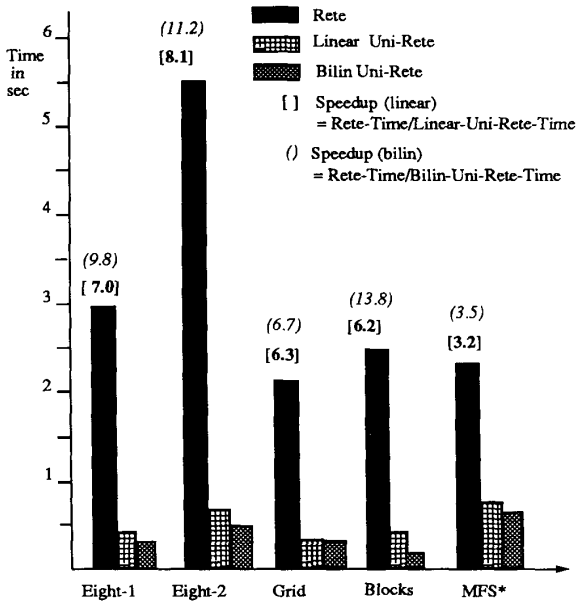

Figure 6-2: Linear/bilinear Uni-Rete speedups over Rete.

generated in matching a condition cannot be predicted in advance. As a result, these algorithms generate tokens dynamically and incur a large memory management overhead. However, in many situations, the size of the token memory can be predicted in advance. The Uni-Rete experience illustrates that such situations can be exploited by static data-structure allocations.

More fundamentally, Uni-Rete raises the issue of exploiting the representation scheme employed by building specialized match algorithms. This phenomenon of exploiting a simpler representation for a faster implementation has been observed elsewhere in computer science — particularly in the domain of RISC machines, where a simpler instruction set is observed to provide a much faster implementation [11].

How does Uni-Rete compare in performance with other match algorithms? In other words, could other match algorithms have been employed instead of Uni-Rete to obtain equivalent or better speedups? The Treat match algorithm [21] is the other match algorithm, besides Rete, that is popular in the production systems area. However, Treat's performance is equivalent to Rete rather than to Uni-Rete. In particular, a comparison between Rete and Treat yields a performance difference of less than a factor of two; and that difference is also debatable (e.g., see [24, 20]). Recently, LEAPS [22] has emerged as a promising new algorithm, providing substantial performance improvements. However, LEAPS is dependent upon LEX or related conflict-resolution strategies in OPS5, and thus cannot be used in production systems such as Soar [14] and others [2, 19] that do not depend on such conflict resolution strategies. Finally, algorithms such as Matchbox [27] and Oflazer's algorithm [26] are focused on parallel implementations, and are not as efficient for uni-processor implementations.

8. Summary and Future Work

Combinatorial match in production systems is problematical in several application areas. The unique-attribute representation is a promising approach to eliminate match combinatorics without sacrificing functionality [28, 31, 33]. The linear match bound in unique-attributes can be exploited by building specialized implementations that yield additional speedups. This paper focused on one such specialized implementation: it reported on a match algorithm for unique-attributes called *Uni-Rete*. Uni-Rete is a specialization of the widely used Rete match algorithm [7]. In tasks done in Soar, *Uni-Rete has shown over 10-fold speedup over Rete.* The paper also discussed the implications of Uni-Rete for non-unique-attribute systems.

Was the comparison presented in this paper a fair one? Both Rete and Uni-Rete systems were developed from the same starting point — CParaOPS5, a highly optimized C-based OPS5 system [10]. Furthermore, as we optimized Uni-Rete, any applicable optimizations were applied to Rete. Thus, the Rete system used for comparison with Uni-Rete is well-optimized: in fact, it is a factor of 2.5 or so faster than the optimized CParaOPS5.

Among the issues for future work, one key issue is continued optimization of Uni-Rete. Some important optimizations are currently missing, e.g., the super-p-nodes are sub-optimaly implemented, the garbage collection and constant-test routines are comparatively inefficient etc. Another interesting issue is experimenting with Uni-Rete for OPS5 systems. As discussed in Section 3, some OPS5 systems — either entire systems or portions of such systems — already adhere to the unique-attribute constraint. Uni-Rete can be applied to such systems to obtain performance improvements. A third issue is extending Uni-Rete to support all of Soar's functionality. This includes supporting Soar's ability to create subgoals. With subgoaling, Uni-Rete's assumption of a single token per beta memory is still valid, but on a per subgoal basis. Subgoaling in Soar can be supported by replacing each wme-pointer location in Uni-Rete by an array. For any single subgoal, Uni-Rete will execute exactly as described in Section 4, except that the wme-pointer locations will be indexed by the subgoal. Alternatively, Soar's functionality could be slightly restricted by allowing only the latest subgoal to match at a time, thus enabling Uni-Rete to support subgoaling without any modifications. Finally, after optimizing match, we will need to optimize other components in the production systems.

References

1. Acharya, A. and Tambe, M. Production systems on message passing computers: Simulation results and analysis. Proceedings of the International Conference on Parallel Processing, 1989, pp. 246-254.

2. Barachini, F. "The evolution of PAMELA". *Expert Systems 8*, 2 (1991), 87-98.

3. Bayazitoglu, A., Johnson, T. R., and Smith, J. W. A unique-attribute representation of annotated models that facilitates learning. Tech. Rept. OSU-LKBMS-92-100, Ohio state university division of medical informatics, 1992.

4. Bouaud, J., and Zweigenbaum, P. A reconstruction of conceptual graphs on top of a production system. Proceedings of the 7th Annual workshop on conceptual graphs, 1992.

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

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

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

8. Forgy, C. and Gupta, A. Preliminary architecture of the CMU production system machine. Hawaii International Conference on Systems Sciences, January, 1986, pp. 194-200.

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

10. Gupta, A., Tambe, M., Kalp, D., Forgy, C. L., and Newell, A. "Parallel implementation of OPS5 on the Encore Multiprocessor: Results and analysis". *International Journal of Parallel Programming 17*, 2 (1988).

11. Hennessy, J. RISC Architecture: A Perspective on the Past and Future. In Sietz, C. L., Ed., *Advances in VLSI*, MIT press, 1990.

12. Ishida, T. Optimizing rules in production system programs. Proceedings of the Seventh National Conference on Artifical Intelligence, 1988, pp. 699-704.

13. Laffey, T.J., Cox, P. A., Schmidt, J. L., Kao, S. M., and Read, J. Y. "Real-time Knowledge-Based Systems". *AI magazine 9*, 1 (1988), 27-45.

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

15. Lee, H.S., and Schor, M. Match algorithms of generalized Rete networks. Tech. Rept. RC 14709 (#65946), IBM TJ Watson research center, 1989.

16. Lehman, J. Fain, Lewis, R.L. and Newell, A. Natural Language Comprehension in Soar: Spring 1991. Tech. Rept. CMU-CS-91-117, School of Computer Science, Carnegie Mellon University, March, 1991.

17. Li, X., Krishnan, R., and Steier, D. MFS: A study of model formulation within Soar. Tech. Rept. Working paper 91-18, School of Urban and Public affairs, Carnegie Mellon University, 1991.

18. McDermott, J. "R1: A rule-based configurer of computer systems". *Artificial Intelligence 19*, 2 (1982), 39-88.

19. Minton, S. Quantitative results concerning the utility of explanation-based learning. Proceedings of the National Conference on Artificial Intelligence, August, 1988, pp. 564-569.

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

21. Miranker, D., and Lofaso, B. "The organization and performance of a Treat-based production system compiler". *IEEE transactions on knowledge and data engineering 3*, 1 (1991), 3-11.

22. Miranker, D. P., Brant, D. A., Lofaso, B., Gadbois, D., On the performance of lazy matching in production systems. Proceedings of the eigth national conference on artificial intelligence, 1990, pp. 685-692.

23. Mitchell, T. M. Becoming Increasingly Reactive. Proceedings of the eighth national conference on artificial intelligence, 1990, pp. 1051-1058.

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

25. Newell, A. *Unified Theories of Cognition*. Harvard University Press, Cambridge, Massachusetts, 1990.

26. Oflazer, K. *Partitioning in Parallel Processing of Production Systems*. Ph.D. Th., Computer Science Department, Carnegie Mellon University, 1987.

27. Perlin, M. The Match Box Algorithm for Parallel Production System Match. Tech. Rept. CMU-CS-89-163, Computer Science Department, Carnegie Mellon University, 1989.

28. Tambe, M. *Eliminating combinatorics from production match*. Ph.D. Th., School of Computer Science, Carnegie Mellon University, May 1991.

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

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

31. Tambe, M. and Rosenbloom, P. A framework for investigating production system formulations with polynomially bounded match. Proceedings of the Eighth National Conference on Artificial Intelligence, 1990, pp. 693-400.

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

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

34. Tambe, M., Kalp, D., and Rosenbloom, P. Uni-Rete: Specializing the Rete match algorithm for the unique-attribute representation. Tech. Rept. CMU-CS-91-180, School of Computer Science, Carnegie Mellon University, 1991.

35. Waterman, D. A., Hayes-Roth, F. *Pattern-directed inference systems*. Academic press, 1978.