Uni-Rete: Specializing the Rete Match Algorithm for the Unique-attribute Representation

Milind Tambe, Dirk Kalp and Paul Rosenbloom[1]

September 1991
CMU-CS-91-180 z

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

The 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 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 [Tambe, Newell and Rosenbloom 90]. 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, and it has shown over 10-fold speedup over Rete in performing match.

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.

[1]Information Sciences Institute, University of Southern California, 4676 Admiralty Way, Marina del Rey, CA 90292.

Keywords: Artificial Intelligence, Knowledge Representation, Expert system tools and techniques, Rule-based systems, Match Algorithms

Table of Contents

List of Figures

1. Introduction

Production systems (or rule-based systems) are used extensively in the field of Artificial Intelligence [Laffey et. al. 88, McDermott 82, Newell 90, Waterman and Hayes-Roth 78]. 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 the production-system computation — production match. Production match is combinatoric (NP-hard), and it has proven to be a problem in different areas of production-system application. In real-time systems based on productions systems, combinatorial match hinders the achievement of guaranteed response times [Laffey et. al. 88, Tambe 91]. In systems that learn new productions, combinatorial match can lead to an actual slowdown with learning [Minton 85, Minton 88, Tambe, Newell and Rosenbloom 90]. Combinatorial production match is also problematical for modeling human cognition [Newell 90] and has unfavorable implications for parallelizing production systems [Acharya and Tambe 89a, Tambe 91].

In [Tambe and Rosenbloom 89], 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. A detailed evaluation of the unique-attribute representation appears in [Tambe 91, Tambe and Rosenbloom 90, Tambe, Newell and Rosenbloom 90].

The linear match bound in unique-attributes can also provide an *additional* benefit: a more efficient implementation technology that yields additional performance improvements. Currently, there appear to be three dimensions of this more efficient implementation technology — specialized match algorithms, specialized uni-processor hardware, and parallelization with lower overheads — all of which require detailed investigation. This paper reports on the results of investigating the first dimension — specifically on a specialized match algorithm for unique-attributes called Uni-Rete. Uni-Rete is a specialization of the widely used Rete match algorithm [Forgy 82]. In tasks done in Soar, an integrated problem-solving and learning system [Laird, Newell, and Rosenbloom 87], *Uni-Rete has shown up to a 10-fold speedup over Rete.* This phenomenon of a simpler representation yielding 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 [Hennessy 90].

This paper is organized as follows: Section 2 describes the Rete match algorithm and the unique-attribute representation. Section 3 introduces the Uni-Rete match algorithm. Section 4 describes our experimental methodology and presents first results from the Uni-Rete implementation. Section 5 describes some modifications to Uni-Rete that lead to additional speedups and presents the results obtained after these modifications. Section 6 is a discussion of relevant issues. Finally, Section 7 summarizes the results and describes issues for future work.

2. Background

Uni-Rete is a specialization of Rete for the unique-attribute representation. To understand Uni-Rete, it is necessary to understand both Rete and unique-attributes. This section first describes Rete and then the unique-attribute representation. The description of Rete presented here is a novel one and aimed at readers unfamiliar with Rete. It only emphasizes those aspects of Rete necessary for our comparison between Rete and Uni-Rete. Readers familiar with Rete and unique-attributes may wish to skip this section.

2.1. The Rete Match Algorithm

Rete [Forgy 82] is a widely used production match algorithm. There are two main optimizations in Rete: *sharing* and *state-saving*. Sharing common parts of condition elements (also called conditions or CEs) in a single production or across different productions reduces the number of tests required to do match. In practice, the effect of sharing has turned out to be quite limited: the speedup due to sharing has been observed to be a factor of 1.1 to 1.6 [Gupta 86, Miranker 87, Tambe, et al. 88]. State saving accumulates partially completed matches from previous recognize-act cycles for use in future cycles. Even if the working-memory elements or *wmes* generated in a cycle fail to match a production, the partial match is saved. Thus, if a new wme is added in a new cycle, only the new wme has to be matched; the partial match from the previous cycles is not repeated. Recently, the Treat match algorithm [Miranker 87] has emerged as a major competitor for Rete. However, for the Soar system, which is used for the experiments in this paper, Rete has been shown to outperform Treat [Nayak, Gupta, and Rosenbloom 88].

We will use the simple production system shown in Figure 1 to illustrate Rete's operation. This figure shows a production P1, which is to be matched with the wmes (W1,W2,..,W6). Production P1 has three conditions and some appropriate action. The symbols appearing in the conditions are either constants (such as STATE, BLOCK, etc.) that test if identical constants appear in identical positions in the wmes, or variables (those enclosed in <>) that bind to symbols appearing in identical positions in wmes matching the condition. Production match involves finding *all* possible instantiations of the production with the the working memory. Here an instantiation refers to a collection of wmes that provide consistent bindings for the variables in the production. In this example, the wmes W1, W4 and W6 provide such an instantiation. Specifically, this instantiation is able to provide a consistent binding B1 for the variable <Z> which requires consistency checks across multiple conditions.

```
(Production P1
   (STATE  <S>  BLOCK  <Z>)
   (BLOCK  <Z>  COLOR  RED)
   (BLOCK  <Z>  VOLUME  8)
 -->
     (Some appropriate action))
```

```
W1: (STATE S1 BLOCK B1)
W2: (STATE S1 BLOCK B2)
W3: (STATE S1 BLOCK B3)

W4: (BLOCK B1 COLOR RED)
W5: (BLOCK B2 COLOR RED)

W6: (BLOCK B1 VOLUME 8)
```

Figure 1: An example production system.

Rete's operations in matching production P1 with the wmes W1,W2,..,W6 can be understood using the analogy of water-flow through pipes. As shown in Figure 2, 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, corresponding to the constant tests in that condition, e.g., the filters in the first pipe (corresponding to the first condition) check if STATE appears in the first field of a wme and BLOCK appears in the third field. As a result of these filters, only particular wmes can pass through a pipe, e.g., only W1, W2, and W3 can pass the filters of the first pipe and are seen in the bucket for pipe 1. Note that the filter BLOCK for the second and third pipes tests an identical field of the wmes. Therefore, this filter can be shared between the second and third pipes as shown in Figure 3. This illustrates the sharing optimization in Rete in a simple form. As discussed later, more complex sharing is possible in Rete.

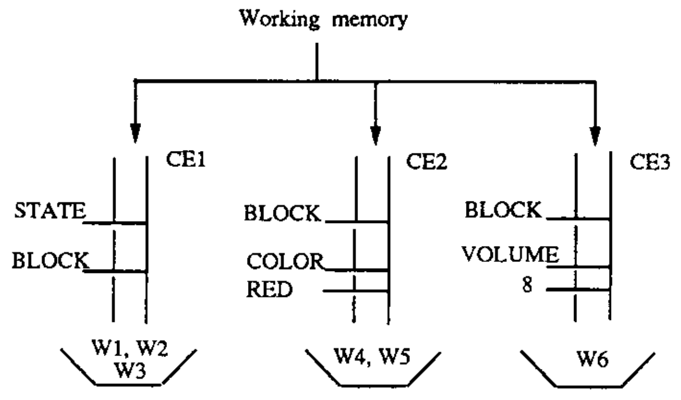


Figure 2: Explaining Rete using the analogy of water-flow through pipes. This figure shows only a portion of Rete, i.e., the match is not completed at this point.

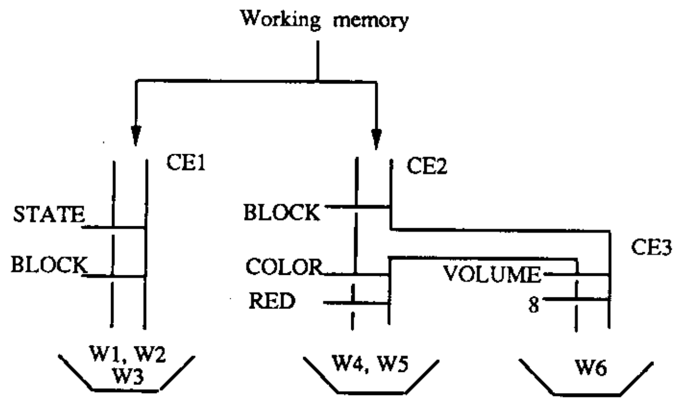


Figure 3: Sharing in Rete. This figure shows only a portion of Rete, i.e., the match is not completed at this point.

At this point, the wmes matching all conditions are known, but a combination of wmes that matches the variables in a consistent manner has not been created. This consistency checking mechanism is shown in Figure 4. The small boxes with Xs inside them and <Z> written over them check for consistency between wmes. Consider the box that joins the buckets for the first two conditions. Since <Z> appears in the fourth field of the first condition and 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) — are seen to have consistent bindings. Therefore, these pairs are stored in the following bucket. Now the second small box checks the bindings for <Z> of each of the two pairs against W6. The combination of W1,W4 and W6 is finally found to be successful and is the result of this match. This instantiation is stored in an instantiation set (also called the conflict set).

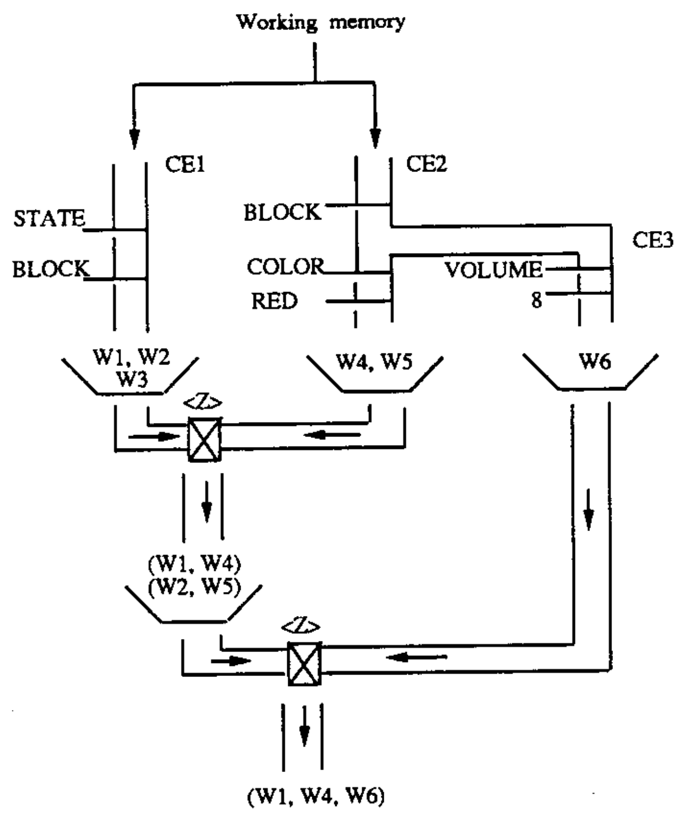


Figure 4: Consistency checks and state-saving in Rete.

Note that the buckets in Figures 2-4 realize state-saving by storing partial results of the match. In particular, consider the case where only W1,W2,..W5 are available in one recognize-act cycle. P1 will fail to match. At this point, a non-state-saving algorithm will not save partial results in buckets as is done in Rete. If W6 becomes available in a later cycle, this non-state saving algorithm will re-match W1,W2,..W5 along with W6. In contrast, Rete will store the partial results of the match with W1,W2,..W5 in its buckets. Thus, if W6 becomes available in a later cycle, only W6 will be matched. Rete will avoid re-matching W1,W2,..W5.

The penalty or cost for saving partial results is seen if a wme, say W4, is deleted. Now W4 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. The instantiation (W1, W4, W6) is also deleted from the instantiation set. In production systems, each cycle produces only a small change in the number of wmes, and hence the penalty for deletion is quite small compared to the amount of work saved [Forgy and Gupta 86, Gupta 86]. State-saving is discussed in further detail in [Miranker 87, Nayak, Gupta, and Rosenbloom 88].

In an actual implementation of Rete, the pipes discussed above are organized in a dataflow network structure as shown in Figure 5. Wmes travel down from the ROOT node. The filters that test for constants (such as STATE, BLOCK etc.) are called constant test nodes. The buckets that store individual wmes are called alpha memories. Combinations of wmes, e.g., (W1, W4), are called tokens and are stored in beta memories. Nodes that check for consistency are called and-nodes. The node that delivers and deletes instantiations is called a P-node. The portion of the network above the alpha memories, including the alpha memories, is referred to as the alpha network. It is also referred to as the *right* portion of the network. The rest of the network containing beta memories is called the *left* portion of the network. The tokens in the beta memories are called left tokens, and the wmes in alpha memories are correspondingly called right tokens.

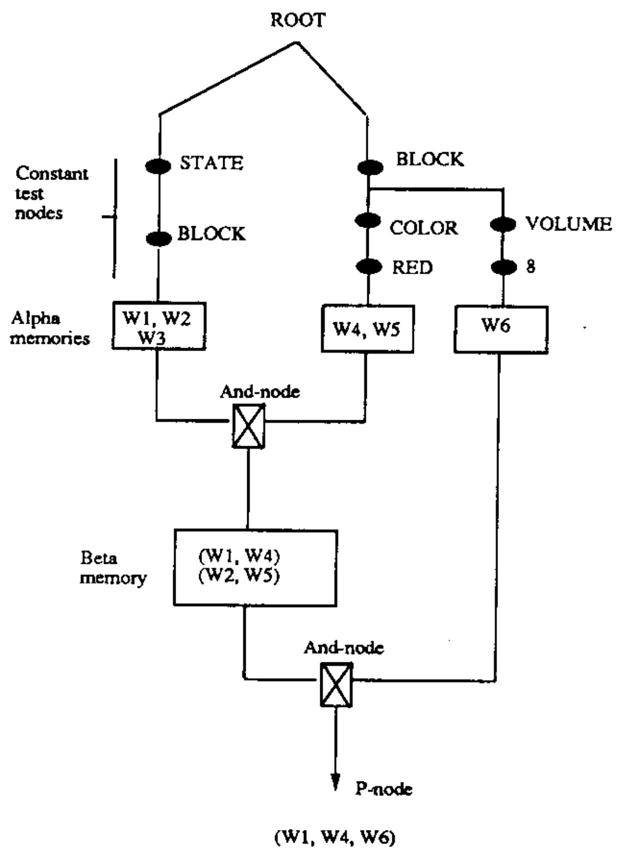


Figure 5: Rete: the real implementation is based on a dataflow network.

A real production system contains large numbers of productions, which cause the network to be quite complex, as indicated in Figure 6. Note that a single production leads to a single sequence of and-nodes and beta memories. In particular, given a single production, the output of a beta memory links to the input of a single and-node. This is the situation in Figure 5, with the output of the beta memory serving as an input to a single and-node. However, in Figure 6, there are two separate outputs from the first beta memory. This indicates the presence of a second production that shares the network up to that beta memory with the original production. The figure also shows that beta memories in a network can contain large numbers of tokens. *Combinatorics in production match refers to the combinatorial numbers of tokens in beta memories* [Tambe and Newell 88]. In the worst case, there can be O(|wmes|^{conds}) number of tokens generated in matching a single production, where |wmes| refers to the number of wmes in the production system and conds refers to the number of conditions in the left hand side of a single production.

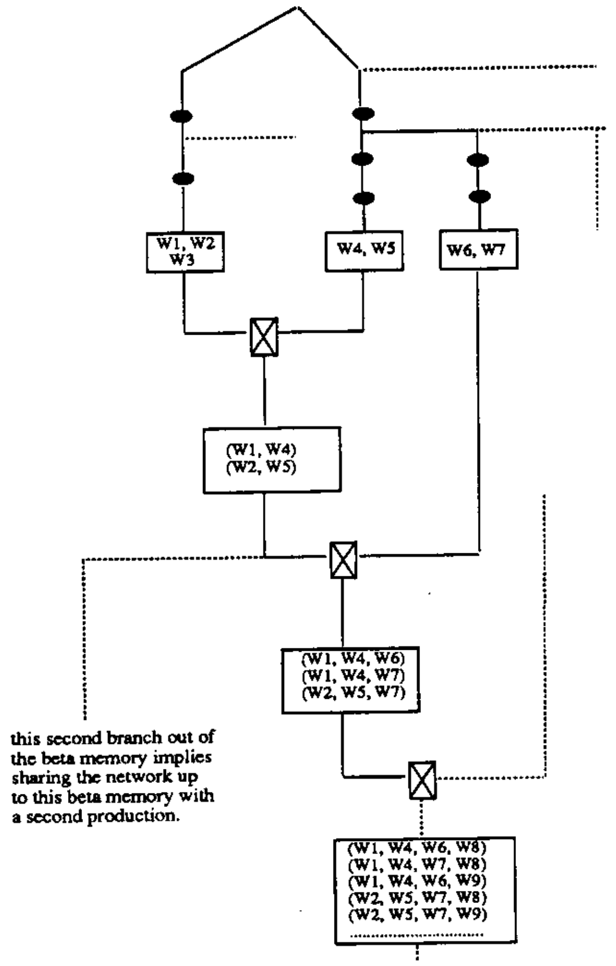


Figure 6: Combinatorics in production match refers to a combinatorial number of tokens in beta memories.

Note that the realization of alpha and beta memories is quite complex. First, given that (i) the number of tokens cannot be predicted in advance, and (ii) a beta memory can be a host to a large number of tokens, Rete uses linked lists to store the tokens. A simple linked list can be quite inefficient and more optimized implementations often rely on hash tables, as shown in Figure 7 [Gupta, et. al. 88, Gupta 86]. The Rete implementation used in this paper also relies on hash tables for implementing beta memories. Despite these hash tables, it is estimated that a majority of time in match is spent in processing beta memories. (Section 4 further illustrates this point.) A second minor point is that the tokens in the alpha/beta memories actually contain pointers to wmes — not wmes themselves. The benefit is that only a single pointer is copied during the creation of a token, rather than copying an entire wme containing 4-5 fields. There is a similar benefit during deletion. However, using pointers forces a level of indirection in accessing the fields in a token.

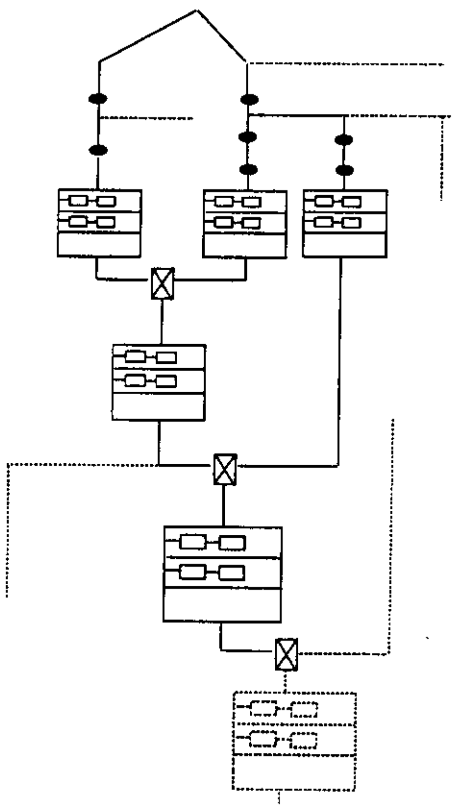


Figure 7: An optimized implementation of alpha or beta memory relies on complex hash tables.

2.2. 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 the ambiguity about which wmes can match a production's condition. 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 1, where wmes are of the form: *(STATE S1 BLOCK B1)*. This is also the representation used 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, wmes W1, W2, and W3 from Figure 1, reproduced below in Figure 8-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. In the unique-attribute representation, W1,W2 and W3 would be represented either as in 8-b or 8-c or some other alternative. Note that the alternatives in 8-b and 8-c adhere to the constraint of allowing only a unique value given fixed class, object and attribute fields.

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

Figure 8: (a) Wmes from Figure 1 that represent an unstructured set of blocks, (b) structuring the set of blocks as a list, (c) an alternative structured-set representation.

One additional representational constraint needed for unique-attributes in our representation is that the variable appearing in the object field (the second field) of a condition needs to be pre-bound, i.e., it needs to appear in an earlier condition. For instance, in production P1 in Figure 1, the variable <Z> appearing in the object field of the second and third conditions is pre-bound. Obviously, the first condition cannot adhere to this constraint of a pre-bound object field. However, the first condition is special, as we will see in Section 6.1.

Given these constraints, the unique-attribute representation guarantees that the match cost is linear in the number of conditions. In particular, beta memories can contain only a single token, and not a combinatorial number of them as is possible in an unrestricted system, i.e., a system without the unique-attribute restriction. Detailed experiments on unique-attributes have demonstrated the ability of unique-attributes to eliminate the combinatorics from the production match, and to outperform the unrestricted representation in various tasks done in Soar [Tambe, Newell and Rosenbloom 90].

Basically, the unique-attribute representation trades off some expressive power to gain efficiency in the production match. This loss in expressive power is not inconsequential. It manifests itself in two issues. First, with unique-attributes, it is difficult to encode unstructured sets in working memory. For instance, in Figure 8-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 lists as in Figure 8-b, as some special structure as in Figure 8-c, or some other alternative. This choice may not always be easy. Second, unique-attributes may cause losses in generality of learned productions. A much larger number of unique-attribute productions may be required to gain the same amount of coverage as an unrestricted production. So far, in the tasks encoded in the unique-attribute representation, these two issues have not been a major problem. For a detailed discussion of unique-attributes, the tradeoffs involved, and their applicability to different AI systems, please see [Tambe 91, Tambe and Rosenbloom 90, Tambe, Newell and Rosenbloom 90].

3. Introducing Uni-Rete

Figure 9 illustrates the implication of the unique-attribute representation for Rete. Each beta memory contains only a single token. As discussed earlier, with unique-attributes, a beta memory can contain a maximum of one token. However, despite this single token, Rete goes through the process of storing it as in Figure 7, and incurs a large memory management overhead. Note that the left-most alpha memory also contains only a single wme.

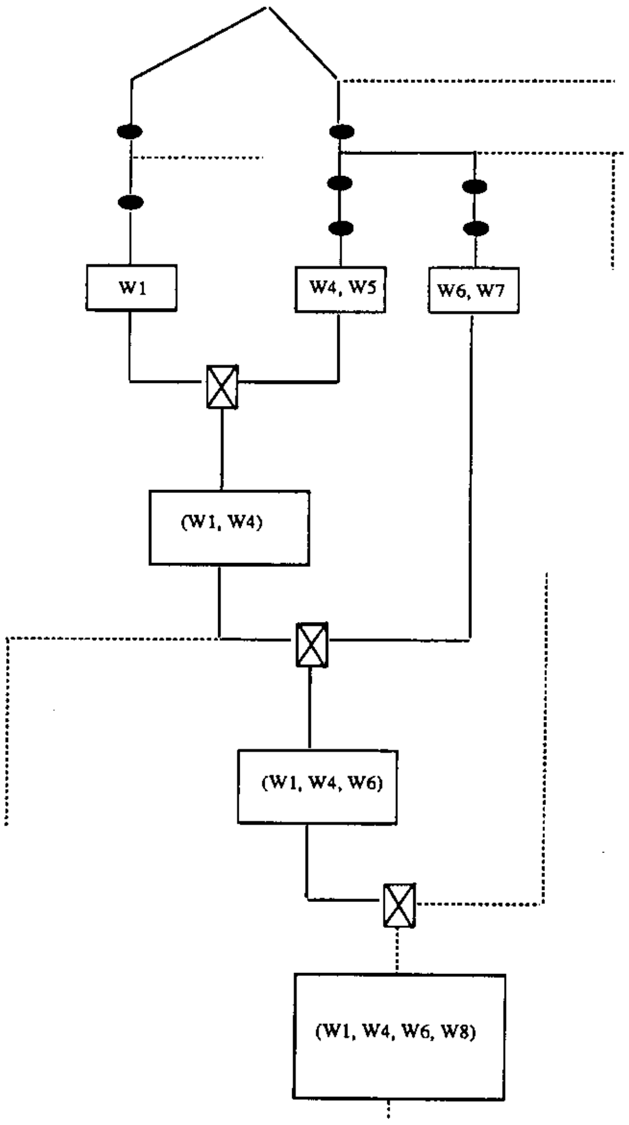


Figure 9: With unique-attributes, each beta memory can contain a maximum of one token.

The intuition behind Uni-Rete is to exploit 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 beta memory, the rest being implicit from the beta memories preceding the current beta memory. Thus, instead of storing tokens as shown in Figure 9, they can be stored as shown in Figure 10. Consider the beta memory containing the token (W1, W4) in Figure 9. In the corresponding beta memory in Figure 10, only W4 is stored. Since the preceding alpha memory contains only a single wme (W1), the token (W1, W4) is implicit in the two memories. Similarly, instead of storing the entire token (W1, W4, W6) as in Figure 9, only W6 is stored in Figure 10, the remaining portion of that token being obvious from the contents of 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 (see Figure 7). This single location per wme actually stores a pointer to a wme and will be referred to as a wme-pointer location. In Figure 10, 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 large numbers of 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.

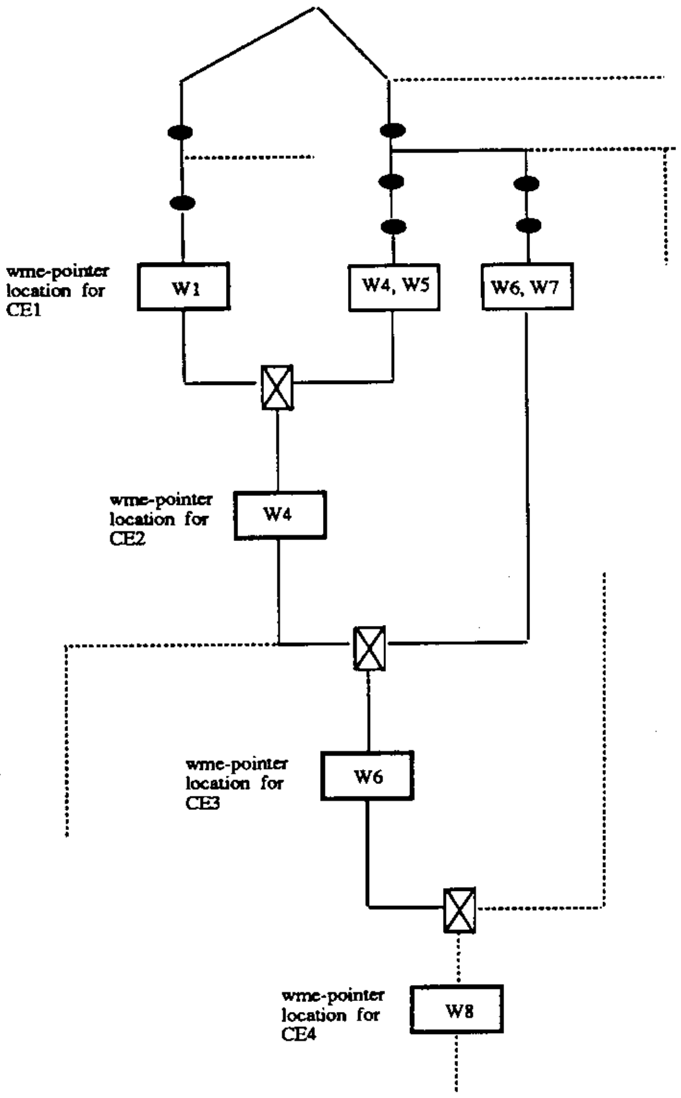

wme-pointer location for CE1
W1
W4, W5
W6, W7
wme-pointer location for CE2
W4
wme-pointer location for CE3
W6
wme-pointer location for CE4
W8

Figure 10: Tokens are stored implicitly in Uni-Rete.

The operation of the Uni-Rete algorithm is now described using the two working-memory operations: add and delete. We will use the network of Figure 10 as an example in this description. This network is generated exactly as in Rete, except that the beta-memory structure of Rete is replaced by a single wme-pointer location. Analogously, the alpha memory for the first condition is replaced by a wme-pointer location.

When a wme is added to the system, the following steps are taken in Uni-Rete:

1. *Perform constant tests and store the wme in an alpha memory*: This portion of Uni-Rete corresponds to Rete. The wme is stored in the alpha memory exactly as in Rete. For instance, suppose the wme W6 is added. It is stored in the second alpha memory as shown in Figure 10. The alpha memory in our implementation is realized using a hash-table.

2. *Check the match for the previous condition*: Check if the wme-pointer corresponding to the previous condition is null. If it is null, there is no match for the previous condition, and hence no action is taken. For instance, since W6 matches the third condition, we check the wme-pointer location for the second condition (CE2). This location is non-null (it contains W4) and hence we proceed to the third step.

3. *Execute the consistency tests with previous conditions*: If the previous condition has a match, i.e., its wme-pointer location is non-null, then execute the required consistency tests between the current wme and the previous wme(s). For instance, in our example of adding W6, supposing some variable <Y> appears in the second and third conditions. Then check the consistency of the bindings for <Y> between W4 and W6.

4. *Store a pointer to the working memory element*: If the test in step 3 succeeds, store a pointer to the new working memory element in the corresponding location. In our example of adding W6, store a pointer to W6 in the wme-pointer location for condition 3, i.e., condition 3 is now successfully matched. If the test does not succeed, do not take any further action.

5. *Match the next condition*: For this step, check the alpha memory of the next condition for wmes that can potentially match. If the test with the next condition succeeds, then a pointer to the successfully matching wme is placed in the wme-pointer location of that condition. Note that there can be only one successfully matching wme, since this is a unique-attribute system.

   In our example, test W6 against the alpha memory for condition 4. If the test succeeds, assign a wme-pointer to the wme-pointer location for condition 4. Continue with step 5 until the next condition is not matched.

The production successfully matches if there is an entry in the wme-pointer location of the last condition.

When a working memory element is deleted with Uni-Rete, the following steps are taken:

1. *Delete the working memory element from the alpha memory*: The deleted wme goes through all the constant tests. On reaching the alpha memory it deletes the wme from that alpha memory. For instance, when W6 is deleted, W6 is found to pass the required constant tests, and hence W6 is removed from the alpha memory of the third condition.

2. *Locate the wme-pointer that points to the deleted wme*: Check if the wme-pointer location of the matched condition contains a pointer to the wme. In our example of deleting W6, look in the wme-pointer location entry for the third condition and check if the wme-pointer points to W6. This implies that W6 had earlier successfully matched the third condition. If it does, proceed, else do nothing.

3. *Nullify located pointer*: The pointer that points to the deleted wme is erased, i.e., a null is entered in that location. In our example of deleting W6, the wme-pointer location of the third condition will be nullified. This implies that there is no match for the third condition.

4. *Proceed in the match, nullifying all other pointers*: After nullifying the value at a particular condition's wme-pointer location, proceed to all successor wme-pointer locations, entering null into all of them. Essentially the match for all the later conditions depends on the match for the current condition. Therefore, if the current condition no longer matches a wme, the later conditions also cannot match. In our example of deleting W6, after entering null into

the third condition's wme-pointer location, proceed to the following wme-pointer location for the fourth condition (containing W8), and enter null into it. Then proceed to the fifth condition, and others following that.

The extension of Uni-Rete to handle negated conditions is straightforward. A negated condition is one that disallows a production from matching if a wme matches it consistently. In Uni-Rete, each negated condition requires an additional flag. This flag is set to zero if a negated condition matches and to 1 other wise. If a wme matching a negated condition is added to the system, then it follows the normal procedure for adding wmes, except that step 5 is replaced by step 4 of deleting wmes — since a successfully matched negated condition does not allow the following conditions to match. It also sets the flag to zero. If a wme matching a negated condition is deleted, then it follows the normal procedure for deleting wmes except that step 4 is replaced by step 5 of adding wmes — since a non-matching negated condition allows the following conditions to match. It also sets the flag to 1. Two other modifications are also required: when a wme is added to a condition immediately following a negated condition, it reverses step 3 of the addition procedure. It only proceeds in the match if the negated condition is not matched, i.e., the flag for the negated condition is set to 1. Similarly when a condition immediately preceding a negated condition executes step 5 for adding a wme, it reverses step 5 — it proceeds in the match only if it cannot find a matching wme for the negated condition and sets the flag to 1.

Figures 11 and 12 provide a more formal description of this algorithm. The steps in these figures correspond to the steps in the informal description provided above. Note that the algorithm in Figures 11 and 12 amounts to implementing an interpreter for Uni-Rete. Our actual implementation is a compiled version of this algorithm based on the OPS83 software technology [Forgy 84]. In the compiled version, pointer chasing is completely avoided. For instance, in step 3 of the addition procedure, consistency tests T_i do not require a traversal of the Uni-Rete data-structure to access the contents of the wme-pointer locations involved. Instead, the addresses of the wme-pointer locations, which are known at compile time, are directly used for accessing their contents.

As noted earlier, the absence of left tokens is the major optimization in Uni-Rete that allows it to perform better than 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 all the previous wmes into the current token. For instance, in Figure 9, Rete would require copying 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 10, Uni-Rete will not require such copying. Third, since left tokens are absent, 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 requires following a course symmetrical to its addition, and removing tokens containing the deleted wme. 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. For instance, in Uni-Rete, in step 5 of the addition procedure, only a single match is possible. Therefore, the loop searching for matches in the alpha memory can terminate as soon as the first successful match is found. In contrast, in the corresponding step in Rete, an unpredictable number of tokens can potentially be generated. Hence, that loop cannot terminate after the first successful match. Further minor simplifications are also possible in Uni-Rete: (i) the instantiation processing is simplified since there is only a single instantiation and (ii) the implementation of negated conditions is also simplified.

Uni-Rete: Procedure for addition of a wme

```
/* Comments: Let $W_i$ be the wme added. Let the production system contain a single production P
 * with conditions $C_1, C_2, .. C_i .., C_N$, such that $W_i$ passes the constant tests of $C_i$. For a condition
 * $C_i$: alpha-memory($C_i$) indicates its alpha memory, wme-pointer($C_i$) indicates its wme-pointer
 * location, $T_i$ indicate its consistency tests. Similarly, instantiation(P) refers to the instantiation of P.
 */

1. Perform constant tests and store $W_i$ in the alpha-memory($C_i$);

2. If $C_{i-1}$ is non-negated then
        if (wme-pointer($C_{i-1}$) <> NULL) goto step 3 else return;
   If $C_{i-1}$ is negated then
        if (flag($C_{i-1}$) = 1) goto step 3 else return;

3. If consistency tests $T_i$ succeed using $W_i$ then
        goto step 4 else return;

4. If $C_i$ is non-negated then
        wme-pointer($C_i$) := pointer to $W_i$;
        j := i+1;
        goto step 5;
   If $C_i$ is negated then
        wme-pointer($C_i$) := pointer to $W_i$;
        flag($C_i$) := 0;
        for (k = i+1 to N) wme-pointer($C_k$) := NULL;
        if P is instantiated, remove instantiation(P) from conflict set;
        return;

5. If $C_j$ is non-negated then
        For each wme $W_k$ in alpha-memory($C_j$)
            if consistency tests $T_j$ succeed using some $W_k$ then
                wme-pointer($C_j$) := pointer to $W_k$;
                if j = N, instantiate production P
                    else
                        j := j+1;
                        goto step 5;
                return;
   If $C_j$ is negated then
        For all wmes $W_k$ in alpha-memory($C_j$)
            if consistency tests $T_j$ succeed using some $W_k$ then
                wme-pointer($C_j$) := pointer to $W_k$;
                flag($C_j$) := 0;
                return;
        if no wme $W_k$ succeeds in consistency tests $T_j$
            flag($C_j$) := 1;
            if j = N, instantiate production P
                else
                    j := j+1;
                    goto step 5;
            return;
```

Figure 11: The Uni-Rete match algorithm: procedure for addition of a wme.

Uni-Rete: Procedure for deletion of a wme

```
/* Comments: Let $W_i$ be the wme added. Let the production system contain a single production P
 * with conditions $C_1, C_2, ..C_i.., C_N$, such that $W_i$ passes the constant tests of $C_i$. For a condition
 * $C_i$: alpha-memory($C_i$) indicates its alpha memory, wme-pointer($C_i$) indicates its wme-pointer
 * location, $T_i$ indicate its consistency tests. Similarly, instantiation(P) refers to the instantiation of P.
 */

1. Perform constant tests and delete $W_i$ from the alpha-memory($C_i$);

2. If wme-pointer($C_i$) = pointer to $W_i$
        goto step 3 else return;

3. wme-pointer($C_i$) := NULL;

4. If $C_i$ is non-negated
        for (k = i+1 to N) wme-pointer($C_k$) := NULL;
        if P is instantiated, remove instantiation(P) from conflict set;
        return;
   If $C_i$ is negated
        flag($C_i$) := 1;
        j := i+1;
        goto step 5;

5. If $C_j$ is non-negated then
        For each wme $W_k$ in alpha-memory($C_j$)
           if consistency tests $T_j$ succeed using some $W_k$ then
              wme-pointer($C_j$) := pointer to $W_k$;
              if j = N, instantiate production P
                 else
                    j := j+1;
                    goto step 5;
              return;
   If $C_j$ is negated then
        For all wmes $W_k$ in alpha-memory($C_j$)
           if consistency tests $T_j$ succeed using some $W_k$ then
              wme-pointer($C_j$) := pointer to $W_k$;
              flag($C_j$) := 0;
              return;
        if no wme $W_k$ succeeds in consistency tests $T_j$
           flag($C_j$) := 1;
           if j = N, instantiate production P
              else
                 j := j+1;
                 goto step 5;
           return;
```

Figure 12: The Uni-Rete match algorithm: procedure for deletion of a wme.

This discussion shows that while the processing of left tokens (in beta memories) is heavily optimized in Uni-Rete, the processing of right tokens (in alpha memories) 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.

4. Experimental Methodology and Initial 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 [Laird, Newell, and Rosenbloom 87]. Unfortunately, while Soar is implemented in LISP, our Rete and Uni-Rete implementations are built in C. Our experience with Soar/PSM-E [Tambe, et al. 88] indicates that integrating such C-based matchers into the existing LISP-based Soar system would be quite difficult. Therefore, we obtained detailed traces of Soar's wme changes for running specific problems. This sequence of wme changes was then input to Uni-Rete and Rete. Given this sequence as an input, the match activity in the C-based Rete and Uni-Rete matchers is identical to Soar's LISP-based matcher.

Figure 13 illustrates the benchmarks used, the number of productions in the benchmarks, the number of left (beta memory) and right (alpha memory) tokens and the ratio of right tokens to left tokens. As discussed earlier, this ratio of right tokens to left tokens is important in determining speedups[1]. 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.

One interesting issue here is the selection of appropriate benchmarks. The unique-attribute representation is a fairly new one. While a large number of tasks have been developed in Soar [Rosenbloom, Laird, Newell, and McCarl 91], only 18-20 of those are encoded in the unique-attribute representation. It is difficult to partition these 18-20 tasks so as to pick out representative tasks as benchmarks for our experiments. We have focused on the right-to-left token ratio and the number of productions as two possible dimensions for such a partition, and selected benchmarks that show some variability along these dimensions. Among the benchmarks, MAX is an artificial set of productions used to illustrate the maximum amount of speedup. In particular, MAX has a very large number of left tokens compared to right tokens (the ratio of right to left is actually very close to zero). *The speedup in MAX is 15.5 fold.* Figure 13 indicates that MFS* has a very high ratio and should show the lowest speedup[2].

Figure 14 shows the run-times for Rete and Uni-Rete for the different benchmarks. The x-axis indicates the different benchmarks. The numbers below the names of the benchmarks indicate their right-to-left token ratios. The y-axis shows the total match time in seconds. This figure shows that Uni-Rete achieves up to 8 fold speedup over Rete. Also, as expected, MFS* achieves the lowest speedup of 3.2

[1]In these measurements, the alpha memory tokens for the leftmost condition are counted as left tokens. This causes a very small distortion in the right-to-left token ratio.

[2]MFS* is a small portion of a large production system called MFS (with approx. 700 productions), being developed at Carnegie Mellon University [Li, Krishnan and Steier 91]. It is possible that our focus on a small portion of MFS may have caused its right-to-left ratio to be higher than the other tasks. For a description of the other benchmarks, please see [Tambe 91].

TASK NAME	NUMBER PRODS	RETE TOKENS		RATIO RIGHT/LEFT
		LEFT	RIGHT	
MAX	20	704880	19	~ 0
EIGHT1	52	3789	559	0.14
EIGHT2	62	7478	809	0.10
GRID	60	2421	361	0.14
BLOCKS	181	2577	385	0.14
MFS *	59	2575	1209	0.46

Figure 13: The benchmarks used for comparing Rete and Uni-Rete.

fold. These measurements were done on a VAX 11/780[3].

5. Bilinearization

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. In a bilinear network, a new condition in a production may join the network at the end of some particular sequence of previous conditions, not necessarily all of them. Figure 15 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. Consider a case where there is a consistency test between CE4 and CE6. In the linear network, CE4 precedes CE6, and the required consistency check can be performed when CE6 is joined in. In the bilinear network shown in Figure 15, CE4 does not precede CE6, and the consistency check cannot be done. Hence, the super-p-node is required to perform the remaining consistency checks. Additionally, the super-p-node also performs the function of the p-node in Rete: generating and deleting instantiations. In general, a single super-p-node combines all the bilinear branches of a single production.

[3]This slow machine was chosen for our experiments partly because the overheads in our timing tool precluded the use of faster machines in our environment. In particular, given the highly optimized nature of Uni-Rete, measuring its execution time became quite difficult on the faster machines. Additionally, the VAX 11/780 machine was very lightly loaded and easily accessible.

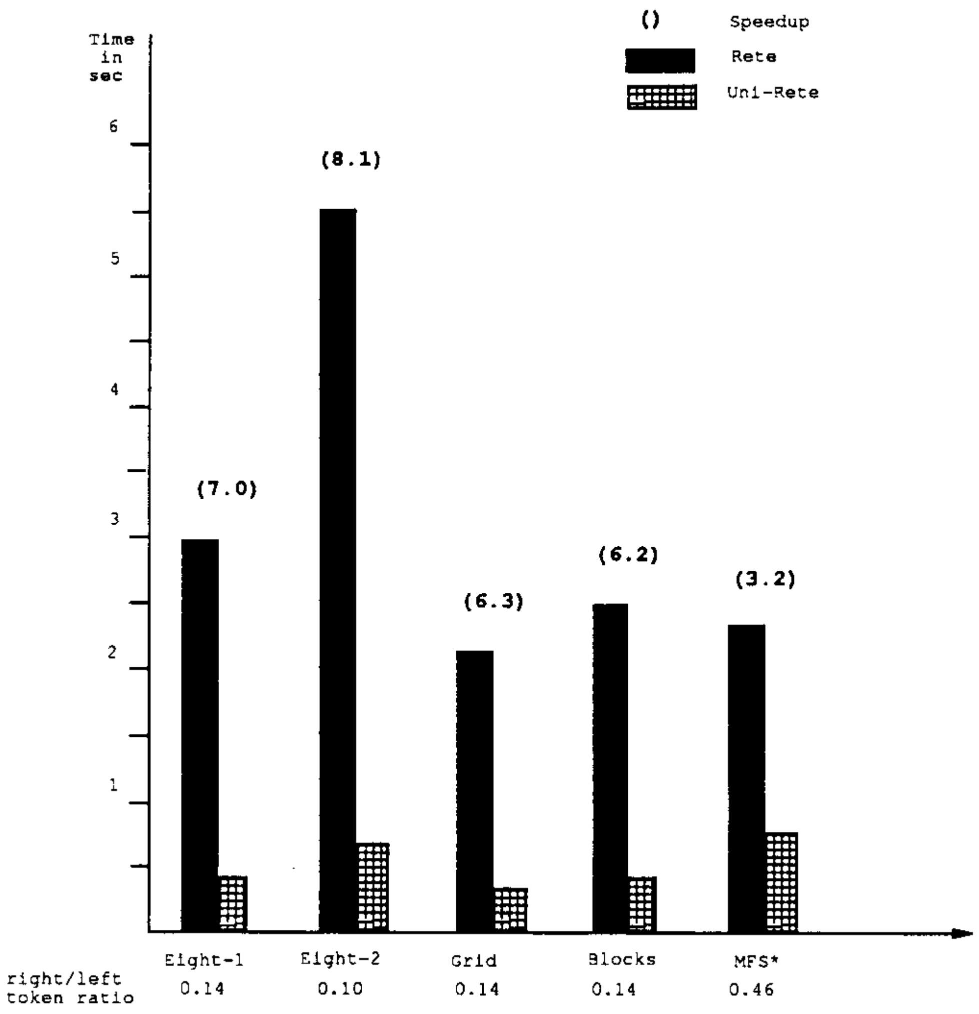


Figure 14: Speedup of Uni-Rete over Rete.

Figure 16 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 sequence of conditions. The sequence (CE1, CE2) in P1 also appears in P2 — hence the linear Uni-Rete shares the two conditions across P1 and P2. However, the sequence of CE5 and CE6 in P1 cannot be shared with CE3 and CE4 in P2. This gives rise to the linear Uni-Rete network shown in Figure 16, with a total of eight wme-pointer locations in it.

In bilinear Uni-Rete, sharing is partially, but not entirely dependent on a common sequence of conditions. In particular, as shown in Figure 16, 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, the number of wme-pointer locations goes down from eight to six.

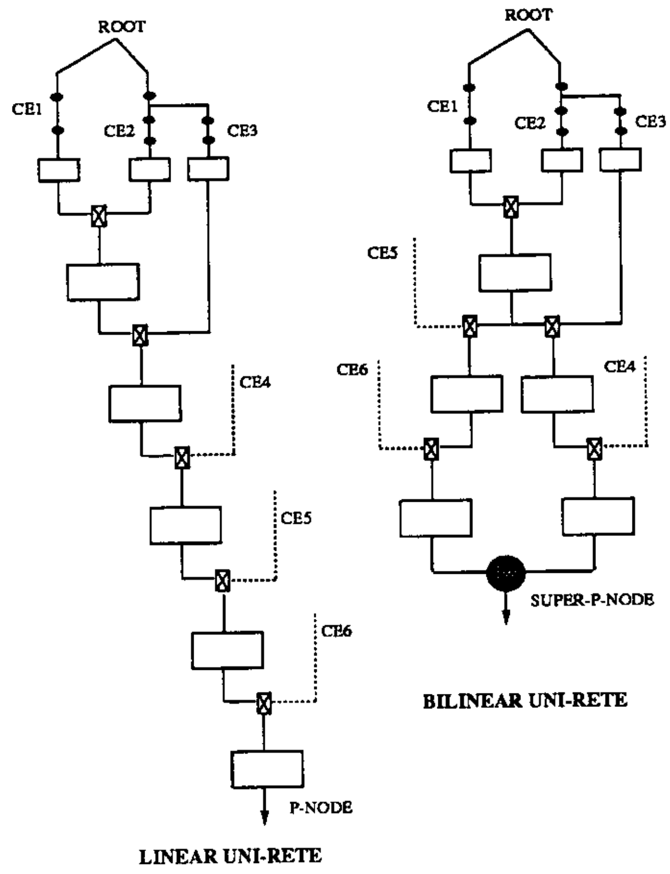


Figure 15: Demonstrating bilinear organization of the Uni-Rete network.

```
(P1                     (P2
    CE1                     CE1
    CE2                     CE2
    CE5                     CE3
    CE6                     CE4
-->                         CE5
    (ACTION))               CE6
                        -->
                            (ACTION))
```

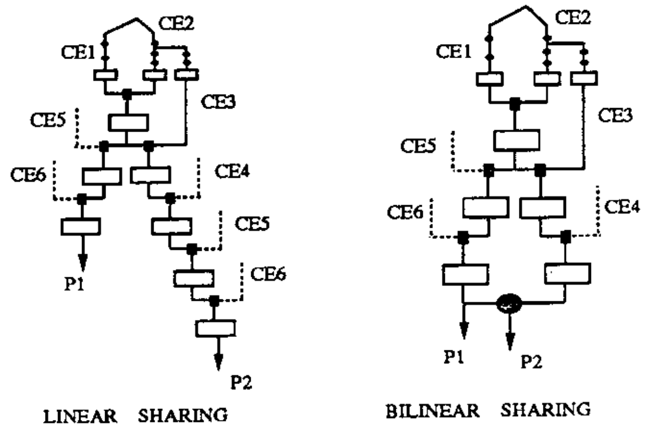


Figure 16: Demonstrating the increase in sharing due to the bilinear organization of the Uni-Rete network.

Thus, bilinear networks can lead to improved sharing. The savings due to sharing can be quite dramatic (as seen later) for larger systems. Intuitively, maximizing the amount of sharing will lead to maximum speedups. The amount of sharing can be maximized by creating as many bilinear branches per production as possible. In the extreme, there can be a single branch for each condition, which will allow a maximum amount of sharing.

However, generating maximally bilinear networks will not necessarily lead to maximum speedups. This is because increased bilinearization leads to more activity in the super-p-nodes, which increases their execution time. The super-p-nodes perform consistency checks between bilinear branches. With a large number of bilinear branches per production, the super-p-nodes perform a larger number of consistency checks, which increases their overhead. This is particularly true for our current super-p-node implementation: as described in Section 6.1, our current super-p-node implementation is quite unoptimized.

We have written a simple program called *bilin* for bilinearizing a production system. It takes two conflicting constraints into account: (i) increased bilinearization leads to increased sharing and hence a reduction in execution time, and (ii) increased bilinearization leads to increased activity in the super-p-nodes and hence an increase in execution time. Section 5.1 discusses *bilin*. Section 5.2 discusses the results of bilinearization due to *bilin*.

5.1. *Bilin*: Creating Bilinear Uni-Rete Networks

Bilin bilinearizes productions to improve their sharing while simultaneously attempting to reduce their super-p-node overhead. The input to *bilin* is a file containing all the productions in a production system. *Bilin* reads the productions one at a time and, for each, outputs a bilinearized version, i.e., a production split according to the condition chains in it. Figure 17 shows the example of production P2 from Figure 16. Recall that P2 was split to create two chains of conditions: (CE1, CE2, CE3, CE4) and (CE1, CE2, CE5, CE6). Figure 17 shows the output of *bilin* that creates these two chains. The @@@ sign indicates the end of a bilinear chain.

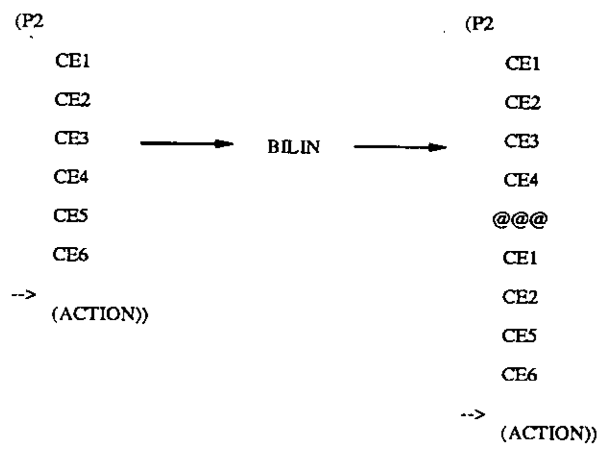


Figure 17: Demonstrating the operation of bilin.

Note that *bilin* has to adhere to the constraint of pre-bound variables in the object field of conditions.

Bilin uses this constraint in bilinearizing by first building a graph of the conditions. In this graph, each condition is a node. If CE1 and CE2 are two nodes in the graph, then a directed arc joins CE1 and CE2 if the variable in the value field (fourth field) of CE1 appears in the object field of CE2. Figure 18 shows a production P4 and its graph.

```
(P4
   (GOAL  <G>  STATE  <S>)            CE1
   (STATE  <S>  BLOCK  <B1>)          CE2
   (BLOCK  <B1>  COLOR  RED)          CE3
   (BLOCK  <B1>  VOLUME <V>)          CE4
   (STATE  <S>  BLOCK  <B2>)          CE5
   (BLOCK  <B2>  COLOR  GREEN)        CE6
   (BLOCK  <B2>  VOLUME  <V>)         CE7
   (VOLUME  <V> QUANTITY  8)          CE8
-->
   (ACTION))
```

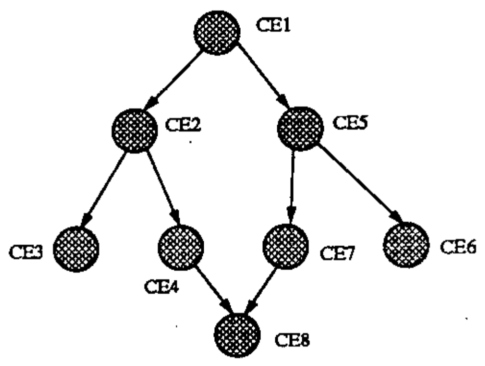


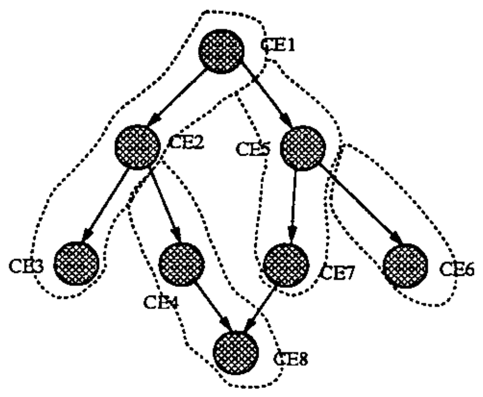
Figure 18: The first step in *bilin*: build a production graph.

After creating this graph, *bilin* starts from the root and traverses it in a depth-first manner. Each time it reaches a leaf node, it outputs the path from the root to the leaf as a branch of the bilinear network. The bilinear network generated in this manner for P4 is shown in Figure 19. If there is more than one root, *bilin* performs identically for each root.

However, in order to avoid excessive bilinearization, *bilin* uses the following heuristics:

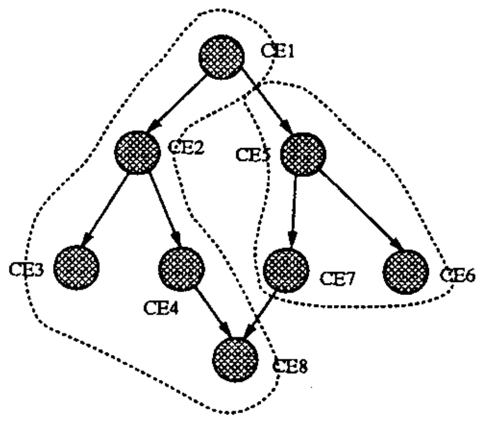
- It bilinearizes a production only if the production contains more than a threshold number of conditions (C_t). Thus, if number of conditions in a production is less than C_t, *bilin* simply outputs the production without bilinearizing. C_t is set by the user. In our experiments, C_t was set to 6.
- *Bilin* uses a depth parameter (D) to limit the amount of bilinearizing. This parameter is also user-supplied. *Bilin* cuts off its depth first traversal if its depth in the condition graph exceeds D. All the conditions below depth D are grouped together into 1 bilinear branch. Thus, given a condition graph as shown in Figure 20, and D of 1, *bilin* will create the branches as shown.

Among our six benchmarks, EIGHT1 and EIGHT2 were bilinearized with D set to 3. The rest were bilinearized with D set to 2. This level of bilinearization was selected after a couple of trial and error experiments. In particular, we did not test the benchmarks for different values of D to select the best bilinearizing level. However, these experiments did confirm that with D set to a very large value the super-p-node overhead caused the system to be somewhat slower than the linear Uni-Rete system. Actually, the parameter D assumes importance because the current implementation of super-p-nodes is unoptimized. As we continue to optimize super-p-nodes, we expect that the parameter D will not retain its importance.



```
(P4
   CE1
   CE2
   CE3
   @@@
   CE1
   CE2
   CE4
   CE8
   @@@
   CE1
   CE5
   CE7
   @@@
   CE1
   CE5
   CE6
-->
   (ACTION))
```

Figure 19: The second step in *bilin*: output bilinear chains.



```
(P4
   CE1
   CE2
   CE3
   CE4
   CE8
   @@@
   CE1
   CE5
   CE6
   CE7
-->
   (ACTION))
```

Figure 20: Controlling the amount of bilinearization by setting the depth parameter D to 1.

5.2. Results

Figure 21 shows the results of bilinearization due to *bilin*. The figure shows the number of wme-pointer locations without any sharing, with linear sharing and with bilinear sharing for our six benchmarks. The numbers in parentheses show the ratio of the number of locations without sharing to number of locations with sharing. Thus, in BLOCKS, the number of wme-pointer locations are reduced by a factor of 1.68 with linear sharing, and by a factor of 5.84 with bilinear sharing. In fact, this reduction from 1.68 to 5.84 is the maximum among this set of benchmarks. The increase in sharing with bilinear networks in MFS* and GRID is quite low compared to the increase in BLOCKS, EIGHT1 and EIGHT2. Since bilinearizing can be increased arbitrarily, MAX was excluded from bilinearizing.

TASK NAME	NUMBER PRODS	NUMBER WM-LOC NO SHARING	NUMBER WM-LOC LINEAR SHARING	NUMBER WM-LOC BILINEAR SHARING
MAX	20	360	360	--
EIGHT1	52	726	489 *(1.48)*	206 *(3.52)*
EIGHT2	62	960	634 *(1.51)*	225 *(4.26)*
GRID	60	1219	592 *(2.05)*	297 *(3.11)*
BLOCKS	181	2284	1356 *(1.68)*	391 *(5.84)*
MFS *	59	358	255 *(1.40)*	174 *(2.05)*

Figure 21: The impact of bilinear networks on sharing in the five benchmarks. WM-LOC refers to wme-pointer locations. The numbers in parentheses show the ratio of the number of locations without sharing to number of locations with sharing.

Figure 22 shows the speedups from bilinearization. The x-axis indicates the different benchmarks. The y-axis shows the total match time in seconds. The match time for three systems are shown: Rete, linear Uni-Rete and bilinear Uni-Rete. The match times for Rete and linear Uni-Rete are those from Figure 14. The maximum increase in speedup from linear to bilinear Uni-Rete is achieved in BLOCKS. This is expected since the bilinear network in BLOCKS caused a maximum increase in sharing. The increase in speedups in EIGHT1 and EIGHT2 are also quite substantial. As expected, MFS* and GRID lag behind.

6. Discussion

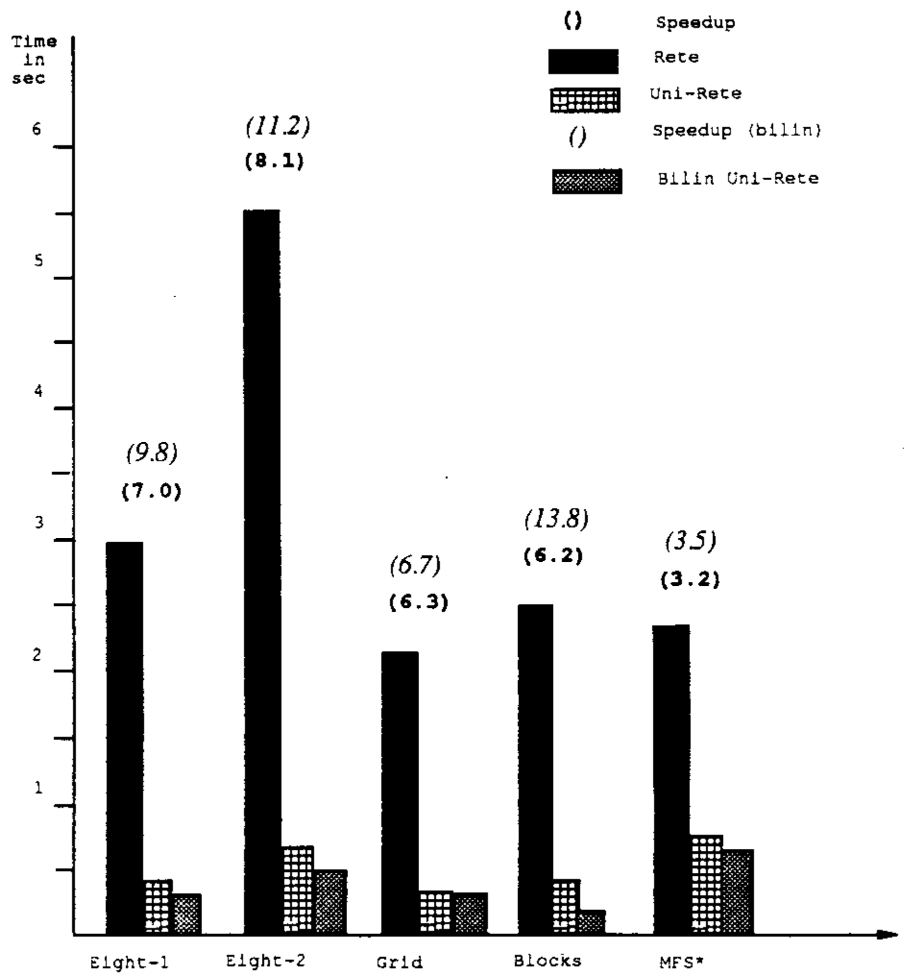


Figure 22: Speedups with bilinear Rete-networks.

6.1. Uni-Rete: Existing Problems

Our existing Uni-Rete implementation is not without some limitations and lack of optimizations. An important limitation of the current implementation is its assumption that the first condition only matches a single wme. In Soar, this single wme is the current goal of the problem-solver. All our benchmarks were such single goal systems.

However, Soar can generate subgoals. In the presence of subgoals, more than a single wme can match the first condition. Each such wme actually corresponds to a new subgoal. With subgoaling, beta-memories can contain more than a single token. The unique-attribute assumption of a single token per beta memory is still valid, but on a per subgoal basis. Subgoaling in Soar can be supported using the technique shown in Figure 23. The figure shows each wme-pointer location replaced by an array. Each index in the array refers to a single subgoal. Thus, the first index refers to the first goal, the second index to the second (sub)goal and so on. Now, the Nth index is used for all match activity related to the Nth subgoal. If the production system has only a single goal, then Uni-Rete executes exactly as described in Section 3, except that the wme-pointer locations are indexed by the first index. If the second (sub)goal is generated, Uni-Rete will use the second index for all the match activity for the second subgoal. With this scheme, the advantages of Uni-Rete (of not creating tokens) is retained; however, a very small indexing overhead is introduced. Note that given a new wme, Uni-Rete does not require scanning the entire wme-pointer location array to identify its matching subgoal index. In particular, for each wme, Soar identifies its owner subgoal. This permits Uni-Rete to immediately identify the subgoal index for that wme.

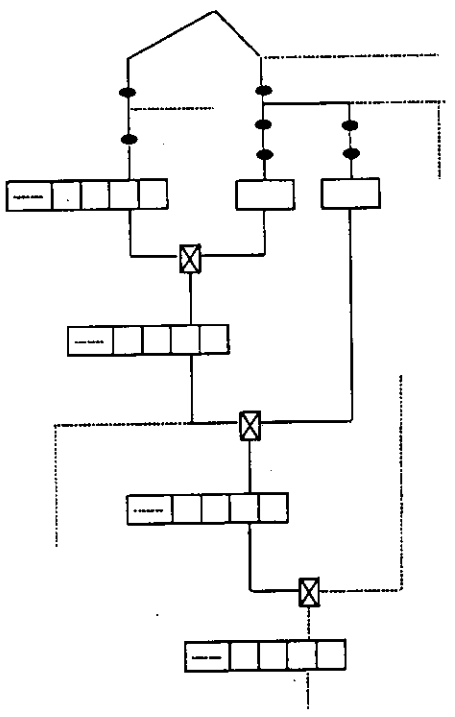


Figure 23: Supporting Soar's subgoaling in Uni-Rete.

Typically, about five to ten subgoals are seen in Soar systems. For such a low number of subgoals, the scheme described in Figure 23 appears appropriate. In terms of memory usage, this scheme does not cost much. Consider the GRID task which contains approximately 300 wme-pointer locations in the bilinear format. With 10 subgoals, the total memory used in this scheme is only 12Kbytes (assuming 4 bytes per wme-pointer location). Large Soar systems contain approximately 1000 productions, i.e., roughly 20 times as many productions as in the GRID task. If we assume 20 times as many wme-pointer locations for such systems, and assume an unrealisticly high number of subgoals (100), the total memory used is still only 2.4 Mbytes.

One interesting issue in this scheme is determining the size of the wme-pointer location array. The size of this array is dependent on the maximum number of subgoals generated during problem-solving. This number cannot be pre-determined by the Uni-Rete compiler. To address this issue, the Uni-Rete system may start with wme-pointer location arrays with a large default size of say 50 or 100. In the unlikely case that this default value is exceeded, a linked list of wme-pointer locations can be allocated and appended to the end of the wme-pointer location array. Alternatively, the user may provide a hint to the system about the maximum depth of subgoaling in a particular task. However, subgoaling needs to be investigated further before commiting to such schemes.

Some important optimizations are missing from Uni-Rete as well:

- The super-p-node implementation is quite sub-optimal. One way to understand this is to view each branch of the bilinear network as a single production condition (called super-conditions). The super-p-node in effect then combines these different super-conditions. Thus, the function of the super-p-node is analogous to the function of beta portion of the Rete/Uni-Rete network which combines different conditions. However, unlike the beta portion of the Rete network, super-p-nodes do not exploit state saving or sharing optimizations. Thus, the super-p-nodes will re-compute the match for the super-conditions on every cycle. Introducing state-saving and sharing in super-p-nodes is a topic for future work.
- In step 4 of the procedure for deleting a wme, the system currently nullifies all the successor locations. Many of these locations are already null. However, testing if a location is null before nullifying it seems to add unnecessary overhead.
- The garbage collection routines, routines that execute the constant tests in the network, and some others, consume a bigger percentage of time in Uni-Rete. The absolute execution time for these routines is the same in Uni-Rete and Rete. In Rete, this time is very small compared to the total match time. However, given the optimized nature of Uni-Rete, these routines need to be optimized as well.

Overall, Uni-Rete is a new algorithm. The Rete match algorithm was developed in mid-1970s, and over the last 10-12 years, many optimizations have been added to it [Acharya and Tambe 89b]. We expect a somewhat similar course of development for Uni-Rete: as we continue to understand it better, we should be able to optimize it further.

6.2. Is This a Fair Comparison?

A comparison of the form described in this paper leads to questions of fairness of this comparison. In particular, how optimized was the Rete system used in our comparison? Clearly, if we started with a highly unoptimized implementation of Rete, the speedups reported here would not be very meaningful.

The Rete system used in this paper originated from the CParaOPS5 system, a C-based parallel OPS5 implementation [Gupta, et. al. 88, Kalp, et. al. 88]. CParaOPS5 is based on the highly optimized OPS83 software technology [Forgy 84]. For OPS5 programs, this system is only a factor of 1.5 to 2-fold slower than an optimized uni-processor based Treat implementation [Miranker and Lofaso 91]. Note that for OPS5 programs, Treat was earlier shown to outperform Rete [Miranker 87].

We removed overheads of parallelism from CParaOPS5, which gave us a speedup of 1.5 or more over a set of three benchmarks. This new system, called Old-Rete, was further optimized to provide us with the Rete system used in our comparison with Uni-Rete. Figure 24 indicates the speedups achieved for our six benchmarks by Rete over Old-Rete. Thus, the Rete implementation is a factor of 2.5 or so faster than the original CParaOPS5 implementation. Note that the Uni-Rete system was also developed from the Old-Rete system. Thus, both the Uni-Rete and Rete systems have the same starting points. Furthermore, as we optimized Uni-Rete, the applicable optimizations were applied to Old-Rete (giving rise to Rete). Here, applicability refers to the ability of that optimization to speed up a program in the unrestricted representation — Rete is to be optimized for the unrestricted representation.

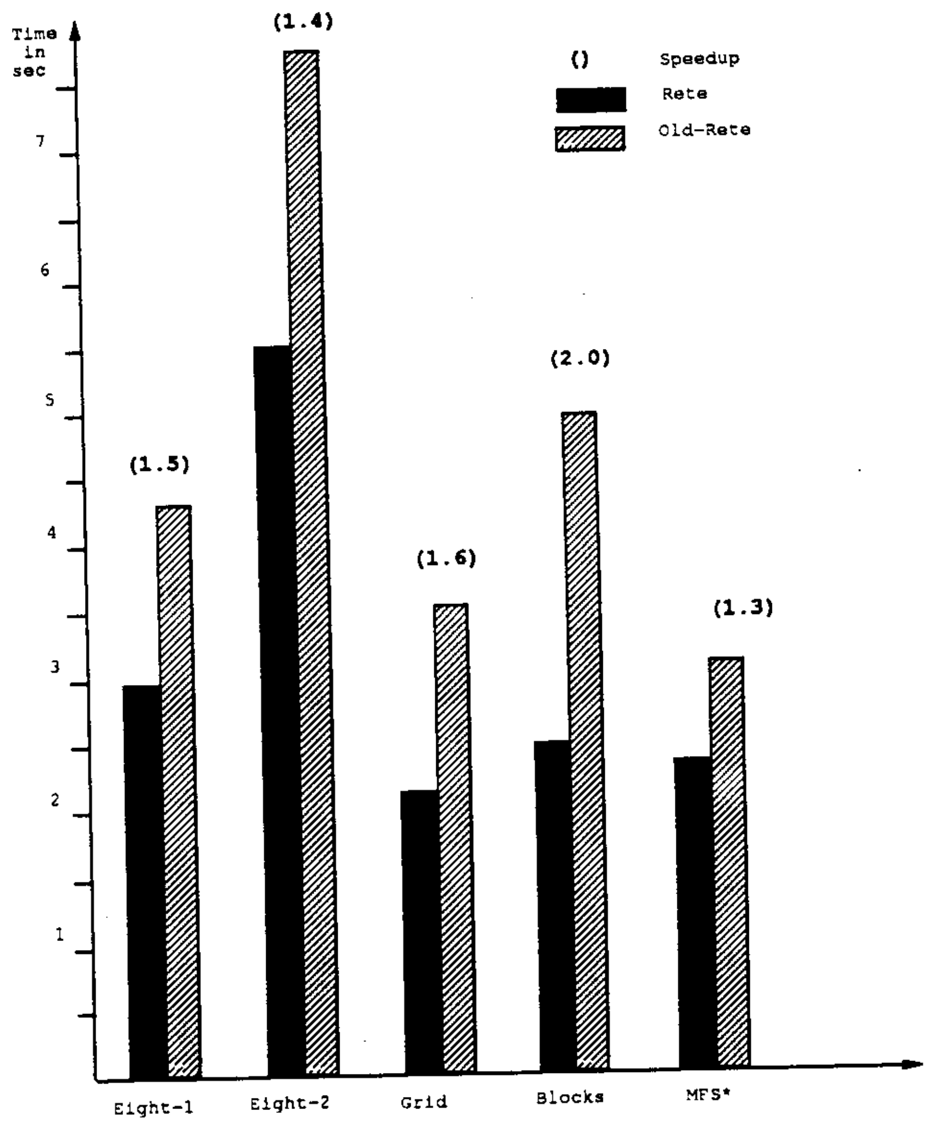


Figure 24: Comparing the performance of Old-Rete with Rete. Rete is used for comparison with Uni-Rete in our experiments.

An interesting point is whether the bilinear optimization is applicable to Rete. Much work has focused on bilinearizing Rete [Gupta 86, Scales 86, Tambe, et al. 88, Lee and Schor 89]. However, the focus of this work has been to improve parallelism and expressiveness of the production system, and not to speed up uniprocessor implementations. In fact, bilinear Rete networks have been observed to increase the uni-processor execution time by causing an increase in the match activity [Gupta 86]. The implementation of a general bilinear network is also fairly complex, and could introduce a much larger overhead [Lee and Schor 89]. Finally, the benefits of sharing in an unrestricted system may not be as much as in a unique-attribute system. As described in Section 2 larger numbers of unique-attribute productions are required for the same coverage as a single unrestricted production. These unique-attribute productions are similar to each other, and lead to more sharing.

In summary, the Rete system used for comparison with Uni-Rete is a well-optimized implementation of Rete. Clearly, some further optimizations are possible in our Rete implementation, since it is a fairly complex system. However, as described earlier, there is much scope for optimizing Uni-Rete as well.

7. Summary and Future Work

The 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 is a promising approach to eliminate match combinatorics without sacrificing production system functionality. A detailed evaluation of the unique-attribute representation appears in [Tambe 91, Tambe and Rosenbloom 90, Tambe, Newell and Rosenbloom 90].

The linear match bound in unique-attributes provides an *additional* benefit: a more efficient implementation technology that yields additional performance improvements. Currently, there appear to be three dimensions of this more efficient implementation technology — specialized match algorithms, specialized uni-processor hardware, and parallelization with lower overheads — all of which require detailed investigation. This paper reported on the results of investigating the first dimension — it reports on a specialized match algorithm for unique-attributes called Uni-Rete. Uni-Rete is a specialization of the widely used Rete match algorithm [Forgy 82]. In tasks done in the Soar, an integrated problem-solving and learning system [Laird, Newell, and Rosenbloom 87], *Uni-Rete has shown up to 10-fold speedup over Rete.* This phenomenon of a simpler representation yielding 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 [Hennessy 90].

An interesting optimization in Uni-Rete is the bilinear network organization. While this optimization provides some speedups, its current implementation is quite unoptimized. Optimizing this implementation is a key issue for future work. We also plan to focus on investigating schemes to overcome the limitations in Uni-Rete, particularly that of subgoaling. A longer term issue is to support the compilation of new productions into Uni-Rete at run-time, a capability required for learning production systems such as Soar. Our long term goal has been to understand the impact of specific representations, such as unique-attributes, on the performance of AI programs [Tambe and Rosenbloom 90]. Uni-Rete has allowed us to understand a key advantage of such representations.

Acknowledgments

We thank Brian Milnes for many interesting discussions about Uni-Rete.

References

[Acharya and Tambe 89a]
Acharya, A. and Tambe, M.
Production systems on message passing computers: Simulation results and analysis.
In *Proceedings of the International Conference on Parallel Processing*, pages 246-254. 1989.

[Acharya and Tambe 89b]
Acharya, A. and Tambe, M.
Efficient implementations of production systems.
VIVEK: A quarterly in artificial intelligence 2(1):3-18, 1989.

[Forgy 82] Forgy, C. L.
Rete: A fast algorithm for the many pattern/many object pattern match problem.
Artificial Intelligence 19(1):17-37, 1982.

[Forgy 84] Forgy, C. L.
The OPS83 Report.
Technical Report CMU-CS-84-133, Computer Science Department, Carnegie Mellon University, 1984.

[Forgy and Gupta 86]
Forgy, C. and Gupta, A.
Preliminary architecture of the CMU production system machine.
In *Hawaii International Conference on Systems Sciences*, pages 194-200. January, 1986.

[Gupta 86] Gupta, A.
Parallelism in production systems.
PhD thesis, Computer Science Department, Carnegie Mellon University, 1986.
Also a book, Morgan Kaufmann, (1987).

[Gupta, et. al. 88] 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.

[Hennessy 90] Hennessy, J.
RISC Architecture: A Perspective on the Past and Future.
Advances in VLSI.
In Sietz, C. L.,
MIT press, 1990.

[Kalp, et. al. 88] Kalp, D., Tambe, M., Gupta, A., Forgy, C., Newell, A., Acharya, A., Milnes, B., and Swedlow, K.
Parallel OPS5 User's Manual.
Technical Report CMU-CS-88-187, Computer Science Department, Carnegie Mellon University, November, 1988.

[Laffey et. al. 88] 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):27-45, 1988.

[Laird, Newell, and Rosenbloom 87]
Laird, J. E., Newell, A. and Rosenbloom, P. S.
Soar: An architecture for general intelligence.
Artificial Intelligence 33(1):1-64, 1987.

[Lee and Schor 89]
Lee, H.S., and Schor, M.
Match algorithms of generalized Rete networks.
Technical Report RC 14709 (#65946), IBM TJ Watson research center, 1989.

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

[McDermott 82] McDermott, J.
R1: A rule-based configurer of computer systems.
Artificial Intelligence 19(2):39-88, 1982.

[Minton 85] Minton, S.
Selectively generalizing plans for problem-solving.
In *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pages 596-599. 1985.

[Minton 88] Minton, S.
Quantitative results concerning the utility of explanation-based learning.
In *Proceedings of the Seventh National Conference on Artifical Intelligence*, pages 564-569. Morgan Kaufmann, 1988.

[Miranker 87] Miranker, D. P.
Treat: A better match algorithm for AI production systems.
In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 42-47. 1987.

[Miranker and Lofaso 91]
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):3-11, 1991.

[Nayak, Gupta, and Rosenbloom 88]
Nayak, P., Gupta, A. and Rosenbloom, P.
Comparison of the Rete and Treat production matchers for Soar (A summary).
In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 693-698. 1988.

[Newell 90] Newell, A.
Unified Theories of Cognition.
Harvard University Press, Cambridge, Massachusetts, 1990.

[Rosenbloom, Laird, Newell, and McCarl 91]
Rosenbloom, P. S., Laird, J. E., Newell, A., and McCarl, R.
A preliminary analysis of the Soar architecture as a basis for general intelligence.
Artificial Intelligence 47(1-3):289-325, 1991.
Also to be published in Kirsh, D. and Hewitt, C. ed., *Workshop on Foundations of Artificial Intelligence*, MIT Press, Cambridge, Massachusetts (In press).

[Scales 86] Scales, D.J.
Efficient matching algorithms for the SOAR/OPS5 production system.
Technical Report KSL-86-47, Knowledge Systems Laboratory, Stanford University, June, 1986.

[Tambe 91] Tambe, M.
Eliminating combinatorics from production match.
PhD thesis, School of Computer Science, Carnegie Mellon University, May, 1991.
(Available as a tech report CMU-CS-91-150).

[Tambe and Newell 88]
Tambe, M. and Newell, A.
Some chunks are expensive.
In *Proceedings of the Fifth International Conference on Machine Learning*, pages 451-458. June, 1988.

[Tambe and Rosenbloom 89]
Tambe, M. and Rosenbloom, P.
Eliminating expensive chunks by restricting expressiveness.
In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 731-737. 1989.

[Tambe and Rosenbloom 90]
Tambe, M. and Rosenbloom, P.
A framework for investigating production system formulations with polynomially bounded match.
In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 693-400. 1990.

[Tambe, et al. 88] 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.
In *Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming: Experience with applications, languages, and systems*, pages 146-160. 1988.

[Tambe, Newell and Rosenbloom 90]
Tambe, M., Newell, A., and Rosenbloom, P. S.
The problem of expensive chunks and its solution by restricting expressiveness.
Machine Learning 5(3):299-348, 1990.

[Waterman and Hayes-Roth 78]
Waterman, D. A., Hayes-Roth, F.
Pattern-directed inference systems.
Academic press, 1978.