CHAPTER 13

Efficient Matching Algorithms for the Soar/OPS5 Production System

D. J. Scales, Stanford University

Abstract

SOAR is a problem-solving and learning program intended to exhibit intelligent behavior. SOAR uses a modified form of the OPS5 production system for storage of and access to long-term knowledge. As with most programs which use production systems, the match phase of SOAR's production system dominates all other SOAR processing. This paper describes the results of an investigation of various ways of speeding up the matching process in SOAR through additions and changes to the OPS5 matching algorithm.

1. Introduction

SOAR [8, 9] is a system intended to exhibit general intelligent behavior. The SOAR architecture combines two paradigms that have been widely used for modeling cognitive behavior: production (rule) systems and systems which achieve goals by search in a problem space. In some models of cognitive behavior [14], production systems are the principal cognitive mechanism. Rychener [16] proposes that production systems are an appropriate and effective architecture for implementing a wide variety of artificial intelligence tasks. The behavior of such pure production systems is completely determined by the sequence of rule executions, and the overall model of control of the systems is the strategy which determines at any point which of the eligible rules should be executed. In SOAR, the underlying rule system is used only for storing long-term knowledge, while the model of control is that of search within problem spaces governed by a fixed decision procedure. The property of rule systems that is crucial for SOAR is that knowledge can be stored in a rule so that it becomes available exactly when it is relevant.

This paper is concerned with the efficiency of the production system used by SOAR. SOAR uses a modified version of a well-known production system, OPS5 [3]. In OPS5, as in many other production systems, a rule becomes eligible to fire when its "conditions" match some set of elements currently in memory. For such systems, the matching procedure which determines the rules that are matched often dominates all other computations and determines the speed of execution of the production system. OPS5 uses an efficient matching algorithm known as the Rete algorithm [2, 4] for determining which productions are eligible to fire. However, the way knowledge is represented in SOAR tasks and the way the SOAR architecture uses the OPS5 production system both have an effect on the efficiency of this matching algorithm. In this paper we investigate ways the Rete algorithm can be made to run faster when used with the SOAR system.

In the first few sections we present background information on OPS5 and SOAR. Sections 2 and 3 describe production systems in general and the OPS5 production system in particular. Section 4 describes the Rete matching algorithm used by OPS5 and some details of the LISP OPS5 implementation of this matching algorithm. In Section 5 we give a general outline of the SOAR architecture, focusing especially on the properties of the SOAR system that are relevant to the efficiency of the Rete algorithm. Section 6 details some statistics on the average operation of the Rete algorithm for SOAR tasks, comparing these with corresponding statistics for OPS5 tasks.

In the remaining sections we describe additions and modifications to the Rete algorithm intended

to speed its operation for SOAR production sets. Implementation details and timing results for each change are presented where relevant. We conclude by discussing briefly some of the other attempts at speeding up the Rete algorithm and by summarizing our results.

2. Production Systems

A *production system* consists of a collection of productions (rules), called the *production memory*, and a *working memory*. Each rule is specified by a set of conditions and a set of actions, such that the rule is eligible to execute its actions (to "fire") when its conditions are satisfied. Working memory is a global database containing data elements which are referenced by the conditions of the productions and created, modified, and removed by the actions of the productions. Production memory may be thought of as a long-term memory storing knowledge about how to do a task (the "program"), whereas the working memory contains temporary information and results accumulated while executing the current instance of the task. Groups of productions often work together in that the actions of one rule firing may cause the conditions of another rule to be satisfied, making that rule eligible to fire. However, one advantage of production systems is that each rule becomes eligible to execute exactly when it is applicable (as specified by its conditions), independent of any other rules in the rule set.

A production system interpreter executes a set of rules by repeatedly performing the following steps, known as the *recognize-act cycle*:

- Determine all productions whose conditions are satisfied, given the current contents of working memory.
- Choose a subset of the productions whose conditions are satisfied.
- Execute the actions of those productions.

Together, the above steps constitute one cycle of execution of the production system. The set of all productions whose conditions are currently satisfied is called the *conflict set*. Thus, the second step above, called *conflict resolution*, consists of determining which of the productions currently in the conflict set should be executed. The conflict resolution strategy is essentially an overall control strategy for the production system which arbitrates when more than one production is applicable in a particular situation.

Production systems differ in a variety of aspects relating to production memory, working memory, and method of conflict resolution:

- Overall organization of working memory and the data type(s) allowed for working-memory elements
- Kinds of matching primitives and other tests on working-memory elements that can occur in the conditions of a production
- Kinds of actions that can appear in a production, including actions on working-memory elements and input/output
- Escape mechanisms for calling arbitrary procedures from within the conditions or actions of a rule
- Method of conflict resolution

3. OPS5

OPS5 is a production system language in which all working-memory elements (wme's) are vectors of symbols (or numbers). The individual components of a wme are referred to as the *fields* of the wme. Each production has the form:

```
(p production-name
  ( ... condition ...)
  ( ... condition ...)
  .
  .
  .
  ( ... condition ...)
  -->
  ( ... action ...)
  ( ... action ...)
  .
  .
  .
  ( ... action ...)
)
```

Since this paper is concerned only with speeding up the condition matcher of OPS5, we shall only discuss the details of OPS5 conditions. The reader is referred to [3] for a complete description of OPS5 syntax.

Each condition of the production is a list of match primitives. Each wme that matches the condition is said to be an *instantiation* of that condition. For example, the condition

```
(state <s> item <> 3)
```

matches wme's whose first field is the symbol "state", whose third field is "item", and whose fourth field is not equal to the number 3. The symbol "<s>" is the name of an OPS5 variable; a symbol is the name of a variable if and only if it begins with "<" and ends with ">". A variable in a condition matches any value of the corresponding field. However, the variable is bound to the value it matches, and any other occurrence of the same variable within the conditions of the production matches only that value. For example, the condition

```
(operator <q> super-operator <q>)
```

matches wme's whose first field is "operator," whose third field is "super-operator," and whose second and fourth fields are equal. The condition

```
(operator <q> super-operator <> <q>)
```

is similar, except the second and fourth fields must be unequal.

An instantiation of a production is a list of wme's such that each wme of the list is an instantiation

of the corresponding condition of the production, and all occurrences of each variable throughout the conditions can be bound to the same value. For example, a list of two wme's A and B instantiate the conditions

```
(goal-context <g> problem-space <p>)
(goal-context <g> state <s>)
```

if the first fields of both A and B are "goal-context", the third fields of A and B are "problem-space" and "state," respectively, and the second fields of A and B are equal. If the conditions above were the conditions of a production, and such wme's A and B existed, a production instantiation, consisting of the name of the production and a list of A and B, would at some point be placed in the conflict set. An instantiation is executed by executing the actions of the production, after replacing variables in the actions by the values to which they were bound for the instantiation.

In practice, wme's are often used to store information about objects, and the fields represent the values of various attributes of the objects. OPS5 provides a way of giving attribute names to the fields of different kinds of wme's. Each wme has a class name, indicated by the value of the first field of the wme, and each class can have names associated with various fields of the wme. Attribute names are indicated by a preceding up-arrow. When an attribute name appears in a condition or wme, it indicates that the following value or pattern refers to the field corresponding to that attribute. For example, the condition

```
(preference ↑object p0001 ↑role problem-space ↑value acceptable)
```

matches wme's whose class name is "preference", whose "object" field is "p0001", whose "role" field is "problem-space", and so on.

OPS5 allows a condition to be negated by preceding it by a minus sign. A negated condition indicates that no wme should match that condition. That is, a set of conditions and negated conditions is matched if and only if there exist a set of wme's which match the positive conditions with consistent bindings of the variables, and, for those variable bindings, there are no wme's that match any of the negated conditions. Hence, the condition list

```
(goal ↑status active ↑type find ↑object block ↑type <x>)
- (block ↑status available ↑type <x> ↑weight < 5)
```

is satisfied only if there is a wme representing an active goal for finding a block of a particular type, but there is no wme representing an available block of that type with weight less than 5.

4. The Rete Algorithm and the Rete Network

The process used by OPS5 to match the conditions of productions against working memory is based on the Rete algorithm. The Rete algorithm builds a lattice structure, called the Rete network, to represent the conditions of a set of productions. The Rete network takes advantage of two properties of production systems which allow for efficient matching:

- The contents of the working memory change very slowly.[1]
- There are many sequences of tests that are common to the conditions of more than one production.

The Rete algorithm exploits the first property by storing match information in the network between cycles, so that it only matches a wme against each production condition once, even if (as is likely) the wme remains in working memory for many cycles. Hence, the Rete network contains two types of nodes, test nodes and memory nodes. A test node indicates tests that should be executed to determine if a particular condition or set of conditions is matched by a wme or set of wme's, while a memory node is used to store information on partial production matches. Because of the storing of partial matches in the network, only the changes to working memory, rather than the whole contents of working memory, need be processed each cycle. Each time a wme is added to working memory, the wme is "filtered" down the network, causing new partial matches to be formed and possibly causing one or more productions to be instantiated and placed in the conflict set. An identical process occurs when a wme is removed from memory, except that partial matches are discarded as the wme filters down the network.

The Rete algorithm takes advantage of the second property by sharing common test and memory nodes as it adds nodes to the network to represent the conditions of each successive production. Because of the structure of the Rete network, the Rete algorithm can easily determine, as it is adding nodes to the network to represent a particular production, whether the nodes required already exist in the network and so can be reused rather than creating a new node.

4.1. The Details of the Rete Network

The memory nodes of the Rete network store partial match information in the form of ordered lists of wme's called *tokens*. A match to a list of conditions (e.g. the left side of a production) is represented by a token in which the first wme matches the first condition, the second wme matches the second condition, and so on. A token stored at a particular memory node in the network represents a successful instantiation of the list of conditions represented by the set of nodes leading into that node. In the discussion below, we often refer to the number of tokens stored in a memory node as the "size" of the node.

Figure 4-1 illustrates the structure of the Rete network for the following condition list:

```
(state  ↑identifier <s> ↑attribute list ↑value <l>)
(object ↑identifier <l> ↑attribute next ↑value <l>)
(state  ↑identifier <s> ↑attribute size ↑value <= 15)
```

[1] Miranker [13] cites production systems such as ACE in which the contents of working memory changes completely with every cycle. However, for most typical applications of production systems, including SOAR tasks, working memory changes slowly.

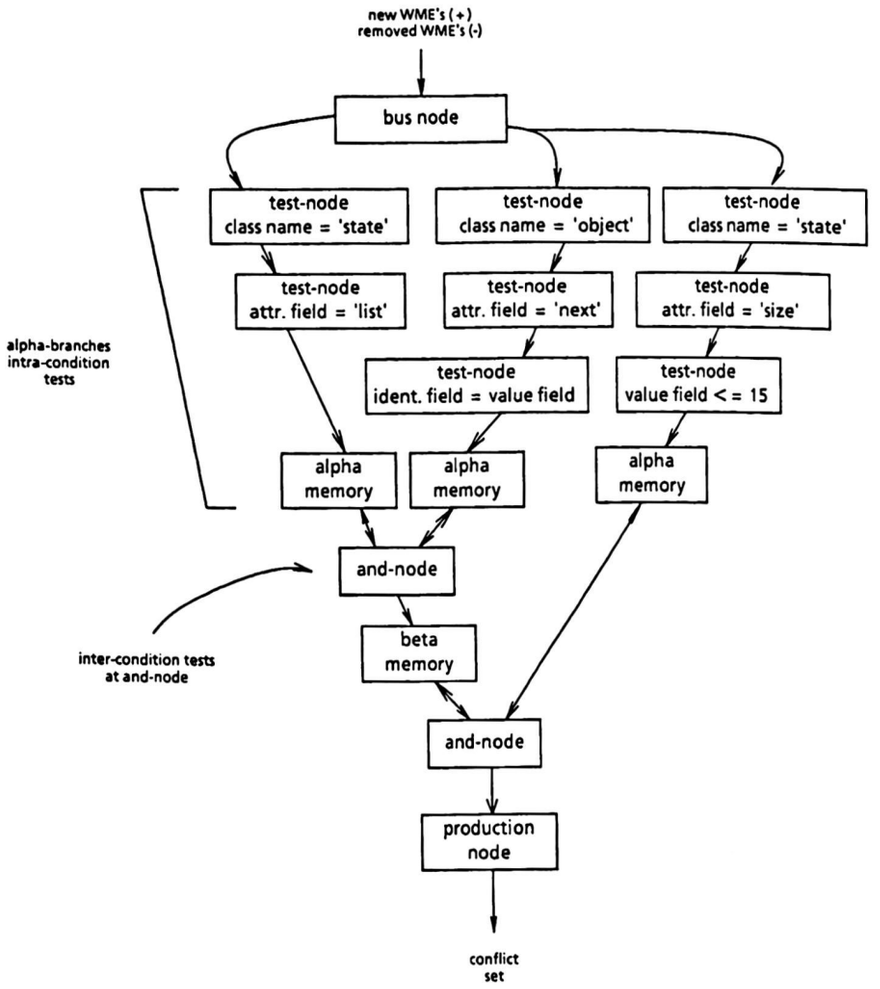

Figure 4-1: Structure of a Small Rete Network

Each condition of a production is represented in the Rete network by a singly-linked list of nodes, called an *alpha branch*. As with dataflow networks, the successor of each node is often referred to as the output of that node. These nodes, referred to as *alpha nodes*, execute the *intra-condition* tests,

those tests which concern only the wme that the condition is matching. These include (1) tests involving constants and (2) tests resulting from the appearance of a variable twice in the same condition. Alpha branches do not include tests matching the occurrence of a variable in the condition with an occurrence of the same variable in a previous condition of the production. Such *inter-condition* tests are handled by the and-nodes described below. For example, the intra-condition tests for the first condition listed above are that (1) the class name must be "state," and (2) the attribute field must be "list." The intra-condition tests for the second condition are that (1) the class name must be "object," (2) the attribute field must be "next," and (3) the values of the identifier and value fields must be the same. Figure 4-1 shows the intra-condition tests for all three conditions within the test nodes that execute them. The requirement that the bindings of <I> in the first two conditions must be the same is ensured by an inter-condition test.

Each alpha branch is terminated by a memory node, in which is stored all wme's that have satisfied all the tests of the alpha branch. Actually, as stated above, memory nodes store tokens, but, since an alpha branch matches one condition, the tokens stored in the memory node below it consist of only one wme. The memory node below an alpha branch is called an *alpha memory*.

The two alpha branches representing the first two conditions of a production are joined together by an *and-node*, which has the alpha memories of the two branches as "inputs". A *beta memory* node is linked to the and-node as its output, storing any tokens which leave the and-node. If there is a third condition in the production, the alpha memory representing that condition is joined by another and-node to the beta memory representing the first two conditions, and again a beta memory is linked as the output of the new and-node. Similarly, the alpha memory of each successive condition of the production is joined by an and-node to the beta memory representing all previous conditions. One input of the and-node always represents the first n conditions of a production, while the other input represents the (n + 1)st condition. The inputs are distinguished by referring to the input representing the first n conditions as the *left input* and to the other as the *right input*. Similarly, the memory nodes that are the left and right inputs of the and-node are referred to as the left and right memory of the and-node, respectively. In Figure 4-1, we have drawn the network so that left inputs enter and-nodes on the left, and right inputs enter on the right.

The left memory of an and-node always contains tokens which match the first n conditions of a production, while the right memory contains tokens matching the (n + 1)st condition of that production. If a token from the left memory and a token from the right memory have the same values for fields labeled by the same variables in the production, then they can be concatenated to form a new token which matches the first n + 1 conditions of the production. Hence, each and-node contains a list of tests (called *and-tests* or inter-condition tests) which insure that the token from the left memory is consistent with the token from the right memory. Finally, a "production" node is linked to the final

and-node that represents all of the conditions of a production. Any token that filters down to a production node indicates an instantiation of the production.

Each alpha branch is linked to the "top" node of the network, the *bus node*. When a wme is added to memory, the network is traversed in a depth-first fashion starting from the bus node. (In the following, we often talk as if a wme or token is moving through the network, although clearly it is the OPS5 matcher that is traversing the network.) Each alpha branch is eventually encountered and the tests on the alpha branch are executed on the wme. If the wme satisfies the test at one node of the alpha branch, it continues to the next node on the alpha branch. However, if it ever fails a test on the alpha branch, the depth-first traversal is immediately backed up and goes down the next alpha branch. When a wme filters down an alpha branch to its alpha memory, it is stored in the alpha memory and continues down to the input (say, the right input) of the and-node below the alpha branch. Each token in the left memory of the and-node is matched against the wme (actually token) which has just "arrived." If a token from the left is consistent with the newly-arrived token, as determined by the and-tests, then the two tokens are combined and the new token continues down the network. Similarly, if a token comes down the left input of an and-node, that token is matched with each token in the right memory. For each match, the just-arrived token is joined with the matching token to form a new token which is sent down the network. Whenever a token reaches a production node, an entry is added to the conflict set indicating a match of the token with the production.

A wme must also be filtered down the network when it is removed from working memory so that all partial matches (tokens) involving it are removed from the network memories. This removal of tokens is accomplished by the same depth-first traversal of the network, except that when a token passes through a memory node, it is removed from the memory list of the node rather than added to it. If a token reaches a production node, the entry corresponding to that token and production is removed from the conflict set. Because a change to a field of a wme may completely change the conditions it matches and the set of partial matches (tokens) of which it should be a part, a change to a wme is handled by removing the old version of the wme from working memory and from the network and then adding the new version.

Negated conditions are handled by the use of another kind of node with two inputs, the not-node. The alpha branch representing the negated condition is built in the usual way, terminated by an alpha memory. The not-node joins the branch of the network representing the conditions preceding a negated condition to this alpha memory. In operation, the not-node maintains, for each token that has entered on the left input, a count of the number of tokens in the right memory with which that token is consistent. This count is initially calculated when the token from the left enters the and-node and is updated every time a token enters the right input. If the token that has entered on the left ever

becomes consistent with none of the tokens in the right memory, that token can be sent to the output as a successful match. Conversely, if the "consistency count" of a token ever changes from zero to non-zero, then that token is no longer a successful match and so must be sent to the output with an indication that all tokens based on it are to be removed. Because the not-node must maintain a consistency count for each of the tokens that has entered from the left, it has its own internal memory for storing the tokens and their consistency counts. Hence, no beta memory is created below a two-input node if the next two-input node is a not-node. Not-nodes will not be mentioned much more below, because they are used so infrequently that their operation does not have much impact on the overall efficiency of the Rete algorithm. Some of the changes described below for speeding up the Rete algorithm are actually slightly more complicated than described, because they must take not-nodes into account.

As mentioned above, the Rete algorithm shares nodes of the network between productions when possible. For example, if two productions have conditions which produce the same sequence of intra-condition tests, then a single alpha branch with those tests can be used for both conditions. By checking the existing network before building new nodes, the Rete algorithm is able to share parts or all of alpha branches, including alpha memories. Even two-input nodes can be shared, if both the input nodes are already shared. Two-input node sharing is infrequent compared to sharing of the alpha branches, since it happens only when two productions have a common sequence of conditions at the beginning of their condition lists. Note that two conditions or sequences of conditions produce the same network of nodes, even if they use different variable names, as long as one can be obtained from the other by a consistent renaming of variables. For example, the conditions

```
(state ↑identifier <s>  ↑attribute item ↑value <x>)
(state ↑identifier <ss> ↑attribute item ↑value <item>)
```

produce the same sequence of test nodes, and so can share an alpha branch. Because of sharing, each node has, instead of one output, a list of outputs, just as the top bus node of the network has many alpha branch outputs. Any token leaving a node goes to each of its outputs. Sharing not only reduces the size of the network, but also the CPU time required to filter a wme down the network and the number of tokens stored in the network (since memory nodes can be shared.) Some statistics on the average amount of node sharing that occurs for typical OPS5 and SOAR production sets are given in Section 6 below.

4.2. The OPS5 Implementation of the Rete Algorithm

We shall now describe some of the specific details of the implementation of the Rete algorithm used by the LISP version of OPS5, to be referred to as the OPS5 Rete algorithm.

Because the traversal of the network is the same for both additions and removals of wme's, the

same OPS5 code implements both adds and removes. The setting of a global flag determines whether tokens entering a memory node will be added to the memory list or removed from the memory list.

In the OPS5 Rete network, each node is just a list structure containing the type of the node, specific information about the action of the node, and a list of pointers to the outputs of the node. For memory nodes, the specific information consists of an indicator of where to store tokens. For alpha nodes, the node type specifies the kind of test to be executed, and the extra information specifies which fields and/or constants are involved in the test. And-nodes contain backpointers to their left and right memory nodes and a list of the and-tests to be executed. Each of the node types is actually the name of a function which, given the specific information about a node of that type, executes the actions of the node, and calls each of the output nodes if necessary. The token that is "entering" the node is stored in a global variable before the call to the node function. Since an and-node may be called from either its left or right input node, the "AND" function must be given an extra bit of information indicating from which input it is being called.

The memories of memory nodes are simple lists of tokens. Adding a token to a memory involves merely adding the token to the beginning of the list, while removing a token from a memory requires a linear search of a list of tokens. Tokens are represented as linked lists (standard LISP lists) of wme's. Linked lists have several advantages over other possible representations of lists, such as arrays. First, tokens vary widely in size, from one wme to as many wme's as there are conditions in the production with the maximum number of conditions. More importantly, representing tokens as linked lists allows structure to be shared among them, just as nodes are shared in the network. The token of size n + 1 that is created in an and-node by joining tokens of size n and size 1 can share the whole structure of the token of size n. (In fact, the token of size n + 1 is created merely by CONS'ing the token of size 1 to the token of size n.) In this way, the tokens take up much less storage than if no structure is shared.

5. SOAR

We now present a general outline of the SOAR architecture and then focus on particular aspects of SOAR that affect the operation of the OPS5 Rete algorithm.

All tasks in SOAR are formulated in terms of achieving goals by search in a problem-space. A task is specified by a set of rules that encode the domain and control knowledge that guide the search. SOAR has a decision procedure that decides the next value of the current problem-space, state, or operator, based on the available knowledge. To allow all relevant knowledge to be made explicit so that it can influence the decision procedure, each decision is preceded by an *elaboration phase* in

which all production instantiations in the conflict set are executed. The execution of the initial set of instantiations may cause more productions to match, so the elaboration phase continues until no more productions are ready to fire (until "quiescence"). If the available knowledge is enough to determine the next decision, then that decision will be made and the process repeats. However, if the decision procedure cannot successfully determine the next decision, due to lack of information, conflicting information, etc., then the SOAR architecture will automatically create a subgoal to resolve this impasse. Processing within subgoals is identical to processing in the main goal, and, indeed, another impasse may occur in a subgoal, resulting, in general, in a whole hierarchy of subgoals. A subgoal is automatically terminated when the impasse that created the subgoal is resolved. The SOAR decision cycle is illustrated in figure 5-1.

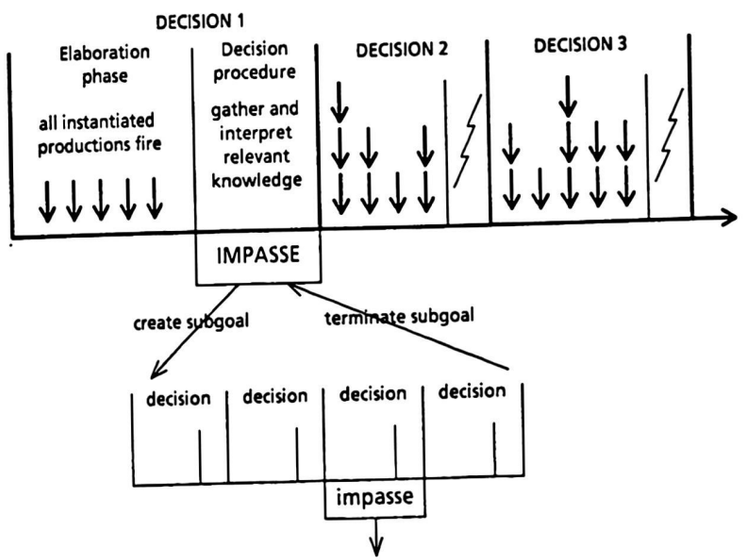

Figure 5-1: The SOAR Decision Cycle

Another important aspect of SOAR is its learning mechanism. Whenever a subgoal terminates and learning is "on", the SOAR architecture creates a new production, called a *chunk*, which summarizes the results of the subgoal. The conditions of the chunk test those aspects of the current situation that were tested in the subgoal, and the actions produce the results that were created in the subgoal. The chunk is added to production memory and will fire, like any other production, when its conditions are satisfied. In fact, if the situation that caused the subgoal to arise ever happens again, the chunk will

fire, producing results that will avoid the impasse and subgoal that happened the first time. The reader is referred to [8], [9] and [10] for a further description of the architecture of SOAR and the applications of chunking.

SOAR uses the OPS5 production system as the underlying rule system which stores long-term knowledge and makes it available when relevant. As such, all knowledge about a particular task or domain is specified by OPS5 productions, and the OPS5 matcher is used to determine which rules are ready to fire at any point. However, several of the properties of the SOAR architecture mentioned above affect the operation and efficiency of the Rete matching algorithm. First, SOAR eliminates the OPS5 conflict resolution, since all eligible production instantiations are fired during the elaboration phase. Another property of SOAR is that most removal of wme's from working memory is done by the SOAR architecture at the termination of individual subgoals. SOAR productions only create wme's, adding information which can be used to guide the processing of the current subgoal. When the subgoal is terminated, all the wme's created during the subgoal are removed except for the results of the subgoal. The only other removals of wme's occur when the architecture changes the value of the problem-space, state, or operator of a goal. Such changes require the removal of the wme containing the old value and the addition of a wme containing the new value. SOAR's creation of chunks also has a great impact on the OPS5 matching algorithm, since it requires the ability to add productions to the network efficiently while the production system is running.

The biggest difference between SOAR's use of OPS5 and its normal use is in SOAR's representation of objects. In OPS5, it is intended that an object be represented by a single wme with appropriate attribute fields for that kind of object. In SOAR, every object (except preferences, which are described below) is represented by a set of wme's, each of which itself represents an attribute/value pair for the object. For example, the wme's

```
(goal-context ↑identifier g0001 ↑attribute problem-space ↑value p0003)
(goal-context ↑identifier g0001 ↑attribute state ↑value s0005)
(goal-context ↑identifier g0001 ↑attribute item ↑value cell1)
(goal-context ↑identifier g0001 ↑attribute item ↑value cell2)
```

represent a SOAR object with identifier g0001, value p0003 (the identifier of another object) for attribute problem-space, value s0005 for attribute state, and values cell1 and cell2 for attribute item. The identifier of a SOAR object is a unique symbol (a LISP "gensym") generated by SOAR when the object is created. The identifier represents the object and is used to refer to the object in the attributes of other objects. For example, the second wme above indicates that the object represented by the symbol s0005 is the value of the state attribute of the object represented by g0001. Almost all SOAR objects are linked through their identifier field either directly or indirectly via other objects to a goal.

The wme's that have four fields (class name, identifier, attribute, and value) are called

augmentations. The only other kind of wme is one used to represent a *preference*, a SOAR object which directly encodes information about the desirability of a particular decision. Each wme representing a preference has class name "preference" and eight attribute fields named object, role, value, reference, goal, problem-space, state, and operator.[2] Thus a typical preference is

```
(preference ↑object x33 ↑role operator ↑value worse ↑reference x32
   ↑goal g14 ↑problem-space p34 ↑state s10)
```

A typical SOAR production is shown in figure 5-2.[3]

```
(p eight*apply-move-tile
   (goal-context ↑identifier <g> ↑attribute problem-space ↑value <p>)
   (space ↑identifier <p> ↑attribute name ↑value eight-puzzle)
   (goal-context ↑identifier <g> ↑attribute state ↑value <s>)
   (goal-context ↑identifier <g> ↑attribute operator ↑value <q>)
   (operator ↑identifier <q> ↑attribute name ↑value move-tile)
   (state ↑identifier <s> ↑attribute blank-cell ↑value <c1>)
   (operator ↑identifier <q> ↑attribute blank-cell ↑value <c1>)
   (operator ↑identifier <q> ↑attribute tile-cell ↑value <c2>)
   (state ↑identifier <s> ↑attribute binding ↑value <b2>)
   (binding ↑identifier <b2> ↑attribute cell ↑value <c2>)
   (state ↑identifier <s> ↑attribute binding ↑value <b1>)
   (binding ↑identifier <b1> ↑attribute cell ↑value <c1>)
   (binding ↑identifier <b1> ↑attribute tile ↑value <t1>)
   (binding ↑identifier <b2> ↑attribute tile ↑value <t2>)
  -->
   (make preference ↑object <s2> ↑role state ↑value acceptable
    ↑goal <g>  ↑problem-space <p> ↑state <s> ↑operator <q>)
   (make state ↑identifier <s2> ↑attribute blank-cell ↑value <c2>)
   (make state ↑identifier <s2> ↑attribute tile-cell ↑value <c1>)
   (make state ↑identifier <s2> ↑attribute binding ↑value <b3>)
   (make state ↑identifier <s2> ↑attribute binding ↑value <b4>)
   (make binding ↑identifier <b3> ↑attribute tile ↑value <t2>)
   (make binding ↑identifier <b3> ↑attribute cell ↑value <c1>)
   (make binding ↑identifier <b4> ↑attribute tile ↑value <t1>)
   (make binding ↑identifier <b4> ↑attribute cell ↑value <c2>))
```

Figure 5-2: A Typical SOAR Production

As is obvious from the figure, variables almost always occur in the identifier field in conditions. Each condition is usually linked to the previous conditions by a previous occurrence of the variable that is in its identifier field. Typically, objects are accessed through the goal to which they are connected. So, most of the conditions of a production are linked in a hierarchical structure rooted in conditions at the beginning of the condition list that match augmentations of a goal.[4] Because of the indirect

[2] A change to SOAR is being considered in which preferences, like all other SOAR objects, are represented by augmentations.

[3] Productions are actually entered into the system in a more compact form. The expanded OPS5 form that is used to build the network for the production is shown here.

[4] Some of these properties depend in part on the way SOAR automatically reorders conditions of productions, as described in Section 9.

representation of objects in SOAR, SOAR productions tend to be much longer than typical OPS5 productions. Where an OPS5 production requires only one condition to match various attributes of an object, a SOAR production requires several conditions. On the other hand, SOAR conditions, which most often match augmentations, are typically much shorter and more regular than OPS5 conditions.

There are several advantages to the representation of objects used in SOAR:

- SOAR objects can have multiple values for the same attribute (a simple representation of sets).
- Multiple productions can give values to different attributes of an object in parallel. This property is necessary if productions are allowed to fire in parallel during the elaboration phase. Parallel firing of rules during the elaboration phase is suggested by some mappings between SOAR processing and human cognitive processing and is also a logical way to speed up SOAR processing.
- Attributes need not be constants, but can be complex objects themselves.
- SOAR tasks can create new attributes for objects during a run.
- Attributes of objects can be matched by variables (by having a variable in the ↑attribute field of a condition).
- It is convenient not to have to declare all the attributes of each type of object, as must be done in OPS5.

A final aspect of SOAR that is relevant to the speed of the Rete algorithm used by is SOAR is that SOAR is written in LISP. An implementation of OPS5 in BLISS that is based on the same OPS5 Rete algorithm runs six times faster than the LISP OPS5 implementation [6]. The reason that the LISP implementation is so much slower than the BLISS one is that LISP programs, even compiled ones, have extra overhead in terms of memory management, type checking and function calls, while BLISS programs compile to very efficient machine code. However, since SOAR is written in LISP, we are interested in speeding up the LISP version of the OPS5 matching algorithm. Most of the results are independent of the use of LISP for implementing the matching algorithm, but some will be dependent in some way on the properties of LISP.

6. Rete Network Statistics for SOAR and OPS5 Tasks

Statistics about the composition of the Rete network and the flow of tokens in the network for various tasks can be obtained by adding statistics-gathering code to the Rete algorithm. Appendix A contains statistics obtained from runs of six different SOAR tasks. (There are actually only four distinct tasks; for two tasks, statistics were gathered for runs both with learning off and learning on. For tasks with learning on, statistics on the composition of the network are for the network at the end of the run, after all the chunks have been incorporated into the network.) Gupta [7] gives similar

statistics for four large OPS5 tasks. Below, we summarize the important facts derived from the statistics in Appendix A and compare, where relevant, the statistics for SOAR tasks with those given by Gupta for OPS5 tasks. These statistics will be useful below in explaining why certain changes speed up the Rete algorithm and in determining which changes proposed are most likely to speed up SOAR tasks.

6.1. Statistics on the Network

The first group of statistics obtained are the statistics concerning the composition of the network itself. Table 6-1 summarizes the distribution of nodes in the Rete network for the OPS5 and SOAR tasks.

	OPS5 tasks	SOAR tasks
% alpha nodes	15 - 25	6 - 11
% alpha memories	13 - 18	6 - 9
% beta memories	7 - 20	33 - 39
% and nodes	23 - 31	39 - 43
% not nodes	3 - 5	1 - 2
% production nodes	14 - 18	5 - 7
network sharing	2.0-2.9	3.2-3.5

Table 6-1: Composition of OPS5 and SOAR Rete Networks

The "network sharing" line indicates the ratio of size of the network when nodes are not shared to the size when nodes are shared. Gupta does not give the average numbers of outputs for the various types of nodes in the network, but it is possible to estimate these based on other statistics given. For SOAR tasks, the average number of outputs of alpha memories ranges from 4 to 8, while for OPS5 tasks the average number of outputs ranges from 2 to 5. For both SOAR and OPS5 tasks, the average number of outputs of and-nodes and beta memories is close to 1. These differences in statistics are easy to understand in terms of the differences between SOAR and OPS5 productions. Because the conditions of SOAR productions are so regular, there is much more sharing among alpha-test nodes for SOAR tasks, hence the smaller fraction of alpha-test nodes in the network for SOAR tasks. Conversely, because SOAR productions have, on average, a greater number of conditions than OPS5 productions, the percentage of and-nodes and beta memories is higher in networks for SOAR tasks. Overall, because of the regularity of SOAR conditions, sharing is greater for SOAR tasks than for OPS5 tasks.

The conclusion from these statistics is that the proportion of testing that occurs at the and-nodes is larger, on average, for SOAR tasks than for OPS5 tasks. Fewer tests are located on the alpha branches in networks for SOAR tasks than in networks for OPS5 tasks, while SOAR networks have a greater fraction of and-nodes than OPS5 networks. Hence, it seems clear that the speed of processing at the and-nodes is crucial to the speed of the Rete algorithm when it is applied to SOAR production sets.

The statistics also indicate that there are, on the average, 1.1 to 1.5 and-tests at each and-node. Hence, at least 50% to 90% of the and-nodes have only one and-test. This case of one and-test happens mainly because of the common SOAR situation in which a condition is linked to previous conditions only through its identifier field. For the OPS5 tasks in [7], the average number of tests at two-input nodes (including not-nodes) ranges from 0.37 to 1.27. These figures indicate that the "connectivity" between conditions in SOAR productions is much more regular than in OPS5 conditions.

6.2. Statistics on Node Operations

The second group of statistics in Appendix A are the dynamic statistics providing counts of various operations executed by the OPS5 Rete algorithm while executing a particular production set.

The percentage of alpha node activations for which the node tests succeed ranges from 5 to 13 percent for the six SOAR tasks and from 3 to 25 percent for the OPS5 tasks. One of the reasons that the "success rate" is so low for alpha nodes is that the network for a task usually contains groups of alpha nodes matching a particular field of a wme against various constants. Within each of these groups, at most one alpha node can succeed. Section 13 describes a method of using indexing to speed up these groups of alpha node tests.

It is clear from the data on memory operations that, for SOAR tasks, alpha memories typically get "more full" (contain more tokens) than beta memories. These statistics are intuitive, since, in general, there will be more matches for a single condition (as represented by the alpha memories) then for a conjunction of several conditions (as represented by the beta memories.) For one of the tasks in Appendix A, the eight puzzle task with learning on, the average size of the beta memories for memory operations is much larger than that for alpha memories. This statistic results from the occurrence of what is called the *cross-product effect* at several of the beta memories. The cross-product effect will be discussed in Section 9.

Though the average size of the alpha memories is greater than that of the beta memories, the average fraction of tokens that have to be checked, for SOAR tasks, when removing a token from a memory is much lower for the alpha memories than for the beta memories. This statistic arises from the fact that for SOAR tasks, the alpha memories operate almost like stacks. Because most wme's are removed only by the SOAR architecture at the termination of a subgoal, most alpha memories grow monotonically during the processing of a stack of goals. When a subgoal finally terminates, almost all the wme's created in the subgoal are removed. This removal is done in the reverse of the order in which the wme's were added to working memory. Because the wme's from the lowest subgoal are the ones most recently created, wme's are removed from working memory and hence from alpha memories in the opposite order from which they were created. Since tokens are always added to the

beginning of memory lists, alpha memories act like stacks, except that some wme's are not removed when a subgoal terminates but remain as results of the subgoal. Hence, usually only the first few tokens of an alpha memory need be searched when removing a particular token from the memory.

One would expect that the argument given above would apply to beta memories. However, on the average, for SOAR tasks, one-third to one-half of the tokens of a beta memory are examined before the particular token to be removed is found. The reason beta memories do not act like stacks is based on a subtle interaction between the operation of the and-nodes and the beta memories. Suppose a token T enters an input of an and-node and matches many tokens in the opposite memory. T is joined with each of the matching tokens to form new tokens $T_1, T_2, \ldots, T_n$, which are successively stored in the beta memory below the and-node. Since tokens are always added to the top of memory lists, the last token created, T_n, is at the top of the memory list of the beta node and the first token created, T_1, is the nth token down in the list when the and-node operation finishes. Now, if token T is removed sometime later, than the tokens $T_1, T_2, \ldots, T_n$ are removed from the beta node in the same order. So, n tokens must be examined to remove T_1, n-1 tokens must be examined to remove T_2, and so on, resulting in an average of n/2 tokens being examined for each remove. The obvious solution to this problem is to reverse the order that tokens are sent out of and-nodes when removals are being done. If this change is made to the operation of the and-nodes, then the beta memories act almost like stacks, as expected. The problem is also eliminated by using hash tables at the beta memories as described in Section 14 or by changing the removal process as described in Section 15.

The statistics on and-node calls for SOAR tasks indicate that from 45 to 85 percent of the time that a token enters the right input of an and-node (an "and-right" call), the opposite (left) memory is empty, so no and-tests need be executed. On the other hand, the fraction of so-called "null and-calls" is usually much lower for tokens entering the left input of and-nodes ("and-left calls"), ranging from 1 to 40 percent. These figures are another reflection of the fact that alpha memories are, on the average, more likely to be filled than beta memories. In all tasks there were more non-null and-left calls than non-null and-right calls. The statistics also indicate that for non-null calls, the size of the opposite memory is usually much greater for and-left calls than for and-right calls. These two facts together explain the statistic that from three to thirty times more and-tests are executed in and-left calls than and-right calls. Obviously, much more matching time is spent processing and-left calls than processing and-right calls.

7. Benchmark for Evaluating Modifications of the Rete Algorithm

To allow comparison of the effects of each of the changes to the Rete algorithm described below, we give for most modifications the resulting change in running time for a particular SOAR task. The task used as a benchmark is one that attempts to solve the eight puzzle, a puzzle which involves moving eight tiles within a three-by-three grid until they are arranged in a desired configuration. The particular implementation of the task used (i.e. the particular set of SOAR productions used to specify the task) is typical in its representation of objects as hierarchies of subobjects and in its use of subgoals for selecting and evaluating operators. It uses productions for most of the usual ways that SOAR tasks use productions: setting up initial problem spaces and states, determining the currently relevant operators, rejecting inapplicable operators, applying operators, evaluating operators and states, printing out some kind of representation of the current state, and checking for the desired state of a goal. Along with the 11 productions that specify the eight-puzzle task, a set of 51 "default" productions are loaded. This standard set of productions is loaded for every task and specifies default actions for responding to various impasses and for evaluating possible decisions (problem-spaces, states, and operators.)

Of course, the behavior of SOAR can vary widely in some respects for different tasks. For example, the eight-puzzle task used as a benchmark has a maximum of three levels of subgoals at one time, whereas the goal hierarchy of other tasks can often become much larger than three. Despite the wide range of SOAR tasks, the benchmark is useful in giving at least a rough estimate of the relative effects of various changes.

The times actually given are the CPU time used by the matcher itself when the eight-puzzle task is run on the Xerox 1132 LISP workstation. The SOAR architecture itself has some overhead in running the decision procedure, creating and removing subgoals, creating chunks, etc., but that processing time is not included in the figures below. The times below also do not include LISP garbage collection time. Garbage collections occur sporadically in LISP, so garbage collection times can vary for identical runs, depending on the state of LISP's storage when the run starts. Also, there is no way to determine precisely the fraction of the garbage collection time which is due to processing of the matcher rather than to processing of the SOAR architecture. In general, garbage collection times vary directly with the amount of storage a program uses and discards. Since all of the changes below either leave unchanged or reduce the storage requirements of the Rete algorithm, the garbage collection times should remain the same or decrease for each of the changes.

All of the times below (except those involving learning in Section 12 below) are for a particular run of the eight-puzzle task taking 143 decision cycles. (Further statistics on the operation of the Rete algorithm for this run are given in Appendix A.) Most of the changes were sequentially added to SOAR, so the running times are based on several versions of SOAR, some already incorporating some of the changes. Some of the changes are completely independent, in that the speedup in seconds when both changes are used is equal to the sum of the speedups when each is used individually. However, most of the changes interact such that the combined speedup of the two changes is less than what would be expected from the individual speedups. The interactions are usually complex enough that there is no easy way to determine the combined speedup from the individual speedups. Hence, the running times given are only rough indications of the relative speedups for the changes.

In the following sections we describe several types of changes to the Rete algorithm. Section 8 describes a simplification to the Rete network that also simplifies several of the speedup modifications. Sections 9, 10, and 11 describe changes that affect the number and distribution of node operations that are executed in matching a particular production set. Section 9 discusses the effect of the order of conditions in productions on the operation of the Rete algorithm and describes the procedure used by SOAR for reordering the conditions of all productions added to production memory. Section 10 discusses the advantages and disadvantages of joining the alpha branches of a Rete network in ways other than a linear chain. Section 11 describes a particular example in which the inclusion of more tests on the alpha branches of a network reduces the overall matching time. Section 12 describes a change to the OPS5 network compiler that allows productions to be integrated into the network during a run. Sections 13, 14, and 15 describe modifications that speed up individual node operations. Section 13 describes a way to eliminate some of the testing of the alpha branches through an indexing technique. Section 14 discusses the tradeoffs of using hash tables at the memory nodes and describes a particular implementation for SOAR. Section 15 describes several ways of speeding up the process of removing tokens from the network. Section 16 describes methods of eliminating the overhead involved in interpreting the Rete network, including compiling the network into executable code. Finally, Section 17 describes some of the other work done in speeding up the Rete algorithm, and Section 18 summarizes our results in speeding up the matching in OPS5 and SOAR tasks.

8. A Simplification of the Rete Network

One complication that arises because of sharing in the network is that alpha memory nodes may have outputs both to the left input of one and-node and the right input of another and-node (or not-node). Such a situation arises when the same condition appears as the first condition in one production and as any condition other than the first in another production. Since an and-node must know from which side a token has entered, each alpha memory node has separate lists of outputs to

the left input of an and-node and to the right input of an and-node.[5] For SOAR networks, the actual number of alpha memories which have outputs to both sides of and-nodes is small. For the six tasks in Appendix A, the number of such alpha memories ranges from 6 to 10, while the total number of alpha memories ranges from 94 to 350. Since this number is so small, one might consider unsharing these memory nodes so that every alpha memory node has outputs to only one side of an and-node (or not-node), thus simplifying the operation of the alpha memory node. Runs of the six tasks in Appendix A indicate that such unsharing either does not affect the matching time or speeds up the matching slightly. The speedup due to the simplification of the operation at the memory nodes at least compensates for the increase in processing time due to the loss of sharing. The change is also advantageous in that it simplifies the implementation of several of the other modifications to the Rete algorithm described below.

Given this change, it is possible to classify every memory node in the network as either a *left memory* or a *right memory*, depending on which side of two-input nodes its outputs go to. Note that all beta memories are left memories, and most alpha memories are right memories. However, some alpha memories—those that represent conditions that occur first in productions—are left memories. In some of the discussion below, it will be convenient to group memories into alpha and beta memories. At other times, it will be convenient to group them as left and right memories.

9. Ordering Conditions

In this section we discuss effects of condition ordering on the efficiency of the Rete algorithm and describe the particular algorithm used by SOAR to order conditions of productions. In Section 9.1 we describe three distinct effects that the ordering of conditions in productions can have on the efficiency of the Rete algorithm. In Section 9.2 we present the basis for and the broad details of the SOAR "condition reorderer". Finally, in Section 9.3 we evaluate the SOAR reorderer.

9.1. Effects of Ordering Conditions on the Rete Algorithm

The ordering of conditions in productions affects the efficiency of the Rete matching algorithm in several ways. First, the ordering of conditions in a production affects the number of partial instantiations (tokens) that must be computed in determining the instantiations of the whole production. Here, we define the number of partial instantiations of an n-condition production to be the sum of the instantiations of the first two conditions, the first three conditions, etc., up to the first n-1 conditions. Hence, the number of partial instantiations of a production is exactly the number of tokens that are

[5] In the OPS5 Rete algorithm, all memory nodes have the same implementation, so beta memory nodes also have two lists of outputs, even though they never have outputs to the right inputs of and-nodes.

stored in the beta memories associated with the production. The cross-product effect, mentioned in Section 6.2 above, occurs when conditions are ordered so badly that large numbers of partial instantiations are created, even though the production may have few or no overall instantiations. Specifically, the cross-product effect happens when several unrelated or nearly unrelated conditions occur consecutively in the conditions of a production. In that case, any (or almost any) combination of matches to each of the individual conditions will match the conjunction, so the number of matches to the conjunction will be the product of the number of matches to each of the individual conditions. Hence, any beta memory representing those conditions is likely to contain a large number of tokens.

An example of the cross-product effect and the effect of varying the ordering of a set of conditions is given in Figure 9-1. In the figure, two orderings of the same condition list are shown. The conditions match "persons" that each have an "age" attribute and a "father" attribute. The conditions are instantiated by two persons <p1> and <p2> (i.e. instantiated by wme's that represent the persons) if <p2> is the father of <p1> and the age of <p2> is the square of the age of <p1>. The figure gives the number of partial instantiations created for the two orderings if there are twenty persons and fifty numbers (whose square is represented) in working memory. Because the first two conditions of the second ordering have no variables in common, the number of instantiations of the two conditions is 400, the product of the number of instantiations of each of the individual conditions. On the other hand, in the first ordering, every condition after the first is linked to a previous condition through the variable in its identifier field, so there is no cross-product effect.

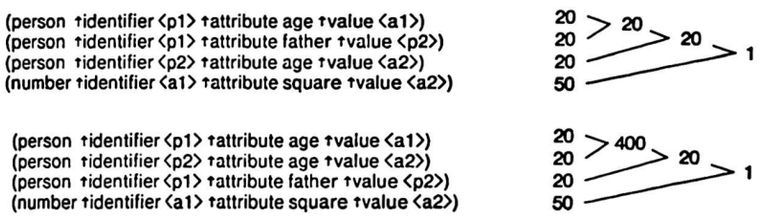

Figure 9-1: Illustration of the Cross-product Effect

The cross-product effect can easily be demonstrated by actual runs of SOAR tasks. For one eight puzzle run in which the conditions had been ordered by the SOAR reorderer, 7922 tokens were created and the matching time was 25 seconds. For five further runs in which the conditions of one of the eight puzzle productions were randomly ordered, the token counts were 65922, 29864, 31058, 64233, and 27795, and the matching time was 1520 seconds, 164 seconds, 233 seconds, 1510 seconds, and 227 seconds, respectively. Obviously, the overhead of the cross-product effect can be enormous if the conditions of a production are not ordered carefully.

A second effect that ordering can have on the Rete algorithm is that some orderings may have greater sharing of beta memories and and-nodes than others. If several productions have common sequences of conditions, then there will be greater sharing if the conditions of the productions are ordered so that these common sequences occur at the beginning of the condition lists.

Third, some productions may have some conditions that match objects that are relatively static, remaining unchanged in working memory for a long time, and other conditions that match objects that are relatively dynamic, with various attributes frequently changing. Recall, in OPS5, that any modification of a wme involves removing the current version of the wme from the network and adding the new version. If a production contains conditions near the beginning of its condition list matching a frequently changing object, then whenever that object is modified, many partial instantiations of the production (tokens) including that object will be removed and then recreated. This problem of changes in matches to the early conditions of productions forcing recomputation of all partial instantiations based on those conditions is called the *long-chain effect*. This problem can be somewhat reduced if conditions matching mostly static objects are placed at the top of the condition list, and conditions matching more dynamic objects are placed near the bottom of the condition list. In this way, fewer tokens are affected by the changes in the dynamic objects.

Unfortunately, the orderings of a condition list that are most efficient in each of these three respects are usually quite different. It is clear, though, that conditions should be ordered mainly to minimize the number of partial instantiations created and only secondarily to maximize sharing or to reduce the recomputation of tokens. The huge overhead that results from the cross-product effect far outweighs any duplication of processing in creating tokens that the latter two goals try to eliminate. In fact, a condition reorderer was added to SOAR mainly because of the cross-product effects caused by many of the chunks. Without the reorderer, the conditions of chunks created by SOAR have no particular order and often cause large numbers of partial instantiations, slowing down the system many times over. (Section 6 mentioned that the cross-product effect occurred in one of the tasks of analyzed in Appendix A. The task was the eight puzzle task with learning on, and the cross-product effect resulted because the conditions of some of the chunks were poorly ordered by the reorderer.) The condition reorderer also runs on all task productions loaded at the start of run, eliminating the need for the user to order the conditions of task productions "by hand."

9.2. The SOAR Condition Reorderer

The goal of the SOAR condition reorderer, then, is to order the conditions of a production so that, for a fixed number of instantiations of the production, the number of partial instantiations of the production is minimized. Clearly, the optimal ordering of the conditions of a production can vary as the contents of working memory and thus the instantiations of the individual conditions vary. Hence,

the reorderer is looking for an ordering that results in the minimum number of partial instantiations of the production, averaged over the likely contents of working memory for the task. Determining the optimal ordering strictly by this definition would be very complex and requires much information about the task that is probably not available. However, the best ordering can be approximated using static properties of the conditions that give information on the likely number of instantiations of the conditions and sets of the conditions.

The SOAR reorderer determines the an ordering incrementally, a condition at a time. That is, the reorderer builds up an ordered condition list out of the original unordered set of conditions by repeatedly determining the "best" condition remaining in the set of unordered conditions. This best condition is then removed from the set of unordered conditions and appended to the ordered condition list, and the cycle is repeated. The SOAR reorderer defines the best condition as the condition that will most likely add the most constraint to the condition list. That is, if there are already n conditions in the list of ordered conditions, the reorderer looks for an (n + 1)st condition such that, for a given number of instantiations of the n conditions, the likely number of instantiations of the n + 1 conditions is minimized. If there are more than one condition which are "best" according to the information available to the reorderer, the reorderer will sometimes "look ahead" to determine which choice among the tied conditions will allow the greatest constraint among the following few conditions added to the condition list.

In classifying conditions according to how much constraint they add to a condition list, we use the term *augmented condition list* to refer to the condition list which results from appending the condition being classified to the original condition list. Hence, the constraint that a condition adds to a condition list is defined in terms of the relation between the number of instantiations of the original condition list and that of the augmented condition list. Also, in the discussion below, we will refer to a variable in a condition as being *bound* with respect to a condition list if the variable occurs *positively* in the condition list. A variable occurs positively in a condition if the condition is itself not negated and the variable appears in the condition alone, without a predicate such as "<>" or ">=". For example, <x> appears positively in

```
(object ↑identifier <x> ↑attribute height ↑value 45)
```

but appears nonpositively in

```
(state ↑identifier <s> ↑attribute location ↑value <> <x>)[6]
```

and

```
- (object ↑identifier <x> ↑attribute height ↑value 45)
```

If a variable of a condition is bound with respect to a condition list, then the values of that variable in

[6] Note that the occurrence of "<> <x>" implies that there is a positive occurrence of <x> in a previous condition to which the "<> <x>" refers.

instantiations of the augmented condition list are limited to the values that the variable takes on in the instantiations on the original condition list.

Three kinds of information are available to the SOAR reorderer. First, it has information on the "connectivity" of the conditions being ordered. Specifically, it "knows" which variables of the remaining conditions are bound by the conditions already in the ordered condition list. Conditions in which all variables are already bound are more likely to add constraint than conditions that contain unbound variables in some fields. Also, since there are only a limited number of wme's with a particular identifier, a condition with a bound variable in the identifier field is likely to be more constraining than another condition with bound variables or constants in the other fields, but an unbound variable in the identifier field. Second, the SOAR reorderer has some SOAR-specific knowledge. For instance, it "knows" that conditions with class name "goal-context" match augmentations of existing goals. Since the conditions of a production are linked together (via their identifier fields) in a hierarchy rooted in the goal conditions, it is best for the goal conditions to be near the beginning of the condition list. Additionally, since there are usually only a few goals in the goal hierarchy at one time, goal conditions are likely to have only a few instantiations and so provide more constraint than other conditions, other things being equal. Third, the SOAR reorderer may be provided by the user with task-specific knowledge about which attributes of objects are *multi-attributes*. A multi-attribute is any attribute of a SOAR object which can have multiple values. Multi-attributes are used frequently in SOAR tasks to represent sets of objects. The user may provide for each multi-attribute the number of values that are likely to be associated with it.

Conditions may be classified broadly into two groups with respect to the amount of constraint they add to a particular condition list. First, there are conditions that add enough constraint that the number of instantiations of the augmented condition list is certain to be less than or equal to the number of instantiations of the original condition list. Second, there are all other conditions, for which the number of instantiations of the augmented condition list may be greater than that of the original condition list. In all further discussion, we consider only conditions which match augmentations. (The classifications below can all be extended without much difficulty to include preferences.) Then, based on the information available to the reorderer, the first group of conditions may be further divided into two classes:

- Conditions in which all fields contain constants or bound variables.
- Conditions in which the identifier field contains a bound variable[7] and the attribute field contains a constant which is not a multi-attribute.

The second group of conditions may also be subdivided into several classes:

[7] We should actually say "bound variable or constant," but the case of a constant in the identifier field is very infrequent.

- Conditions in which the identifier field contains a bound variable, and the attribute field contains a constant attribute which is a multi-attribute.
- Conditions in which the identifier field contains a bound variable, but the attribute field contains an unbound variable. Even if the value field contains a constant or a bound variable, there may still be more than one instantiation of the condition consistent with each instantiation of the original condition list (and so more total instantiations for the augmented condition list than for the original condition list).
- Conditions which have an unbound variable in the identifier field. Again, even if constants or bound variables appear in the attribute and value fields, the number of instantiations of the augmented condition list may be greater than the number of instantiations of the original condition list.

We now give in broad terms the definition of "best condition" used by the SOAR condition reorderer that is based on the above classification and the reorderer's knowledge about goal conditions. The best condition is the condition with the lowest rank in the following ranking:

1. Any condition with a bound variable in its identifier field, and constants in both the attribute and value fields.
2. Any condition with a bound variable in its identifier field and a constant or bound variable in both the attribute and value fields.
3. Any condition with a bound variable in its identifier field, a constant in the attribute field, an unbound variable in the value field, and a class name of "goal-context". As mentioned above, conditions matching goal augmentations are preferred to other kinds of conditions, other things being equal.
4. Any condition with a bound variable in its identifier field, a constant non-multi-attribute in the attribute field, and an unbound variables in its value field.
5. Any condition with a bound variable in its identifier field and a constant multi-attribute in its attribute field. The conditions within this subgroup are ordered further according to the number of values the multi-attribute is likely to have. (A default value is used if the expected number of values of a particular multi-attribute is not provided by the user.)
6. Any other condition with a bound variable in its identifier field and an unbound variable in the attribute field.
7. Any condition with an unbound variable in its identifier field and a class name of "goal-context." This aspect of the definition leads to a goal condition, if there is one, being chosen as the first condition of the ordering (when no variables are bound.)
8. Any other condition with an unbound variable in its identifier field.

(As should be clear from their definitions, ranks 1 through 4 are further subdivisions of the first group above, while ranks 5 through 8 cover conditions in the second group.)

If there are multiple conditions with the lowest rank, the tie is broken randomly for all ranks except 4 and 5. For ties with ranks 4 and 5, the tying conditions are evaluated by a look-ahead in which each condition is temporarily added to the condition list and the remaining conditions are reclassified. The best tying condition is the one which, if added to the condition list, minimizes the the rank of the next

condition added to the condition list. There may still be tied conditions after this one-step look-ahead, in which case the look-ahead is extended even further.

9.3. Results

Using minimal task-specific information (namely, information on multi-attributes), the SOAR condition reorderer described above does a fairly good job ordering conditions in order to minimize the cross-product effect. From limited observation, the reorderer almost always produces orderings which are as good or nearly as good as the orderings which would be made "by hand" by the user who understood the meaning and intended use of the productions. For example, for the same eight puzzle run in which 7922 tokens were created when conditions were ordered by the SOAR reorderer, the token count was only reduced to 7859 after several attempts to find better orderings by hand. There are, however, some cases when the SOAR reorderer orders the conditions of chunks badly. These cases suggest that the reorderer should do a better lookahead when there are several connected conditions with multi-attributes to be ordered. Also, there are some cases when a particular ordering can produce a savings by increasing sharing or reducing the long-chain effect, without causing the cross-product effect. The reorderer misses such orderings, since it has no heuristics or information for determining when such orderings may be advantageous. At this point, these cases do not seem to be frequent enough or easy enough to recognize that it is worth modifying the reorderer to handle them.

The problem of ordering the conditions of a production so as to minimize the number of partial instantiations is one instance of a problem that is frequent in database and artificial intelligence work. A *conjunctive problem* may be defined as a set of propositions (or relations or patterns) which share variables and must be satisfied simultaneously. The work required in determining instantiations of such conjunctions often depends on the ordering of the conjuncts. Smith and Genesereth [17] have characterized several methods for determining a good ordering of the conjuncts. The SOAR reorderer may be described as following the "connectivity" heuristic in ranking conditions according to the occurrence of bound variables in the conditions. It also uses a "cheapest-first" heuristic in choosing goal-context conditions (which are likely to have few instantiations) over other conditions, and in ordering conditions with constant attributes according to the expected number of values of the attribute. By combining these two heuristics, the reorderer appears to avoid some problems with using either alone. Additionally, the SOAR reorderer has a heuristic to cover a problem not handled by either of the other two heuristics, that of choosing the first condition on which to "anchor" the rest. The SOAR reorderer "knows" that it is best to start off a condition list with a goal condition, since such conditions are likely to have few instantiations and are usually the root conditions to which the rest of the conditions are linked.

10. Non-linear Rete Networks

In this section we discuss the use of networks with structures other than the linear structure of the standard Rete network. We give only a general discussion of the advantages and disadvantages of using non-linear networks, since there has been only a preliminary investigation of their use for SOAR productions.

The standard Rete network has a fixed linear structure in that each alpha branch representing a condition of a production is successively joined to the network representing the conjunction of the previous conditions. It is possible for the conditions to be joined together in a non-linear fashion. For example, a binary Rete network for a four-condition production can be built by joining the alpha branches representing the first two conditions, joining the alpha branches representing the last two conditions, and then joining the results. The functioning of the and-nodes is the same in non-linear networks as in linear networks, except that the tokens from the right input may have any size, rather than a constant size of 1. Figure 10-1 illustrates several possible network structures. The last of the three is referred to as *bilinear*, because each of several groups of alpha branches are joined together in a linear fashion, and the resulting subnetworks are themselves joined together in a linear network.

The ordering problem of Section 9 can now be seen as a special case of a more general problem of determining the optimal structure of the Rete network for joining the alpha branches of the conditions of a production. In Section 9, the linear structure of the standard Rete network was assumed, so the problem reduced to finding the best order in which the alpha branches should be successively joined.

The possible effects of differing network structures on the efficiency of the Rete algorithm are the same as the effects described in Section 9.1 of reordering conditions within the linear network structure. First, the structure of the network affects the number of partial instantiations that must be computed for a production. Thus, if four conditions are joined by a linear network, the partial instantiations that are created and stored in the beta memories are the instantiations of the first two conditions and the instantiations of the first three conditions. If the conditions are joined by a binary network, the partial instantiations are the instantiations of the first two conditions and the instantiations of the last two conditions. As Figure 9-1 illustrates, any change in the way in which conditions are joined can have a great effect on the number of partial instantiations that are computed.

Second, varying the structure of the network can vary the amount of sharing of and-nodes and beta-memories between productions. In linear networks, sharing at most and-nodes, except those near the beginning of a linear chain, is very unlikely because it requires that the preceding alpha branches of the chain be shared. Thus, any conditions common to several productions must appear at the beginning of their respective condition lists for the and-node(s) that joins them to be shared.

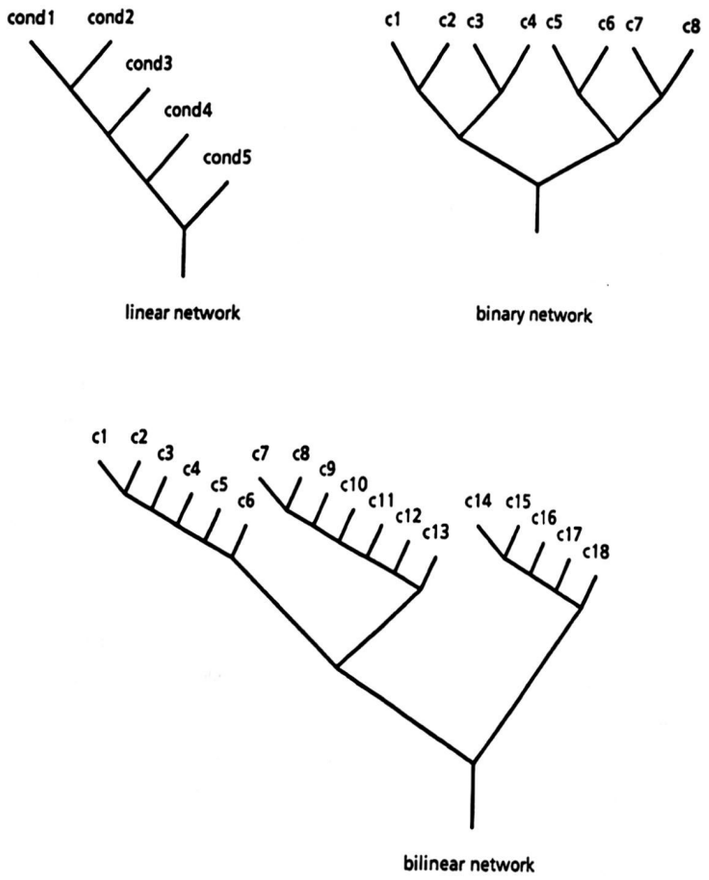

Figure 10-1: Several Possible Structures for the Rete Network

Non-linear networks allow greater sharing by breaking up the long linear chains into smaller subnetworks which can each be shared. In fact, there can be and-node sharing within a single production if it has several repeated conditions. For example, consider the condition list:

```
(person ↑identifier <p1> ↑attribute father ↑value <f1>)
(person ↑identifier <f1> ↑attribute name   ↑value <nm>)
(person ↑identifier <p2> ↑attribute father ↑value <f2>)
(person ↑identifier <f2> ↑attribute name   ↑value <> <nm>)
```

An instantiation of such a condition list would yield a pair of persons <p1> and <p2> whose fathers have different names. In the Rete network for the condition list, the first and third conditions would

share an alpha branch (since they have the same sequence of intra-condition tests), as would the second and fourth conditions. If a binary network is used for the condition list, there is further sharing of the and-node joining the first two conditions and the and-node joining the second two conditions.

Third, the use of a non-linear network can eliminate the long-chain effect that is characteristic of linear networks. For instance, if the conditions of a production matching distinct objects are placed in separate subnetworks of a bilinear network, then changes to one object will affect only the tokens in the subnetwork matching that object, and will not affect matches to different objects in the other subnetworks. One case in which the use of a non-linear network is particularly advantageous is for a production in which several of the conditions are intended to match one or more permanent objects in working memory. For example, one production of the eight puzzle task determines if the overall goal has been achieved by comparing the tile configuration of the current state with the tile configuration of another object in working memory representing the desired state. If a non-linear network is used for matching this production, then the conditions matching the desired tile configuration can be placed in a separate subnetwork, and the match to the desired state is only done once. However, if a linear network is used, the work of matching the conditions representing the desired state is recomputed often when the matches to preceding conditions in the production change.

Unfortunately, the cross-product effect occurs more frequently for non-linear networks than for linear networks. With linear networks, because each successive condition adds more constraint to all previous conditions, there is almost always an ordering of the conditions that avoids the cross-product effect. However, in non-linear networks in which conditions are joined first in smaller groups before all being joined together, it is sometimes impossible to get enough constraint in the subnetworks to avoid the cross-product effect. The occurrence of the cross-product effect often cancels out any of the benefits that may result from using non-linear networks to increase sharing or avoid the long-chain effect.

A logical network structure for SOAR productions would be a bilinear network in which each linear subnetwork contains the conditions matching one SOAR object. Such a network structure would allow sharing of similar groups of conditions in different productions that match the same type of SOAR object and would make the match of each object independent of the other objects in the production. The linear subnetworks could be thought of as "super alpha branches," since they completely match SOAR objects, just as alpha branches completely match OPS5 objects. The difference between such alpha branches and "super alpha branches" is that many partial instantiations may be created for each match to the "super alpha branch," whereas only one token is created for each match to an alpha branch. The disadvantage of such a bilinear network structure is that the object subnetworks may match many objects that are immediately ruled out by other conditions of the production in a linear network. Much of the constraint in linear networks of conditions results from

the intermixing of the conditions representing the different objects. That source of constraint is lost if conditions representing different objects are put in different subnetworks.

Hence, it is not clear whether non-linear networks are in general useful for speeding matching of SOAR productions.[8] There are some special cases in which it is advantageous to build a non-linear network for a production, but there is no obvious general criteria for determining when non-linear networks are useful.

11. Putting More Tests on the Alpha Branches

In Section 6 we indicated that the ratio of matching tests done at the and-nodes to tests done by the alpha-nodes is much greater for Rete networks built from SOAR production sets than for OPS5 production sets. The "joining" operation of the and-node is more complex than the simple tests of the alpha nodes. Hence, it seems likely that any inclusion of extra tests on the alpha branches that reduces the number of tokens entering and-nodes (but does not eliminate any legal instantiations of productions) will result in an overall speedup of the matching process.

One example of this possibility for speeding up the match has to do with conditions which match augmentations of goals. A newly-created goal initially has augmentations that indicate that the problem-space, state, and operator of the goal are undecided:

```
(goal-context ↑identifier g00005 ↑attribute problem-space ↑value undecided)
(goal-context ↑identifier g00005 ↑attribute state         ↑value undecided)
(goal-context ↑identifier g00005 ↑attribute operator      ↑value undecided)
```

As the problem-space, state, and operator are determined by the decision procedure, these augmentations are removed, and new ones created with the decided values. In a production fragment such as:

```
(p eight*apply-move-tile
    (goal-context-info ↑identifier <g> ↑attribute problem-space ↑value <p>)
    (space-info ↑identifier <p> ↑attribute name ↑value eight-puzzle)
    (goal-context-info ↑identifier <g> ↑attribute state ↑value <s>)
```

it is clear that <p> is not intended to match "undecided". However, the first condition will match any problem-space augmentation of a goal, even if the value field contains "undecided". It is only by the conjunction of the first two conditions that the possibility that <p> matches "undecided" is eliminated, since there is no wme with identifier "undecided". Hence, the processing of the and-node joining the first two conditions can be reduced by including an extra test on the alpha branch that ensures that <p> does not bind to "undecided." This test can be added by changing the first condition to:

[8] Non-linear networks are definitely useful in SOAR for one reason unrelated to efficiency. A non-linear network is required to represent a production that contains a conjunctive negation (negation of a conjunction of several conditions). The ability to express conjunctive negations is especially needed in SOAR, given SOAR's representation of objects.


```
(goal-context-info ↑identifier <g> ↑attribute problem-space
                                   ↑value { <> undecided <p> } )
```

The braces indicate that the "<> undecided" test and the <p> variable binding both apply to the value field of the condition.

Every SOAR production begins with several conditions matching goal augmentations. If a "<> undecided" test is added to all conditions with a variable in the value field that match problem-space, state, and operator augmentations, the effect is surprisingly large. For the eight puzzle run of 143 decision cycles, the number of tokens that are created is reduced from 22242 to 15275, resulting in an 18% speedup from 33 seconds to 27 seconds. The addition of the "<> undecided" tests can be done by the production compiler, so that the user need not worry about them.

12. Modifying the Rete Network During a Run

Once wme's have been added to working memory, the memory nodes of the network contain tokens indicating partial matches involving the current elements of working memory. If a new production is then added to the network, any unshared memory nodes of the new production will be empty and must be "primed", i.e., updated to contain tokens representing partial matches of the current working memory contents to the new production. If the memory contents are not updated, then the tokens that should be in the memory and all tokens further down in the network that are based on these tokens will never be created. Hence, if the memories are not primed, any instantiations of the new production which are based on these tokens will never be created and added to the conflict set. OPS5 compiles in the normal way productions that are added to production memory during a run, but it does not prime the unshared memories of the new production. This is not sufficient for SOAR, which must have the capability to add chunks to production memory so that they match working memory properly.

If there were no sharing of network nodes, the priming could be done simply by saving a pointer to each alpha branch as it is built and, after the whole production is compiled, sending each wme in working memory down each of the new alpha branches in turn. Previously, SOAR took a related approach. The nodes of all productions loaded in at the beginning of a run were shared. However, the network structure built for each chunk during the run was not shared with the other productions, so the simple procedure described above could be used. Thus, no chunk was shared with any of the original productions or any other chunks. This loss of sharing increases the number of nodes in the network and the average number of tokens in the network. Such loss of sharing is especially significant because chunks tend to be complicated productions, longer than most of the original productions, and because often a pair of chunks are either completely identical or are identical up to the last few conditions.

The procedure for priming new memory nodes when the newly added productions are shared with the existing network is slightly more complicated, though it requires less processing, since shared memory nodes (i.e. nodes used in building the production that already existed before the new production was compiled) need not be updated. To describe the procedure, we first need to define the term *beta branch*. The beta branch of an and-node, not-node, or beta memory is the unique path from that node to the bus node which follows the left-input of the two-input nodes and the (single) input of all other nodes. This path passes through a sequence of and-nodes, not-nodes, and beta memories and then follows the alpha branch of the first condition of a production to the bus node.

Because not-nodes contain memories, they must be primed as well as alpha and beta memories. Priming occurs as the network of a new production is being built, whenever a new (i.e. unshared) memory node or not-node is created. A new alpha memory is primed by sending each wme in working memory down the alpha branch leading to the new alpha memory, stopping at the alpha memory (if the wme gets that far). A new beta memory or not-node X is primed as follows. If there is no other memory node on the beta branch above X, then all wme's in working memory must be sent from the bus node down the beta branch to the new node X. Otherwise, all tokens in the memory node (or not-node) M lowest on the beta branch above X must be sent from M to X. M is already primed either because it was shared or because it was already updated earlier in building the current production. (The two cases of whether or not there is a memory node on the beta branch above X can be combined if the bus node is thought of as a memory node containing all wme's currently in working memory.)

The importance of this modification of the OPS5 Rete algorithm is shown by its effect on the running of the eight puzzle task with learning on. The size of the network resulting from the default and eight puzzle productions is 1051 nodes and would be 3347 nodes if there were no sharing. During one run of the eight puzzle with learning on, eleven chunks are created, increased the total "unshared" size of the network to 4926 nodes. If there is no sharing among the original network and the new networks created for each of the chunks, the total number of actual nodes in the network increases to 2091, and the average number of tokens in the network is 3297. If the new chunks are integrated into the original network, the network size only increases to 1469 nodes and the mean number of tokens in the network is only 2180. The result is a speedup of the matching time for the run from 188 seconds to 166 seconds, a reduction of 12 percent. The speedup would be much greater for longer runs in which the ratio of the number of chunks created to the number of initial productions is larger.

13. Using Indexing to Eliminate Alpha Tests

By convention, all OPS5 conditions begin with a constant class name. Since the OPS5 production compiler creates the test nodes of alpha branches in the order of the tests in the condition, a test for a particular constant class name is always the first test on the alpha branch of that condition. In SOAR, not only do all conditions begin with a constant class name, but most conditions have the following form:

(constant-class-name ↑IDENTIFIER variable ↑ATTRIBUTE constant-attribute-name ↑VALUE value)

Such a condition matches the most common type of wme, the augmentation. The variable in the identifier slot has most likely appeared in previous conditions and almost certainly does not appear anywhere in the value field. Hence, it is not involved in any of the tests on the alpha branch corresponding to the condition. The attribute field of a SOAR condition is almost always a constant attribute name, since it is usually possible and desirable to eliminate variable attributes in conditions by using an extra level of indirection in storing information about the object.[9] In the typical case, then, of a constant class name, variable identifier, and constant attribute name, the tests on the condition's alpha branch consist of:

1. A test that the class name of the wme is constant-class-name
2. A test that the value of the attribute field of the wme is constant-attribute-name.
3. Any intra-condition test(s) associated with the value field. (E.g. matching the value field to a constant.)

Because the class-name test is the first on each alpha branch, this test can always be shared between any conditions with the same class name. Hence, because of sharing, there are actually only as many alpha branches leaving the bus node as there are class names, though each of these branches may separate into many alpha branches below the class name test. Any wme being filtered down the network will only satisfy one of these class name tests. The filtering process may be sped up, then, by building a table which indicates which alpha branch should be followed for each class name.

A similar technique can be used to speed up the attribute-name tests immediately below class-name tests. Again, because of sharing, below any class-name test for an augmentation class, there is one branch for each possible attribute name for that class, plus possibly other branches corresponding to conditions which don't contain a constant attribute name. While all of these other branches must be traversed as usual, only one of the constant attribute branches need be traversed, as indicated by a table indexed by the attribute names.

---

[9] There are some cases when variable attributes are both useful and needed. For example, variable attributes are necessary to reference attributes which are themselves SOAR objects with various attributes.

One simple method in LISP of implementing a table that maps class names to the corresponding alpha branch is to put a pointer to each alpha branch on the property list of the corresponding class name. Such a method uses LISP's symbol table (often a hash table) to achieve the mapping. Also, the table for each class name which maps attributes into branches can be represented in LISP by an association list, also on the property list of the class name. When indexing of class names and of attributes is implemented in this way, the matching time for the eight puzzle run is reduced from 107 seconds to 95 seconds, a decrease of 11 percent. The number of alpha branch operations is reduced from 60811 alpha node tests to 6234 indexing operations and 5481 alpha node tests.

14. Using Hash Tables at Memory Nodes

In this section we discuss the possibility of speeding up the Rete algorithm by storing tokens at memory nodes in hash tables rather than in simple lists. First we discuss the general advantages and disadvantages of using hash tables. Next we discuss an implementation for networks built from SOAR production sets which avoids some of the disadvantages. Finally, we give timing results on speedup of SOAR tasks when hash tables are used.

14.1. Advantages and Disadvantages of Hash Tables

Using hash tables at some or all of the memory nodes holds promise for speeding up two aspects of the Rete matching algorithm. First, removal of tokens from memory nodes will be faster if hash tables reduce the average number of tokens that must be examined before the one to be removed is found. Second, storing the tokens in hash tables could speed up processing at the and-nodes significantly. When a token comes down one input of an and-node, a sequence of and-tests is executed between that token and each token in the memory node at the other input. Suppose the tokens in the opposite memory node have been stored in a hash table according to some field of the tokens which is involved in an equality test at the and-node. Then, only tokens in the opposite memory that have the appropriate hash value (according to the equality and-test) need be checked to see if they satisfy the and-tests. For example, the tests at the and-node joining the two conditions

```
(goal-context ↑identifier <g> ↑attribute problem-space ↑value <p>)
(goal-context ↑identifier <g> ↑attribute state ↑value <s2>)
```

to the condition

```
(preference ↑object <s2> ↑role state ↑value acceptable ↑goal <g>)
```

are that the object attribute of the wme from the right (the preference) must equal the value attribute of the second wme of the token from the left, and the goal attribute of the right wme must equal the identifier attribute of the first wme of the left token. Suppose the tokens in the right memory are stored in a hash table according to the value of their object attribute (the <s2> field), and a token comes down on the left whose second wme has a value s0008 for its value field (i.e. for <s2>). Then,

only those tokens in the right hash table that have the hash value for s0008 need be checked for possible matching with the new left token. For example, if bucket hashing is used and s0008 hashes to bucket 8, then only wme's in bucket 8 of the right memory need be tested. Tokens in any other bucket could not possibly have s0008 for their object attribute. Similarly, if the tokens in the left memory were stored according to the value of their "<s2>" or "<g>" field, then the value of the "<s2>" or "<g>" field in a token coming down from the right could be used to pick out the particular bucket in the left hash table in which matching tokens must be.

One possible problem with using hash tables at the memory nodes is that the overhead involved in hashing (computing the hash value, doing a table lookup, handling collisions) may use up more time than is saved by speeding up the remove and and-node operations. In the OPS5 Rete algorithm, the process of adding a token to a memory node consists of just adding the token to the front of a list, so that process is definitely slowed down when hash tables are used at memory nodes. Hopefully, the speedup in the remove and and-node operations will more than compensate for the additional overhead of hashing.

Another overhead resulting from the use of hash tables occurs if a memory of an and-node uses a hash table, but, for some reason, no field of a token from the opposite input can be used to eliminate from consideration all wme's except those with a certain hash value. In this case, the entire contents of the hash table must be enumerated so that all the wme's in the memory can be tested. Because much of the hash table may be empty, this process of enumerating all the wme's in the memory node can take significantly longer than if the tokens in the memory node are stored in a simple list.

14.2. An Implementation of Hashing for SOAR Tasks

For general OPS5 tasks hashing would be disadvantageous at many of the memory nodes for several reasons. First, an and-node may have no equality and-tests, so there would be no field that tokens from either memory could be hashed on to reduce the processing at that particular and-node. Second, because of sharing, memory nodes (especially alpha memories) may have several and-nodes as outputs. In general, each of the and-nodes will require that the tokens at the memory node be hashed on a different field. Hence, unless sharing is reduced so that each set of and-nodes that require hashing on a particular field have their own memory node, then the use of the hash table will speed up the processing at only some of the and-nodes. At the other and-nodes the entire hash table would have to be enumerated, which, as explained above, would slow down the and-node processing considerably.

SOAR productions have several properties that make hash tables more usable and advantageous. The most important property is that nearly every condition in a SOAR production (except the first) is linked to a previous condition through its identifier field. That is, almost every condition has a variable

in its identifier field which appears somewhere in a previous condition. Thus, almost every and-node will have an and-test which tests that the identifier field in a wme from the right input is equal to some field of a token from the left input. Hence, for hash tables at the right memories of and-nodes, wme's should always be hashed on the value of their identifier field. For hash tables at the left memories, tokens should be hashed on the field that must match the identifier field of the right token (wme). With this scheme, it is only for left memory nodes that we run into the problem of and-nodes that share a memory requiring tokens to be hashed according to different fields. Since, as indicated by the statistics in Appendix A, only about 10-15% of beta memories (which make up most of the left memories) have more than one output, the loss of sharing that would be required in order to use hash tables at left memory nodes would not be great.

The type of hash table that should be used seems rather clear from a couple of considerations: (1) the overhead for access to the table should be low and should degrade gracefully as the table becomes more and more filled; (2) it should be be easy to access all of the elements in the table with a particular hash value. The appropriate type of hash table then seems to be a hash table with buckets represented by linked lists. With bucket hash tables, the need to expand hash tables and rehash when they are full is avoided, and it is simple to pick out all the tokens that hash to a particular value.

Another issue is whether each memory node should have a local hash table, or there should be one global hash table in which all tokens from "hashed" memory nodes are stored. If each memory has its own hash table, it would be best to vary the size of the table in relation to how filled the memory will get. However, unless statistics are saved from previous runs, this information is not available when the network is built, so each memory node must be given a hash table of the same size. (It is known that alpha memories in general get more filled than beta memories, so it is probably best for alpha memory hash tables to be bigger than beta memory hash tables.) The use of a global hash table avoids the problem of having an uneven distribution of tokens in hash tables at the individual memory nodes, so the size of the global hash table can be significantly smaller than the sum of the sizes of hash tables at all the memory nodes. On the other hand, if a global hash table is used, then the hash function by which tokens are stored would have to be based on the particular memory node for which the token is being stored, as well as on the value of a field of the token. Also, each token stored in the table must be tagged to indicate which memory node it comes from, and these tags must be examined during hash table operations to ensure that a particular token in the hash table comes from the memory node of interest. Hence, the tradeoff between local hash tables and a global hash tables is that the overhead for accesses is lower when local hash tables are used, but less storage is required if a global hash table is used.

14.3. Results

The implementation that was was actually tested was local bucket hash tables. Timings were obtained for three variants in which hash tables were used only at left memories, only at right memories, or at all memories. The results indicate that hash tables are most effective at right memory nodes. In fact, the matcher slowed down with hash tables at the left memories. Specifically, for an eight-puzzle run in which matching time was 70 seconds without the use of hash tables, the matching time was 82 seconds with hash tables at the left memories alone, 55 seconds with hash tables at the right memories alone, and 63 seconds with hash tables all memories. When hashing at left memories is used, the loss of sharing increases the size of the network by seven memory nodes out of 1068.

To some extent, the timings obtained depend on the relative speeds of array references and other operations that the use of hash tables requires versus the speed of the and-test operations that are eliminated by the hashing. For some LISP implementations, array referencing may be much faster in relation to list and test operations, and for these, hashing at left memory nodes may in fact speed up the algorithm. In any case, it is clear that it is particularly advantageous to hash at right memory nodes.

Besides the loss of sharing when left memory hashing is used, the reason the results are better for right memory hashing than for left memory hashing seems to be that right memories (most of the alpha memories) get more filled than left memories (mostly beta memories). If a memory has only a few elements, the overhead of hashing is larger than the speedup gained for removes and and-node operations. In fact, though most tasks sped up by more than 20% with right memory hashing, one SOAR task did not speed up at all for the first 100 decision cycles, because the average size of the right memory when a token entered the left side of an and-node was only about 2.8. The task did speedup for the second 100 cycles, when the average size of the right memory on an and-left call was about 6.1.

A variant of strict bucket hashing was found to speed up and-node processing even further when right memory hashing is used. In this variant, each bucket contains a number of lists, rather than a single list of tokens. Each of the lists of the bucket is headed by a hash key that hashes to that bucket and contains all the tokens that have that particular hash key. With this arrangement, the number of tokens that must be tested is reduced even further, since only tokens with the key required by the token entering on the left need be tested against that token. For example, using the example given in Section 14.1, suppose the tokens in the right memory are stored in this variant of a bucket hash table according to their <s2> field. If a token entering on the left has a value of s0008 for its value field and s0008 hashes to the 8th bucket, then only tokens in the s0008 sublist of the 8th bucket need be tested against the entering token. This variant of bucket hashing simulates the case in which all possible keys hash to a different bucket, but uses a smaller hash table at the cost of a linear search through

the sublists of the buckets. The results indicate that the increased processing required to maintain the sublists of the buckets is more than compensated by the reduced number of and-tests that must be executed. For the same eight-puzzle run given above in which the match time was 70 seconds without hash tables and 55 seconds with right memory hashing, the matching time with right memory hashing using this variant was 46 seconds, yielding an additional speedup of 16% and an overall speedup for hashing of 34%.

15. Speeding Up Removes

From a programming and algorithmic standpoint, it is convenient and even elegant that the operation of the Rete algorithm, except for the action of the memory nodes, is identical for the addition of wme's to working memory and for the removal of wme's from working memory ("removes"). However, from the standpoint of efficiency, the and-tests executed during a remove are redundant. For each set of and-tests executed between two tokens during a remove, the same and-tests were executed earlier, with the same result, whenever the newer of the two tokens was created. Therefore, one possibility for speeding up removes is to save the outcome each time and-tests are executed to determine if two tokens are consistent. If the outcome of each such consistency test between token A and another token is stored with token A in its memory node, then none of these consistency tests will have to be executed when token A is removed. Unfortunately, such a system has many obvious disadvantages, including the large amount of extra storage required to store the outcomes. Another possibility for speeding removes is to maintain for every wme in working memory a list of the locations of all tokens which contain that wme. Such information would make removes very fast, since the tokens containing the removed wme could be removed from their memory nodes directly without the necessity of traversing the network. Again, however, this system has a large storage overhead. Additionally, the processing overhead whenever a token is created is greatly increased, since a pointer to the memory node where the token is stored must be saved for each wme in the token.

A final possibility that requires no storage of extra information eliminates almost all processing at and-nodes during removes at the cost of increased processing at the memory nodes. Consider the situation when a token T enters an input of an and-node A during a remove operation. The beta memory B below the and-node contains one or more tokens that are the result of concatenating token T with another token, if T is consistent with tokens in the opposite memory of A, according to A's and-tests. Such tokens in B may be identified and removed by scanning B's memory for all tokens which contain T as a left or right subpart. Hence, with this modification to the Rete algorithm, whenever a token T enters one of the inputs of an and-node A during a remove, it is immediately sent on to each of the outputs of the node. Each memory node B below the and-node receives T, which has size one less than the size of the tokens that are actually stored at B, and removes all tokens that

contain T as a left or right subpart. Each such token that is removed is sent on to the outputs of B, causing further removes. If no tokens are found with T as a subpart, then T was not consistent with any of the tokens in the opposite memory of A, so no tokens need be removed from B, and the traversal of the network backs up right away.

This change to the Rete algorithm eliminates (for removes) the consistency tests between a token entering an and-node and all the tokens in the opposite memory, but requires that the contents of the memory node(s) below the and-node be scanned. The scanning operation is much simpler and almost always faster than executing the consistency checks between tokens at the and-node, especially since beta memories do not tend to grow very large. The modified algorithm is especially efficient when a token is consistent with many tokens in the opposite memory, since all the resulting tokens can be removed from the beta memory below the and-node in one sweep of the memory, rather than one at a time. For one run of the eight puzzle, this modification to the remove process decreased the matching time from 46 seconds to 33 seconds, a speedup of 28%.

16. Eliminating the Interpretive Overhead of the Rete Algorithm

The Rete network is essentially a declarative representation of a program which must be interpreted by the Rete matching algorithm each time a wme is added to or removed from working memory. Its structure makes it easy to add new productions to production memory and to determine what tests that are common to more than one production can be shared. However, much of the actual processing time of the OPS5 Rete algorithm is taken up with interpreting the Rete network, rather than with actually executing the matching and storage operations. One of the manifestations of this interpretive overhead is the significant percentage of the matching time taken up by the APPLY function. LISP's APPLY function is used by the OPS5 Rete algorithm to map test and node names contained in the network into functions that actually execute the node and test functions. For one run of the eight puzzle task, the APPLY function alone used up 10% of the matching time. Other overhead in interpreting the Rete network includes accessing the information in the nodes, iterating through lists of node outputs or and-tests, and transferring control between the functions that execute the nodes and tests. All this interpretive overhead could be eliminated by compiling the network. Ideally, compilation would yield a function or set of functions which explicitly contained all the tests and memory operations represented by the nodes of the network, such that the operations are executed in the same order as they are in the depth-first traversal of the network.

As with most programming systems, the speedup gained through compiling is balanced by a number of disadvantages. First, the compiling process may be complex and time-consuming, greatly lengthening the initial production loading process. Second, the resulting compiled code may be difficult or impossible to modify as productions are added to or removed from production memory,

forcing a large recompilation to be done for any changes to the production set. Such a disadvantage is crucial for SOAR, since chunks are added to production memory during runs. Third, since each node must be expanded into code that executes the functions of that node, the size of the code compiled to execute the network will certainly be bigger than the size of the memory required to represent the network. In Section 16.1 we describe a method of eliminating much of the interpretive overhead of the LISP OPS5 Rete algorithm without compiling the network. In Section 16.2 we discuss the issues involved in actually compiling the network to executable code and present the results for several implementations.

16.1. Reducing the Interpretive Overhead Without Compiling

Much of the overhead of interpreting the Rete network can be eliminated without actually compiling the network, thereby avoiding the disadvantages described above. The APPLY calls can be replaced by more specific code that executes faster. APPLY is a general function which takes any name and executes the function with that name. However, there are only a small, fixed number of node names and test names in the network, so the APPLY calls can be replaced by code in the matcher which specifically tests a node name or a test name for each of its possible values and calls the corresponding function. Essentially, the mappings between a node or test name and the corresponding function are "hard-wired" into the matching code. Other changes that are even more dependent on the particular implementation of the OPS5 Rete algorithm[10] can be made to reduce the interpretive overhead. The result is a large speedup for all tasks. For one eight-puzzle run, the matching time went down from 95 seconds to 74 seconds, a speedup of 22%.

16.2. Compiling the Network

One issue in compiling the Rete network into executable code is whether to compile the network directly into native machine code or into LISP code, which is then compiled into machine code by the LISP compiler. Obviously, the first approach provides the means for the greater efficiency, since the code for the node operations can be optimized for the machine instruction set. However, it also makes the network compiler machine-dependent. The implementations described here compile the network to LISP code, in part because there was no documentation available for the Xerox 1132 workstation on how to produce machine code from LISP.

Another issue is the complexity of the node operations. In the compilation process, every node is compiled into a piece of code that directly executes the operations of the node. Hence, code to do the and-node operations is duplicated many times over in the resulting compiled code, though with each instance specialized to contain the and-tests, output calls, etc., of a particular and-node. Because the code is duplicated so many times, it is desirable to keep the code that executes the and-node operations simple. However, changes to the Rete algorithm—such as the use of hash tables at the memories—complicate the and-node processing. Hence, there is a tradeoff between using simple and-node code in order to keep down the size of the compiled code or using more complex and-node code which takes advantage of other speedups besides compiling.

Another significant issue is the number of LISP functions into which to compile the network. One consideration is that the larger the number of functions into which the network is compiled, the greater the overhead in calling between node functions. An opposite consideration is that the greater the number of node functions, the smaller the amount of recompilation that must be done when a new production is added to the network, since only the functions corresponding to new nodes or nodes that have been changed (by the addition of new outputs) must be compiled. A third consideration is that it is simplest if each two-input node is compiled into a different function. The two-input nodes must maintain some state information while tokens that they create traverse the subnetwork below them. For instance, a two-input node must maintain information on the token that activated it, on which input the token entered, and on which token in the opposite memory is being processed. Such information is most easily maintained in a few local variables in separate functions which execute each two-input node.

Based on these considerations, a compiler was built that compiles the network in either of two slightly different ways. In the first approach, every node of the network is compiled into a separate function. In the second approach, all of the alpha branches are compiled into a single function, and each two-input node and the memory below it (if there is a memory node below it) are compiled into a single function. The compiler also allows a choice of whether or not the compiled code uses hash tables at the right memories.

The speedups obtained by compiling the network are summarized in Table 16-1.

	No Hash Tables	Hash Tables
Interpreted Network	74 sec	46 sec
Compiled Network	63 sec	39 sec
% speedup	15%	15%

Table 16-1: Times for Interpreted vs. Compiled Networks

Regardless of whether hash tables were used or not, there was no measurable difference in matching time between code in which all nodes were compiled into separate functions and code in which all the alpha branches were compiled into a single function. Hence, that factor is not included in the table.

[10] Such as reducing the number of function calls made by the interpreter and making the code that iterates over the outputs of a node more efficient.

The times given in the "Interpreted Network" line are for the Rete algorithm modified as described in Section 16.1 to reduced interpretive overhead. The speedup obtained from compiling the network is much less than was expected. Clearly, the changes of Section 16.1 eliminate the majority of the interpretive overhead of the Rete algorithm that compiling the network is intended to eliminate. Specifically, using the timings from Section 16.1 and the "No Hash Table" column of the table, the modifications to reduce the overhead of interpreting the Rete network give (95-74)/(95-63) = 66% of the speedup that results from compiling the network.

The code size of the compiled network was large, even for small production sets. The eight puzzle network contains 1048 nodes and can be represented by about 7,000 to 10,000 bytes depending on the encoding scheme. The size of the code that executes the eight puzzle network on the Xerox 1132 workstation is 75,000 to 90,000 bytes, depending on how many functions into which the network is compiled and whether hashing is used at the right memory nodes. Though no analysis has been done, the large size of the code probably adversely affects its execution speed, since there is only a limited amount of physical memory available for the network code, LISP system functions, and the storage of the working memory elements and tokens.

Because compiling the network only yields a small additional speedup and has many disadvantages, including large code size and long compiling time, this method of speeding up the Rete algorithm was not pursued further.

17. Related Work

A variety of work has been done in the area of speeding up production systems. One frequent approach has been has been to index certain features or combinations of features of conditions to allow quick determination of the productions that may be matched by the contents of working memory. Such indexing methods [12] have differed in the amount of indexing done and the amount of information stored between production cycles. Forgy [2] summarizes several of these indexing schemes. The Rete algorithm may be viewed as one variation in which conditions and productions are completely indexed by a discrimination network that determines exactly the productions that are instantiated by working memory.

Work specifically on speeding up the Rete algorithm has proceeded both in the areas of software and hardware. Forgy [2] proposed several ways of speeding up the Rete network, including using hash tables at the memory nodes to speed up removes and using "binary search" nodes or "index" nodes to eliminate redundant execution of mutually exclusive tests. A reimplementation in BLISS of the original LISP OPS5 resulted in a six-fold speedup [6]. The culmination of the attempts at speeding up the Rete algorithm in software was OPS83 [5], a system in which the network is compiled into machine code. OPS83 is four times faster than BLISS OPS5 [6].

Forgy [2] also proposed a machine architecture that would be capable of interpreting the Rete network at high speed. Most of the other work on hardware support for the Rete algorithm has been in the area of parallel processors. Gupta [7] provides a comprehensive summary of the expected gains from executing parts of the matching process in parallel. He concludes that the maximum speedup for any of the proposed uses of parallelism in the Rete algorithm is not likely to exceed twenty-fold. He proposes the use of hash tables at memory nodes to make processing time of and-node activations less variable, thus making parallel processing of and-node operations more advantageous. The DADO group at Columbia has also investigated various parallel production system matching algorithms for use on the highly-parallel DADO machine, including several parallel versions of the Rete algorithm [18].

Miranker [13] proposes a variant of the Rete algorithm, called TREAT, for use on DADO. The TREAT algorithm builds a network of alpha branches terminated by alpha memories, as in the Rete algorithm, but eliminates the and-nodes and beta memories. No information on instantiations of more than one condition is stored between cycles. Hence, the TREAT algorithm must, on each cycle, recompute the and-node tests (what Miranker calls a join reduction) between all of the alpha memories of any production that may be affected by the changes to working memory on the previous cycle. However, because there is no fixed order in which the alpha memories must be joined, the join reductions may be dynamically ordered according to the size of each of the alpha memories. Also, the join reductions for a particular production can be stopped whenever any of the join reductions yields zero instantiations, since then there are no possible instantiations of the whole production. One special case of this is that no computation need be done if any of the alpha memories has no tokens in it. Miranker developed the TREAT algorithm to run on a parallel machine (in which many of the join reductions could be done in parallel), but claims on the basis of preliminary tests that the TREAT algorithm runs at least as fast as the Rete algorithm on a uniprocessor.

18. Summary of Results

In this section we summarize the results we have obtained in attempting to speed up the Rete algorithm as it is used by SOAR. First, we discuss the applicability of each of the changes described to general OPS5 tasks. Then we briefly summarize the results obtained for SOAR tasks for each of the changes.

18.1. Results for OPS5

The applicability to OPS5 tasks of the changes and additions to the Rete algorithm described in the preceding sections can be summarized as follows:

- Condition ordering is not as vital for OPS5 tasks as it is for SOAR tasks. The main reason for this is that typical OPS5 conditions have more fields and specify more intra-condition tests than do SOAR conditions, and so tend to have fewer instantiations. Thus, the order in which conditions are joined does not matter as much as it does in SOAR. Similarly, it does not matter as much for OPS5 conditions whether the conditions are joined by a linear or non-linear structure. However, the cross-product effect can still occur in OPS5 productions if the conditions are ordered badly. Some of the general connectivity principles of the SOAR reorderer could probably be adapted for use in an OPS5 condition ordering algorithm.
- OPS5 rules can create new rules during a run by executing the OPS5 "build" action with a form representing the new production as an argument. Hence, the modification to the network compiler to allow new productions to be integrated into the existing network in a way that they will be matched properly is useful for general OPS5 tasks.
- Indexing can be used as described in Section 13 to speed up the class-name tests of the alpha branches. The network compiler could probably be modified to notice other sets of mutually exclusive tests on the alpha branches and use indexing to speed these up as well.
- As was explained in Section 14, there are a number of problems with using hash tables at the memory nodes for general OPS5 tasks that are lessened for SOAR tasks. Because the implementation described depended on several properties of SOAR tasks, it is not clear whether the use of hash tables would be advantageous for OPS5 tasks.
- The three ways described in Section 15 for speeding up removal of tokens from the network do not depend on any property of SOAR tasks. In particular, it is likely that a speedup would be obtained for OPS5 tasks by the use of the third method described in that section.
- Finally, the results on eliminating the interpretive overhead of the Rete algorithm are applicable to OPS5 tasks. That is, as with SOAR tasks, it is probably best to attempt to reduce the interpretive overhead through ways other than compiling the network.

18.2. Results for SOAR

The results for SOAR tasks of each of the changes investigated can be summarized very briefly:

- The ordering of conditions in SOAR productions has a great effect on the efficiency of the match. An effective algorithm for ordering conditions can be built, which uses little knowledge of the domain or the intended meaning of the productions.
- While the use of non-linear Rete networks can sometimes positively affect the efficiency of the match, it is not clear whether they are useful in general for matching SOAR productions.
- It is likely that any method of adding tests to the alpha branches that reduces the amount of testing done at the and-nodes will speed up the matching process.

- Modifying the network compiler so that chunks are integrated into the existing Rete network improves the efficiency of SOAR learning tasks.
- Indexing techniques are quite useful in reducing the number of tests executed in the alpha branches of Rete networks for SOAR tasks.
- Hash tables can be useful at right memory nodes for speeding up and-node operations.
- The removal of tokens from the network can be sped up by the modification to the Rete algorithm described in Section 15.
- It seems best to try to reduce the interpretive overhead of the Rete algorithm through ways other than compiling the network into executable code.

Table 18-1 summarizes the timing results for some of the changes described. As indicated in Section 7, all the timing results are for a particular run of the SOAR eight puzzle task. ADD time and REMOVE time refer to the matching time spent adding wme's to the network and removing wme's from the network, respectively. SOAR time refers to the total time for the run, including the matching time and the time used by the SOAR architecture. SOAR consistently uses about 25 seconds of the run, regardless of the matching time. The fifth column indicates the percent of the original matching time that the matching time of each succeeding version is.

The modifications to the Rete algorithm are referenced in the table by the following keywords. INDEX refers to the changes described in Section 13 that use indexing to reduce the number of alpha tests executed. SPEED-INTERP refers to changes described in Section 16.1 for reducing the overhead of interpreting the network. LEFT-HASH, RIGHT-HASH, and ALL-HASH refer to the modifications to the Rete algorithm in which hash tables are used at left memories, right memories, and all memories, respectively. SUBHASH refers to the change described in Section 14.3 in which tokens are grouped by hash key within the buckets of the hash table (hashing at right memories only). QUICK-REMOVE refers to the the third way described in Section 15 for speeding up removes. COMPILE-NET and HCOMPILE-NET refer to the compilation of the network to executable code, without and with hash tables at the right memories, respectively. NE-UNDEC refers to the change described in Section 11 of inserting "<> undecided" tests in some goal conditions.

Line (11) of the table indicates that the cumulative effect of the INDEX, SPEED-INTERP, RIGHT-HASH, SUBHASH, QUICK-REMOVE, and NE-UNDEC changes is to reduce the matching time for the eight puzzle to a quarter of its original value. Similar speedups for the combination of these changes and for the changes individually have been obtained for a number of SOAR tasks. Additionally, all learning tasks are sped up by the modification described in Section 12 that allows chunks to be integrated into the network during a run. The four-fold speedup shown in the table is especially significant in that the final result is that the matching time for the run is about fifty percent of the total SOAR time. Hence, the matching process no longer dominates the remaining processing of the SOAR architecture.

version	SOAR time	ADD time	REMOVE time	MATCH time	% of (1)
(1): SOAR 4.0	131.0	55.11	52.01	107.12	100
(2): (1) + INDEX	120.0	49.09	46.28	95.37	89
(3): (2) + SPEED-INTERP	97.8	37.51	36.10	73.61	69
(4): (3) + COMPILE-NET	88.0	32.14	30.96	63.10	59
(5): (3) + LEFT-HASH	105.0	42.78	38.77	81.55	76
(6): (3) + RIGHT-HASH	78.9	28.17	26.48	54.65	51
(7): (3) + ALL-HASH	88.5	33.07	29.68	62.75	59
(8): (6) + SUBHASH	69.5	23.42	22.42	45.84	43
(9): (8) + HCOMPILE-NET	62.8	19.83	19.08	38.91	36
(10): (8) + QUICK-REMOVE	56.8	23.09	10.04	33.13	31
(11): (10)+ NE-UNDEC	50.3	19.25	8.21	27.46	26

Table 18-1: Timing Results for Some of the Modifications

Acknowledgements

I would like to thank Harold Brown, John Laird, and particularly Paul Rosenbloom for comments on drafts of this paper. Thanks also to John Laird and Paul Rosenbloom for their guidance and advice throughout the period that I was doing the work described in this paper.

I was supported by a National Science Foundation Fellowship while doing the work reported in this paper. Computer facilities were partially provided by NIH grant RR-00785 to Sumex-Aim.

Appendix A. Network Statistics

The six tasks for which statistics are given in the tables below are:

- Eight puzzle (EIGHT)
- Eight puzzle with learning on (EIGHTL)
- Missionaries and cannibals (MC) - SOAR task that attempts to solve the Missionaries and Cannibals problem
- Neomycin-Soar (NEOM) - an implementation of part of Neomycin [1] in SOAR
- R1-Soar [15] (R1) - an implementation of part of R1 [11] in SOAR
- R1-Soar with learning on (R1L)

All of the tasks run not only with the task productions, but with the 51 default productions mentioned in Section 7.

	EIGHT	EIGHTL	MC	NEOM	R1	R1L
# of task productions	11	26	15	196	262	300
# of default productions	51	51	51	51	51	51
# of nodes in network without sharing	3335	5521	3501	12115	16322	23556
# of nodes in network with sharing	1048	1591	1088	3578	4628	7149
sharing factor	3.2	3.5	3.2	3.4	3.5	3.3
% of alpha nodes	10.6	7.0	11.1	8.8	8.4	6.1
% of alpha memories	9.0	6.0	9.4	7.9	7.6	5.5
avg. # of outputs of alpha memories	4.7	7.5	4.4	5.5	5.7	8.3
% of beta memories	34.2	38.6	33.4	34.1	34.7	38.9
avg. # of outputs of beta memories	1.1	1.1	1.1	1.2	1.1	1.1
% of and-nodes	39.5	43.1	39.0	40.5	40.5	42.9
avg. # of and-tests at and-node	1.2	1.6	1.3	1.1	1.2	1.2
avg. # of outputs of and-nodes	1.0	1.0	1.0	1.0	1.0	1.0
% of not-nodes	0.9	0.6	1.1	1.8	2.1	1.7
avg. # of not-tests at not-node	1.7	1.7	1.7	1.3	2.0	1.8
avg. # of outputs of not-nodes	1.0	1.0	1.0	1.0	1.0	1.0
% of production nodes	5.9	4.8	6.1	6.9	6.8	4.9

Table A-1: Statistics on Composition of Rete Network

In the statistics on removes in Table A-2 below, "avg number of tokens examined" refers to the average number of tokens that must be examined to find the token to be removed. "And-left calls" and "and-right calls" refer to the cases when a token enters the left input and right input of an and-node, respectively. "Null-memory" calls refer to cases when a token enters an and-node and the

opposite memory is empty. The third, fourth, and fifth statistics given for and-left and and-right calls are averaged over only non-null-memory calls.

	EIGHT	EIGHTL	MC	NEOM	R1	R1L
Total changes to working memory	3117	1572	8837	3667	4124	3667
Adds to working memory	1775	1015	4651	2271	2243	2266
Removes from working memory	1342	557	4186	1396	1881	1401
Alpha node calls:						
Number of calls	60811	30304	185609	147887	251741	247178
% successful calls	12.7	12.6	11.4	8.8	5.0	4.6
Adds for alpha memories:						
Number of adds	2486	1406	6189	5441	4955	5461
Avg number of tokens in memory	15.6	14.6	23.7	31.7	24.9	27.8
Removes for alpha memories:						
Number of removes	1941	835	5655	3449	4232	3534
Avg number of tokens in memory	15.2	9.5	25.3	28.6	27.8	25.6
Avg number of tokens examined	1.3	1.2	1.6	3.4	4.2	3.6
Adds for beta memories						
Number of adds	18981	52230	29672	7740	10056	10555
Avg number of tokens in memory	5.6	81.7	5.1	3.8	5.7	5.7
Removes for beta memories:						
Number of removes	18769	46280	29538	7102	9558	9418
Avg number of tokens in memory	6.6	79.0	6.1	4.7	6.8	6.9
Avg number of tokens examined	4.1	44.5	1.8	2.4	2.2	2.5
And-left calls						
% of null-memory calls	3.5	0.9	3.8	39.6	21.8	21.2
# of non-null-memory calls	38836	100046	65835	17970	22944	22717
avg length of opposite memory	8.8	21.9	12.7	6.8	4.9	5.1
avg # of consistent tokens in opposite memory	1.0	0.9	0.9	0.9	0.8	0.8
avg # of and-tests executed for non-consistent tokens	1.0	1.1	1.1	1.1	1.1	1.2
avg total # of tests executed	9.3	24.6	13.9	8.4	6.2	6.5
total # of tests executed	361613	2458238	911896	150836	142656	147628
And-right calls						
% of null-memory calls	73.3	65.6	46.4	84.6	79.4	81.4
# of non-null-memory calls	8189	8433	36753	13691	17386	15767
avg length of opposite memory	1.70	12.1	3.7	2.7	2.5	2.5
avg # of consistent tokens in opposite memory	0.3	0.6	0.2	0.3	0.2	0.2
avg # of and-tests executed for non-consistent tokens	1.0	1.0	1.0	1.0	1.0	1.0
avg total # of tests executed	1.4	11.8	3.6	2.5	2.3	2.3
total # of tests executed	11632	99086	133992	34267	39927	36083

Table A-2: Statistics on Node Operations

References

1. Clancey, W. J. & Letsinger, R. NEOMYCIN: Reconfiguring af Rule-based Expert System for Application to Teaching. In *Readings in Medical Artificial Intelligence: The First Decade*, Clancey, W. J. and Shortliffe, E. H., Eds., Addison-Wesley, Reading, 1984, pp. 361-381.

2. Forgy, C. L. *On the Efficient Implementation of Production Systems.* Ph.D. Th., Computer Science Department, Carnegie-Mellon University, 1979.

3. Forgy, C. L. *OPS5 User's Manual.* Computer Science Department, Carnegie-Mellon University, 1981.

4. Forgy, C. L. "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem". *Artificial Intelligence 19* (September 1982), pp. 17-37.

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

6. Forgy, C.L., Gupta, A., Newell, A., & Wedig, R. Initial Assessment of Architectures for Production Systems. National Conference for Artificial Intelligence, AAAI, Austin, 1984.

7. Gupta, A. *Parallelism in Production Systems.* Ph.D. Th., Computer Science Department, Carnegie-Mellon University, 1986.

8. Laird, J. E. *Soar User's Manual.* Xerox Palo Alto Research Center, 1986.

9. Laird, J. E., Newell, A., & Rosenbloom, P. S. Soar: An Architecture for General Intelligence. In preparation.

10. Laird, J. E., Rosenbloom, P. S., & Newell, A. "Chunking in Soar: The Anatomy of a General Learning Mechanism". *Machine Learning 1* (1986).

11. McDermott, J. "R1: A Rule-based Configurer of Computer Systems". *Artificial Intelligence 19* (September 1982).

12. McDermott, J., Newell, A., & Moore, J. The Efficiency of Certain Production System Implementations. In *Pattern-Directed Inference Systems*, D. A. Waterman & F. Hayes-Roth, Ed., Academic Press, 1978.

13. Miranker, D. P. Performance Estimates for the DADO Machine: A Comparison of Treat and Rete. Fifth Generation Computer Systems, ICOT, Tokyo, 1984.

14. Newell, A. Production Systems: Models of Control Structures. In *Visual Information Processing*, W. Chase, Ed., Academic Press, 1973.

15. Rosenbloom, P. S., Laird, J. E., McDermott, J., & Orciuch, E. "R1-Soar: An Experiment in Knowledge-intensive Programming in a Problem-solving Architecture". *IEEE Transactions on Pattern Analysis and Machine Intelligence 7*, 5 (1985), pp. 561-569.

16. Rychener, M. D. *Production Systems as a Programming Language for Artificial Intelligence Applications.* Ph.D. Th., Computer Science Department, Carnegie-Mellon University, 1976.

17. Smith, D. E., & Genesereth, M. R. "Ordering Conjunctive Queries". *Artificial Intelligence 26* (1985), pp. 171-215.

18. Stolfo, S. J. Five Parallel Algorithms for Production System Execution on the DADO Machine. National Conference for Artificial Intelligence, AAAI, Austin, 1984.