Annotations by Trev

Production Matching for Large Learning Systems

Robert B. Doorenbos

January 31, 1995
CMU-CS-95-113

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Thesis Committee:
Jill Fain Lehman, Computer Science Department
Tom Mitchell, Computer Science Department
Manuela Veloso, Computer Science Department
Paul Rosenbloom, University of Southern California/Information Sciences Institute

This research was sponsored in part by a National Science Foundation Graduate Fellowship, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. Views and conclusions contained in this document are those of the author and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of Wright Laboratory or the United States Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notation thereon.

Keywords: artificial intelligence, large-scale production systems, machine learning, production match, Soar

Abstract

One of the central results of AI research in the 1970's was that to achieve good performance, AI systems must have large amounts of knowledge. Machine learning techniques have been developed to automatically acquire knowledge, often in the form of if-then rules (productions). Unfortunately, this often leads to a *utility problem* — the "learning" ends up causing an overall slowdown in the system. This is because the more rules a system has, the longer it takes to match them against the current situation in order to determine which ones are applicable.

To address this problem, this thesis is aimed at enabling the scaling up of the number of rules in production systems. We examine a diverse set of testbed systems, each of which learns at least 100,000 rules. We show that with the best existing match algorithms, the match cost increases linearly in the number of rules in these systems. This is inadequate for large learning systems, because it leads to a utility problem. We then examine the causes of this linear increase, and develop techniques which eliminate the major causes. The end result is an improved match algorithm, Rete/UL, which is a general extension of the existing state-of-the-art Rete match algorithm. Rete/UL's performance scales well on a significantly broader class of systems than existing match algorithms. The use of Rete/UL rather than Rete significantly reduces or eliminates the utility problem in all the testbed systems.

To my parents

Contents

List of Tables

List of Figures

Acknowledgments

More than anybody else, I think it was John Laird who perked my interest in AI and especially in problem-solving and learning systems. Teaching my undergraduate AI class at Michigan, John kept me awake through two-hour lectures at 8:30am. (What higher praise could there be?) I considered joining several AI projects when I came to CMU, and eventually settled on Soar, not because of John's affiliation with it, but in part because I found the Soar group's discussions of cognitive psychology so interesting. (Ironically, I ended up writing a thesis having nothing to do with cognitive psychology.) I wish I could say I knew exactly what I was doing when I chose Allen Newell for my advisor, but in truth, I think I just got really lucky. Only much later did I realize the significance of one's choice of advisor, and I now believe that picking Allen for my advisor was one of the best decisions I've ever made. Allen was constantly supportive, patient, inspirational, and enthusiastic, and he got me through the hardest parts of being a graduate student — floundering around, learning how to do research, and looking for a thesis topic. I wish he were here now to give me one of his handshakes.

Since Allen's death, Paul Rosenbloom has been my primary technical advisor. He made sure I kept my sights high, helped me flesh out and debug ideas, and basically provided a good substitute for Allen. This thesis has been greatly improved by Paul's careful reading of earlier drafts. Jill Fain Lehman, my local advisor, has been a great source of support and advice and has provided a sounding board for many of my ideas. Thanks to her, my writing is (I hope) much easier for non-experts to understand. Although Paul has technically been my "outside" committee member, that function has really been served by Tom Mitchell and Manuela Veloso. Besides reminding me that not everyone uses Soar, they provided lots of good advice and suggestions which strengthened the thesis. I also want to thank a few other people who helped improve the thesis. Luc Tartar's idea of using the Cauchy-Schwarz inequality greatly simplified the proof of Lemma 5.3. Section 5.8.2 came out of a discussion I had with Sal Stolfo. Oren Etzioni, Sue Older, and David Steier provided helpful comments on drafts of the thesis.

The Soar group has been a constant source of support, encouragement, and friendship. I was fortunate to have a great officemate, Gregg Yost, to answer my countless questions as I was learning Soar. Thad Polk and Rick Lewis introduced me to afternoon croquet games, a fine tradition which I have proudly continued. I am especially grateful to Milind Tambe and Anurag Acharya, my friends and colleagues in match algorithm research, who helped me learn the ins and outs of Rete and were always there to provide interesting discussions and challenging questions. I am also grateful to Josef Krems, Joe Mertz, Craig Miller, Josef Nerb, Constantine Papageorgiou, and David Steier for providing testbed systems without which this thesis would have been impossible.

At Dow Chemical, Gary Strickler, Kip Mercure, and the rest of the Camile group gave me a chance to cut my teeth on some challenging programming problems. I gained a lot of valuable experience there, for which I am indebted to them.

I want to thank my family for all their love and support. With three people in the family all working on Ph.D.'s at the same time, we haven't always managed to get together and see each other as often as we'd like. Maybe someday we'll all have tenure and that will change.

I've had some terrific friends both in Pittsburgh and elsewhere. Allen Bellas has been my co-conspirator in writing top ten lists on which *Lloyd Bridges* is always item #7 (for perfectly obvious reasons[1]). Shirley Tessler does the New York Times crossword with ease (in ink!), and someday will probably write a computer program that does it for her. Karen Fogus and I only barely seem to stay in touch, but she has always been there for me when I need her — thanks, Karen. Around here, Jim, Alan, Stewart, Jade, Rich, John, Karen, Mark, Greg, John, John, Gin, Phoebe, Dave, Mei, Mark, and Nina have made life at CMU and in Pittsburgh interesting and fun. I'm especially glad to have friends like Susan Hinrichs, Sue Older, and Scott Reilly. I hope I meet people as wonderful as them wherever I go.

[1] Well, we could have *Lloyd Bridges* as #5 on every list, but that would be stupid!

Chapter 1

Introduction

One of the central results of AI research in the 1970's was that to achieve good performance, AI systems must have large amounts of knowledge. "Knowledge is power," the slogan goes. Humans clearly use vast amounts of knowledge, and if AI is to achieve its long-term goals, AI systems must also use vast amounts. Since hand-coding large amounts of knowledge into a system is slow, tedious, and error-prone, machine learning techniques have been developed to automatically acquire knowledge, often in the form of if-then rules (*productions*). Unfortunately, this has often led to a *utility problem* (Minton, 1988b) — the "learning" has caused an overall slowdown in the system.

For example, in many systems, learned rules are used to reduce the number of basic steps the system takes in order to solve problems — by pruning the system's search space, for instance. But in order to determine at each step which rules are applicable, the system must match them against its current situation. Using current techniques, the matcher slows down as more and more rules are acquired, so each step takes longer and longer. This effect can outweigh the reduction in the number of steps taken, so that the net result is a slowdown. This has been observed in several recent systems (Minton, 1988a; Etzioni, 1990; Tambe et al., 1990; Cohen, 1990).

Of course, the problem of slowdown from increasing match cost is not restricted to systems in which the purpose of rules is to reduce the number of problem-solving steps. A system acquiring new rules for any purpose can slow down, if the rules significantly increase the match cost. And intuitively, one expects that the more productions there are in a system, the higher the total match cost will be.

The thesis of this research is that we can solve this problem in a broad class of systems by improving the match algorithm they use. In essence, our aim is to enable the scaling up of the number of rules in production systems. We advance the state-of-the-art in production match algorithms, developing an improved match algorithm whose performance scales well on a significantly broader class of systems than existing algorithms. Furthermore, we demonstrate that by using this improved match algorithm, we can reduce or avoid the utility problem in a large class of machine learning systems.

1.1 Previous Approaches to the Utility Problem

Previous research on the problem of slowdown from increasing match cost has taken two general approaches. One approach is simply to reduce the number of rules in the system's knowledge base, via some form of selective learning or forgetting. (Markovitch and Scott, 1993) provides a general framework, *information filters*, for analyzing this approach. Examples include discarding learned rules if they turn out to slow down the matcher enough to cause an overall system slowdown (Minton, 1988a), disabling the learning component after some desired or peak performance level has been reached (Holder, 1992), learning only certain types of rules (e.g., nonrecursive) that are expected to have low match cost (Etzioni, 1993), and employing statistical approaches to ensure (with high probability) that only rules that actually improve performance get added to the knowledge base (Gratch and DeJong, 1992; Greiner and Jurisica, 1992).

Unfortunately, this approach alone is inadequate for the long-term goals of AI, because with the current state of match technology, it solves the problem by precluding the learning of vast amounts of knowledge. Whenever a system learns a moderate number of rules, the match cost increases substantially, eventually reaching a point where it slows down the overall system. Selective learning approaches by themselves thus prevent more than a moderate number of rules from ever being acquired. Such approaches can be complemented, however, by other approaches to reducing match cost. For example, in a system which saves a rule only when its benefits outweigh its costs, using other approaches that reduce match cost (without affecting rules' benefits) will change the "break-even" point, allowing a system to save and make use of more rules.

The second major approach taken by previous research has been to reduce the match cost of individual rules, taken one at a time. Many techniques have been developed for this. For instance, (Tambe et al., 1990) prevents the formation of individual *expensive rules* that have a combinatorial match cost by restricting the representation a system uses. Prodigy's *compression* module (Minton, 1988a) reduces match cost by simplifying the conditions of rules generated by Prodigy/EBL. Static (Etzioni, 1990) and Dynamic (Pérez and Etzioni, 1992) analyze the structure of a problem space to build much simpler rules with lower match cost for many of the same situations as Prodigy/EBL. (Chase et al., 1989) generalizes or specializes the conditions of search control rules so as to reduce their match cost. A similar approach is taken by (Cohen, 1990), where all but a few of the conditions of search control rules are dropped. (Kim and Rosenbloom, 1993) incorporates extra "search control conditions" into learned rules in order to reduce their match cost.

This line of work is helpful for reducing the cost of individual rules, enabling a system to learn more rules before an overall slowdown results. Unfortunately, an overall slowdown can still result when a large number of individually cheap rules together exact a high match cost. As the number of rules in the system increases, the average match cost grows; this is called the *average growth effect* (Tambe et al., 1990) or the *swamping effect* (Francis and Ram, 1993).

To address this problem, this thesis takes a new approach, complementary to the above two, examining the relatively unexplored area of reducing the aggregate match cost of a large number of rules without regard to the cost of each individual rule. Certain domains may afford ad-hoc methods here — for instance, in natural language processing, it may be possible to index learned

rules according to particular words or lexical categories they involve (Samuelsson and Rayner, 1991; Liu and Soo, 1992) — but no general method has been developed yet.

1.2 Overview of the Thesis

The increase in match cost due to an increase in the number of individually inexpensive productions is the focus of this thesis. The ultimate objective is to support large learned production systems, i.e., systems that learn a large number of rules.

An empirical study forms the core of the thesis. We examine a number of large learning systems and use them as testbeds for the algorithms we develop and study. These systems use a variety of problem-solving techniques in a variety of domains, and were written by a variety of people with different research interests. Each system learns at least 100,000 rules. The systems are implemented using Soar (Laird et al., 1987; Rosenbloom et al., 1991), an integrated problem-solving and learning architecture. Soar provides a useful vehicle for this research for three reasons: it provides a mechanism for learning new rules (*chunking*), it already incorporates one of the best existing match algorithms (Rete), and it has a large user community to provide the set of testbed systems.

Our examination of these very large production systems reveals new phenomena and calls into question some common assumptions based on previous observations of smaller systems. The best existing match algorithms — Rete (Forgy, 1982) and Treat (Miranker, 1990) — both slow down linearly in the number of rules in all our testbed systems. Such a linear slowdown is a serious problem for machine learning systems, as it will lead to a utility problem if enough rules are learned. Thus, the best available match algorithms do not scale very well with the number of rules in these systems.

Within the Rete match algorithm, the starting point for our development of an improved matcher, there are three sources of this linear slowdown. Two of these are pervasive, arising in many or all of our testbed systems, and are also expected to arise in a broad class of additional large learning systems.

We develop an improved match algorithm, called Rete/UL, by incorporating into Rete some changes which avoid these two sources of slowdown. With Rete/UL, a third source of linear slowdown remains in one testbed, but it is much less severe than the first two sources, and does not arise at all in most of our testbed systems.

We evaluate Rete/UL empirically, by measuring the performance of both Rete/UL and the basic Rete algorithm on each of our testbed systems and comparing the results. Rete's linear slowdown as the number of rules increases is eliminated by Rete/UL in all but one of the systems, and significantly reduced by Rete/UL in that one system. Rete/UL scales well on a significantly broader class of systems than do Rete and Treat. Moreover, with 100,000 rules in each testbed system, Rete/UL runs approximately two orders of magnitude faster than Rete.

Finally, we examine the match algorithm's impact on the utility problem. This is again done empirically, measuring the speedup attained or slowdown incurred by each of our testbed systems when they learn 100,000 or more rules. The use of Rete/UL rather than the basic Rete algorithm is shown to significantly reduce or eliminate the utility problem in all these systems.

1.3 Delimiting the Scope of the Thesis

To clarify the scope of this thesis, we mention a few related issues here which will *not* be addressed in this thesis. First, while the high cost of matching learned rules is a major cause of the utility problem in many systems, it is not the only possible cause. For example, (Mooney, 1989) shows that the particular way learned macro-rules are *used* also affects their utility. It is important to avoid any other possible sources of the utility problem in addition to avoiding a high match cost, but other sources are not the focus of this work.

Second, this thesis does not claim to *completely* solve the problem of high match cost. Indeed, this problem may never be completely solved, since matching a single rule is NP-hard in sufficiently expressive formalisms, and most commonly-used formalisms are sufficiently expressive. Unless P=NP, all match algorithms will have very bad *worst-case* behavior. Fortunately, the worst case does not always arise in practice.

Third, this work is aimed at match cost in *large* systems. Techniques that reduce match cost in systems with 10,000–100,000 rules may not necessarily be helpful in systems with 10–100 rules, since very small systems may have different characteristics than very large ones. Although anecdotal evidence so far suggests that our improved match algorithm can be quite helpful in small systems, different match optimizations may be needed to avoid having the match cost increase in systems that learn only a small number of rules. Since future AI systems will need to use large amounts of knowledge, it is important to focus on match cost in large systems.

Fourth, this thesis focuses on scaling up only the number of *rules* in production systems. Within the production systems community, some recent work has been aimed at improving the scalability of match algorithms (Acharya and Tambe, 1992; Miranker et al., 1990). However, this work focuses on scalability along a different dimension — it is aimed at scaling up the amount of data the rules manipulate, not the number of rules. Optimizations for both dimensions may eventually be combined to produce algorithms able to function efficiently with millions of rules and millions of data elements, but that is a topic for future research.

Fifth, this thesis will assume that all data is resident in main memory, and will not address issues of disk storage. Other researchers have investigated how production systems can be implemented using databases (Sellis et al., 1988) or how production rules can be used in database management systems (Hanson and Widom, 1993). In these systems, productions and working memory are stored on disk, rather than in main memory. Current disk access times are simply too slow to support adequate performance for the large learning systems examined in this thesis.

Sixth, this thesis focuses on match algorithms for sequential machines, not parallel ones. Of course, one way to address the increasing match cost in systems learning many rules is simply to use massive parallelism; e.g., if we can afford to devote one processor to each rule, we can avoid any increase in the time spent in matching, as long as we do not run out of processors. However, it is both interesting from a scientific standpoint and useful from a practical standpoint to investigate how effective uniprocessor algorithms can be in these systems. Studies of the performance of sequential algorithms can also inform the design of efficient parallel algorithms.

Finally, this thesis deals exclusively with *total* matching, not *partial*. (Veloso, 1992) has investigated efficient matching of large numbers of learned cases (previously-encountered problems

and their solutions). This requires finding a set of partially matched cases in a large knowledge base, while the current work requires finding all complete matches. It is unclear to what extent similar techniques can be useful in both situations, but this is an interesting possibility for future work.

1.4 Organization of the Thesis

The basic Rete match algorithm, the starting point for this research, is described in Chapter 2. The description is in tutorial style, using high-level pseudocode to illustrate the major data structures and procedures Rete uses. Readers who are already experts in production matching can probably skim or skip this chapter. Chapter 3 describes the systems we use as testbeds, and then presents empirical observations of the behavior of the basic Rete algorithm as we scale up the number of rules in each testbed. Among these observations is the fact that Rete slows down linearly in the number of rules in these systems. Chapters 4 and 5 explore two pervasive causes of this slowdown and develop techniques to improve the match algorithm so as to avoid them. Chapter 6 gives some theoretical analysis of match cost and also explains the remaining small source of linear slowdown in one of our testbed systems. Chapter 7 examines the impact of the choice of match algorithm on the utility problem, demonstrating that our improved match algorithm reduces or eliminates the utility problem in all our testbed systems. Finally, Chapter 8 summarizes the results and contributions of the thesis and suggests some interesting directions for future work.

We discuss related work at appropriate points throughout the thesis. We have already discussed some other approaches to the utility problem, and some other areas of work on production systems. Related work on production match algorithms is discussed primarily in Chapter 2 and Sections 3.4, 3.6, and 5.8. Related work on the utility problem is discussed primarily in Chapter 7.

Chapter 2

The Basic Rete Algorithm

Start of the explanation, this paper isn't about Rete, but uses it so the author has decided to provide a "tutorial" base overview

Since the Rete match algorithm provides the starting point for much of the work in this thesis, this chapter describes Rete. Unfortunately, most of the descriptions of Rete in the literature are not particularly lucid,[1] which is perhaps why Rete has acquired "a reputation for extreme difficulty."(Perlin, 1990b) To remedy this situation, this chapter describes Rete in a tutorial style, rather than just briefly reviewing it and referring the reader to the literature for a full description. We will first give an overview of Rete, and then discuss the principle data structures and procedures commonly used to implement it. High-level pseudocode will be given for many of the structures and procedures, so that this chapter may serve as a guide to readers who want to implement Rete (or some variant) in their own systems. Readers who are already familiar with Rete or who just want to read about the research contributions of this thesis should skim or skip this chapter. Sections 2.6 and higher discuss advanced aspects of Rete and can be skipped on first reading; most of the rest of the thesis is understandable without them.

Before beginning our discussion of Rete, we first review some basic terminology and notation. Rete (usually pronounced either "REET" or "REE-tee," from the Latin word for "network") deals with a *production memory* (PM) and a *working memory* (WM). Each of these may change gradually over time. The working memory is a set of items which (in most systems) represent facts about the system's current situation — the state of the external world and/or the internal problem-solving state of the system itself. Each item in WM is called a *working memory element*, or a *WME* (pronounced either "wuh-MEE" or "WIH-mee") for short. In a "blocks world" system, for example, the working memory might consist of the following WMEs (numbered `w1`–`w9`): The WME notation here corresponds to our `InternalRepresentation` type for facts

```
w1: (B1 ^on B2)          w6: (B2 ^color blue)
w2: (B1 ^on B3)          w7: (B3 ^left-of B4)
w3: (B1 ^color red)      w8: (B3 ^on table)
w4: (B2 ^on table)       w9: (B3 ^color red)
w5: (B2 ^left-of B3)
```

Okay, so these are "Facts" in their example system. They describe color blocks that are stacked on top of or beside each other.

So the first WME `w1` has the fact:
B1 (Block 1) is ^on (on top of) B2 (Block 2)

`w4` is the fact that:
B2 (Block 2) is ^on the table, where table is kinda an arbitrary symbol that the calling system would need to interpret.

[1] The definitive description, Forgy's thesis (Forgy, 1979), is one of the better ones, but the much wider-read *Artificial Intelligence* journal article (Forgy, 1982) tends to swamp the reader in the details of a particular very low-level machine implementation.

To simplify our description, we will assume that WMEs take the form of triples (three-tuples); we will write these as `(identifier ^attribute value)`. The names of the parts — "identifier," "attribute," and "value" — have no special significance to the matcher. We sometimes abbreviate "identifier" as "id" or "attribute" as "attr." We will refer to the parts as the three *fields* in a WME; e.g., the WME `(B1 ^on B2)` has `B1` in its identifier field. Each field contains exactly one symbol. The only restriction on what symbols are allowed is that they must all be constants: no variables are allowed in WMEs. Rete does not require this particular representation — numerous versions of Rete supporting others have been implemented, and we will discuss some of these in Section 2.11. We choose this particular form here because:

I think the author means constants in the mathematical sense, so constants here are not limited to primitive values, they can be complex data.

- It is very simple.
- The restriction to this form of WMEs entails no real loss of representational power, since other less restricted representations can be straightforwardly and mechanically converted into this form, as we will see in Section 2.11.
- The testbed systems described later in this thesis use this representation (as do all Soar systems).

The production memory is a set of *productions* (i.e., rules). A production is specified as a set of *conditions*, collectively called the *left-hand side* (LHS), and a set of *actions*, collectively called the *right-hand side* (RHS). Productions are usually written in this form:

```
(name-of-this-production
  LHS       /* one or more conditions */
 -->
  RHS       /* one or more actions */
)
```

For our rules, the LHS would be the `what` block, and the RHS would be then `then` block

Match algorithms usually ignore the actions and deal only with the conditions. The match algorithm serves as a module or subroutine used by the overall system to determine which productions have all their conditions satisfied. Some other part of the system then handles those productions' actions as appropriate. Consequently, this chapter will also focus on the conditions. Conditions may contain variables, which we write in angle brackets; e.g., `<x>`. In our "blocks world" example, the following production might be used to look for two (or more) blocks stacked to the left of a red block:

```
(find-stack-of-two-blocks-to-the-left-of-a-red-block
  (<x> ^on <y>)
  (<y> ^left-of <z>)
  (<z> ^color red)
 -->
  ... RHS ...
)
```

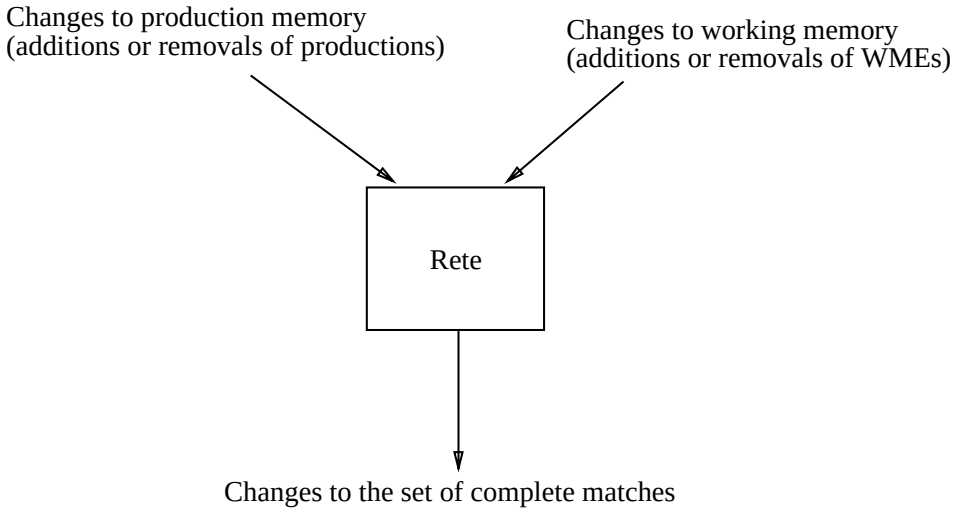


Figure 2.1: The Rete algorithm as a "black box" with inputs and outputs.

A production is said to *match* the current WM if all its conditions match (i.e., are satisfied by) items in the current working memory, with any variables in the conditions bound consistently. For example, the above production matches the above WM, because its three conditions are matched by `w1`, `w5`, and `w9`, respectively, with `<y>` bound to `B2` in both the first and second conditions and `<z>` bound to `B3` in both the second and third conditions. The match algorithm's job is to determine which productions in the system match the current WM, and for each one, to determine which WMEs match which conditions. Note that since we have assumed WMEs all take the form of three-tuples, we can also assume that conditions have a similar form — a condition not in this form would never be satisfied by any WME, so it would be useless.

As illustrated in Figure 2.1, Rete can be viewed as a "black box." As input, it takes information about changes to the current working memory (e.g., "Add this WME: ...") or to the set of productions (e.g., "Add this production: ...").[2] Each time it is informed of one of these changes, the match algorithm must output any changes to the set of matching productions (e.g., "Production ... now matches these WMEs: ...").

2.1 Overview

We begin with a brief overview of Rete. As illustrated in Figure 2.2, Rete uses a dataflow network to represent the conditions of the productions. The network has two parts. The *alpha* part performs the constant tests on working memory elements (tests for constant symbols such as `red` and `left-of`). Its output is stored in *alpha memories* (AM), each of which holds the current set of working memory elements passing all the constant tests of an individual condition. For example, in the figure, the alpha memory for the first condition, `(<x> ^on <y>)`, holds the

[2] Not all implementations of Rete support dynamic addition and deletion of productions — some require all productions to be specified at the start.

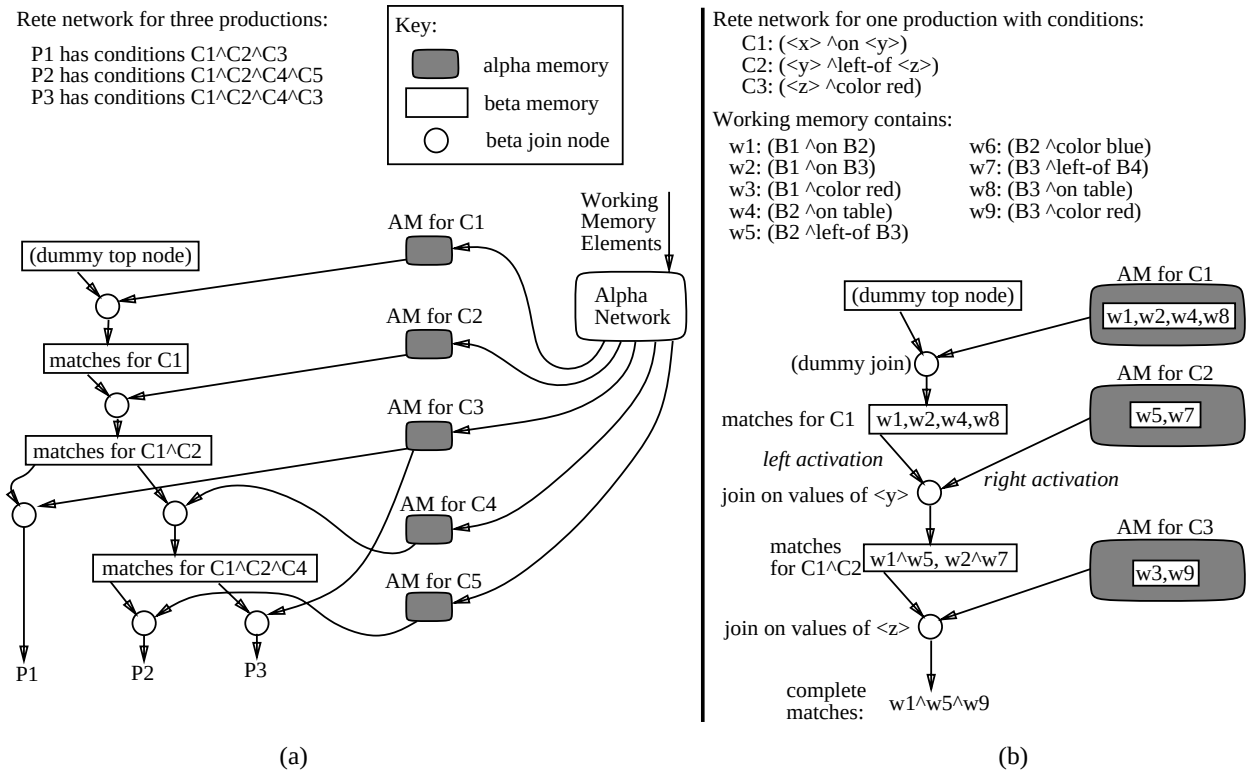


Figure 2.2: Example network used by Rete, (a) for several productions, and (b) instantiated network for one production.

WMEs whose attribute field contains the symbol `on`. The implementation of the alpha network is discussed in Section 2.2. The *beta* part of the network primarily contains *join nodes* and *beta memories*. (There are a few other types of nodes we will discuss later.) Join nodes perform the tests for consistency of variable bindings between conditions. Beta memories store partial instantiations of productions (i.e., combinations of WMEs which match some but not all of the conditions of a production). These partial instantiations are called *tokens*.

Strictly speaking, in most versions of Rete, the alpha network performs not only constant tests but also *intra-condition* variable binding consistency tests, where one variable occurs more than once in a single condition; e.g., `(<x> ^self <x>)`. Such tests are rare in Soar, so we will not discuss them extensively here. Also, note that a "test" can actually be any boolean-valued function. In Soar, almost all tests are equality tests (checking that one thing is equal to another), so we concentrate on them here. However, Rete implementations usually support at least simple relational tests (greater-than, less-than, etc.), and some versions allow arbitrary user-defined tests (Allen, 1982; Forgy, 1984; Cruise et al., 1987; Pasik et al., 1989; Miranker et al., 1991). In any case, the basic idea is that the alpha network performs all the tests which involve a single WME, while the beta network performs tests involving two or more WMEs.

An analogy to relational databases may be helpful here. One can think of the current working memory as a relation, and each production as a query. The constant tests in a condition represent a SELECT operation over the WM relation. For each different condition c_i in the system, there

is an alpha memory which stores the result $r(c_i)$ of that SELECT. Now, let P be a production with conditions $c_1, \ldots, c_k$. The matches for P with the current working memory (if P has any matches) are given by $r(c_1) \bowtie \cdots \bowtie r(c_k)$, where the JOIN operations perform the appropriate variable binding consistency checks. The join nodes in the beta part of the Rete network perform these joins, and each beta memory stores the results of one of the intermediate joins $r(c_1) \bowtie \cdots \bowtie r(c_i)$, for some $i < k$.

Whenever a change is made to working memory, we update these SELECT and JOIN results. This is done as follows: working memory changes are sent through the alpha network and the appropriate alpha memories are updated. These updates are then propagated over to the attached join nodes, *activating* those nodes. If any new partial instantiations are created, they are added to the appropriate beta memories and then propagated down the beta part of the network, activating other nodes. Whenever the propagation reaches the bottom of the network, it indicates that a production's conditions are completely matched. This is commonly implemented by having a special node for each production (called its *production node*, or *p-node* for short) at the bottom of the network. In Figure 2.2 (a), `P1`, `P2`, and `P3` at the bottom of the network would be p-nodes. Whenever a p-node gets activated, it signals the newly found complete match (in some system-dependent way).

The bulk of the code for the Rete algorithm consists of procedures for handling the various node activations. The activation of an alpha memory node is handled by adding a given WME to the memory, then passing the WME on to the memory's successors (the join nodes attached to it). Similarly, the activation of a beta memory node is handled by adding a given token to the memory and passing it on to the node's children (join nodes). In general, an activation of some node from another node in the beta network is called a *left* activation, while an activation of some node from an alpha memory is called a *right* activation. Thus, a join node can incur two types of activations: a *right activation* when a WME is added to its alpha memory (i.e., the alpha memory that feeds into it), and a *left activation* when a token is added to its beta memory (the beta memory that feeds into it). Right and left join node activations are normally implemented in two different procedures, but in both cases, the node's *other* memory is searched for (already existing) items having variable bindings consistent with the new item; if any are found, they are passed on to the join node's children.

To activate a given node, then, we use a procedure call. The particular procedure depends on the type of the node; for example, left activations of beta memory nodes are handled by one procedure, while a different procedure handles left activations of join nodes. To propagate the dataflow from a certain node to its successors, we iterate over the successors and call the appropriate activation procedure for each one. We determine which procedure is appropriate by looking at the type of the node: we either use a `switch` or `case` statement which branches according to the node type, each branch calling a different procedure, or we make the procedure call through a jumptable indexed by the node type.[3] (Some compilers convert a `switch`

[3] A jumptable is an array containing the addresses of several different procedures or blocks of code. It might appear that the structure of the network makes the type of a successor node completely predictable — alpha and beta memories always followed by join nodes, join nodes always by beta memories — but this is not the case. A join node may also be followed by a production node. When we discuss negative conditions in Sections 2.7 and 2.8, we will see other types of nodes that can follow join nodes and alpha memories. Beta memories, however,

statement into a jumptable for efficiency.)

There are two important features of Rete that make it potentially much faster than a naïve match algorithm. The first is its use of *state-saving*. After each change to WM, the state (results) of the matching process is saved in the alpha and beta memories. After the next change to WM, many or most of these results are usually unchanged, so Rete avoids a lot of recomputation by keeping these results around in between successive WM changes. (Rete is designed for systems where only a small fraction of WMEs change in between successive times rules need to be matched. Rete's state-saving would not be very beneficial in systems where most WMEs change each time.)

The second important feature of Rete is its *sharing* of nodes between productions with similar conditions. Different kinds of sharing occur in different parts of the network. There can be sharing within the alpha network, as discussed below in Section 2.2. At the output of the alpha network, when two or more productions have a common condition, Rete uses a single alpha memory for the condition, rather than creating a duplicate memory for each production. In Figure 2.2 (a), for example, the alpha memory for `C3` is shared by productions `P1` and `P3`. Moreover, in the beta part of the network, when two or more productions have the same first few conditions, the same nodes are used to match those conditions; this avoids duplication of match effort across those productions. In the figure, all three productions have the same first two conditions, and two productions have the same first three conditions. Because of this sharing, the beta part of the network forms a tree.[4]

Implementations of Rete can be either interpreted or compiled. In interpreted versions, the network just described is simply stored as a data structure which the interpreter traverses. In compiled versions (e.g., (Gupta et al., 1988); also see (Miranker and Lofaso, 1991) for a good description of a compiled version of a different match algorithm), the network is not represented explicitly, but is replaced by a set of procedures, usually one or two procedures per node. For example, where an interpreted version would apply a generic "left-activation-of-join-node" procedure to a particular join node (by passing the procedure a pointer to that node's data structure), a compiled version would instead use a special procedure created for just that particular node. Of course, the compiler creates the special procedure by partially evaluating an interpreter's generic procedure with respect to the particular node. (This chapter will describe the generic interpreter procedures.) The advantage of a compiled version is, of course, its faster speed. Its disadvantages are (1) larger memory requirements (the compiled procedure for a node usually takes more space than the interpreted data structure) and (2) difficulty of run-time addition or deletion of productions (modifying compiled code is harder than modifying an interpreted data structure — although at least one compiled version of Rete has tackled this

can still only be followed by join nodes. Of course, another way to implement this would be to keep several lists of successors on each node, one list for each possible type of successor, but this would yield only a slight speedup at the cost of significantly increasing space usage. (Each Rete node is a fairly small data structure, and the large systems we use in this thesis have millions of nodes, so adding a few extra pointers to each node requires a lot of extra space.)

[4] In general, it would form a forest, but we obtain a tree by adding a dummy node to the top of a forest. Some Rete implementations do not use a dummy top node and instead have the uppermost join nodes take inputs from two alpha memories. The use of a dummy top node, however, simplifies the description and can also simplify the implementation.

problem (Tambe et al., 1988)).

In the rest of this chapter, we will describe Rete in more detail, giving high-level pseudocode for its basic data structures and procedures. The pseudocode will be given section by section, and sometimes a later section will revise pseudocode given earlier, in order to add support for a feature being discussed in the later section. A complete version of the pseudocode, including some important improvements introduced in later chapters, appears in Appendix A.

The Rete module of a system has four entry points: *add-wme*, *remove-wme*, *add-production*, and *remove-production*.[5] We begin by discussing what happens on a call to *add-wme*: Section 2.2 describes what the alpha network does, Section 2.3 describes what alpha and beta memory nodes do, and Section 2.4 describes what join nodes do. We discuss *remove-wme* in Section 2.5, and *add-production* and *remove-production* in Section 2.6. Next, we discuss some more complicated features of Rete: Section 2.7 shows how to handle negated conditions (conditions which test for the *absence* of a WME), and Section 2.8 shows how negated conjunctions (testing for the absence of a combination of WMEs) can be handled. We then give a few implementation notes in Section 2.9 and survey some other optimizations that have been developed for Rete over the years in Section 2.10. Finally, Section 2.11 discusses the generality of the Rete algorithm, including its applicability to less restricted representations of WMEs.

2.2 Alpha Net Implementation

When a WME is added to working memory, the alpha network performs the necessary constant (or intra-condition) tests on it and deposits it into (zero or more) appropriate alpha memories. There are several ways of finding the appropriate alpha memories.

2.2.1 Dataflow Network

The original and perhaps most straightforward way is to use a simple dataflow network. Figure 2.3 illustrates such a network for a small production system whose rules use just ten conditions (`C1`–`C10`). The network is constructed as follows. For each condition, let $T_1, \ldots, T_k$ be its constant tests, listed in any order (the left-to-right ordering from the source text of the condition is commonly used). Starting at the top node, we build a path of k nodes corresponding to $T_1, \ldots, T_k$, in that order. These nodes are usually called *constant-test nodes* or *one-input nodes* in the literature. As we build this path, we share (i.e., reuse) existing nodes (for other conditions) containing identical tests whenever possible. Finally, we make the alpha memory for this condition be an output of the node for T_k.

Note that this construction pays attention only to the constants in the condition, while ignoring the variable names. Thus, in Figure 2.3, conditions `C2` and `C10` share an alpha memory even though they contain different variable names. Also note that it is possible for a condition to contain no constant test at all (`C9` in the figure, for example), in which case its alpha memory is simply a child of the top node in the alpha network.

[5] Some implementations also have a *modify-wme* entry point; we discuss this in Section 2.10.

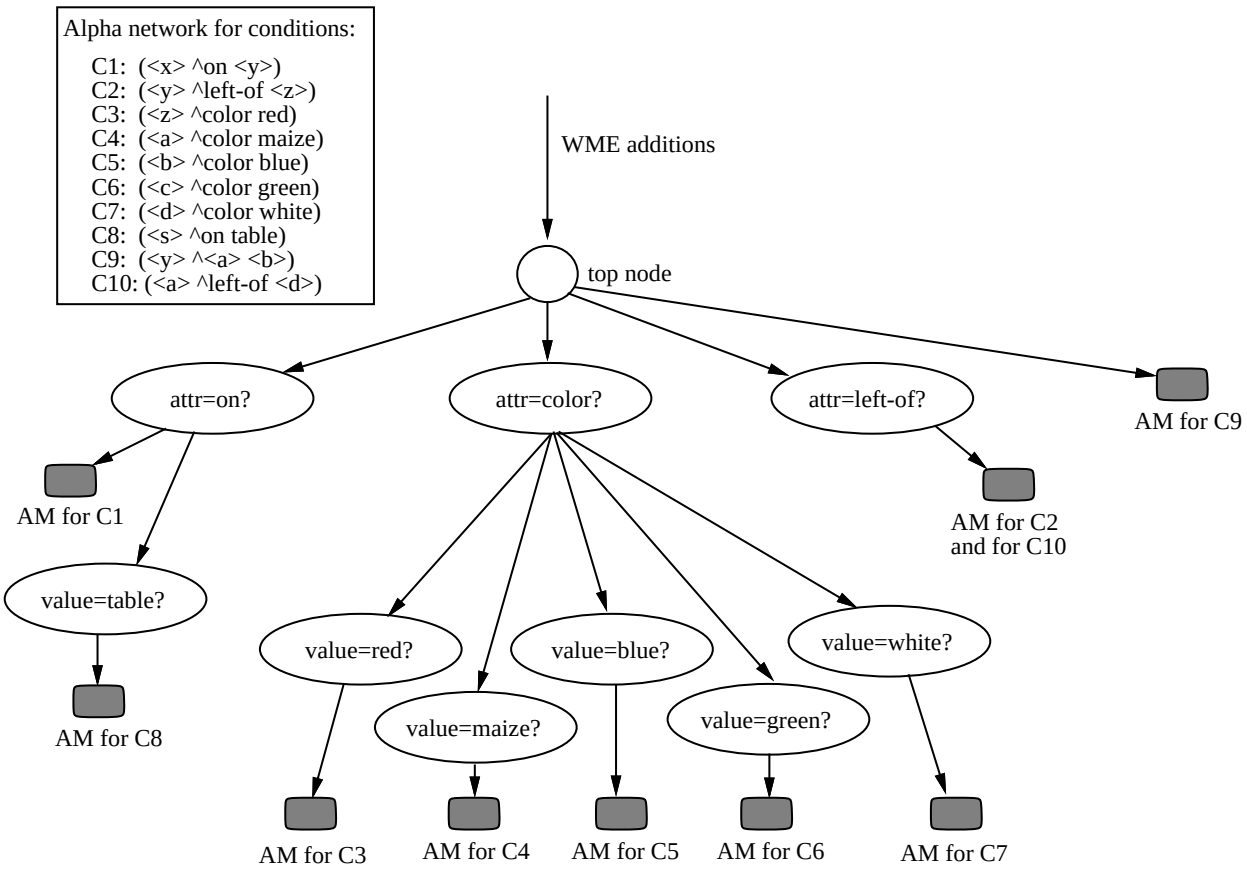


Figure 2.3: Example dataflow network used for the alpha network.

Each of these nodes is just a data structure specifying the test to be performed at the node, the alpha memory (if any) the node's output goes into, and a list of the node's children (other constant-test nodes):

structure constant-test-node:
    field-to-test: "identifier", "attribute", "value", or "no-test"
    thing-the-field-must-equal: **symbol**
    output-memory: alpha-memory or **nil**
    children: **list** of constant-test-node
end

(The "no-test" will be used for the top node.) When a WME is added to working memory, we simply feed it into the top of this dataflow network:

procedure add-wme (w: WME) {*dataflow version*}
    constant-test-node-activation (the-top-node-of-the-alpha-network, w)
end

```
procedure constant-test-node-activation (node: constant-test-node; w: WME)
    if node.field-to-test ≠ 'no-test' then
        v ← w.[node.field-to-test]
        if v ≠ node.thing-the-field-must-equal then
            return {failed the test, so don't propagate any further}
    if node.output-memory ≠ nil then
        alpha-memory-activation (node.output-memory, w) {see Section 2.3.1}
    for each c in node.children do constant-test-node-activation (c, w)
end
```

The above description assumes that all tests are tests for equality with some constant symbol. As mentioned earlier, the tests performed by the alpha network can include tests other than equality. For example, a condition might require a certain field in the WME to have a numeric value greater than seven. To support such tests, we would expand the *constant-test-node* data structure to include a specification of what kind of test is to be performed, and modify the *constant-test-node-activation* procedure accordingly.

2.2.2 Dataflow Network with Hashing

The above implementation of the alpha network is simple and straightforward. It has one serious drawback in large systems, however. As the reader may have already guessed from Figure 2.3, it can lead to a lot of wasted work when the fan-out from a node is large. In the figure, the node for `attr=color?` has five children, and their tests are all mutually exclusive. Whenever a WME passes the `attr=color?` test, five more tests will be performed, one at each child, and at most one could possibly succeed. Of course, there is no limit to how many of these mutually exclusive children a node can have, and the number could grow as a system learns new rules over time. An expert system for interior decoration might learn the names of more and more specific colors; a system learning medicine might learn about more and more diseases. As the system grows, the amount of "wasted" work performed at points like this in the alpha network would also grow, so the matcher would become increasingly slow.

The obvious solution to this problem is to replace this large fan-out point in the network with a special node which uses a hash table (or balanced binary tree) to determine which *one* path the activation needs to continue down (Forgy, 1979). In the above example, instead of activating five children, we would look at the contents of the "value" field of the WME, and hash directly to the appropriate child. In fact, the child node can be eliminated, since the hashing effectively performs the same test. Figure 2.4 shows the result of using this hashing technique. Extending the previous pseudocode to handle these hash tables is straightforward.

2.2.3 Exhaustive Hash Table Lookup

As long as WMEs are required to be three-tuples, there is a simple and elegant way to implement most of the alpha network using just a few hash table lookups (Acharya, 1992). Assume for the moment that all the constant tests are equality tests — i.e., that there are no "greater-than,"

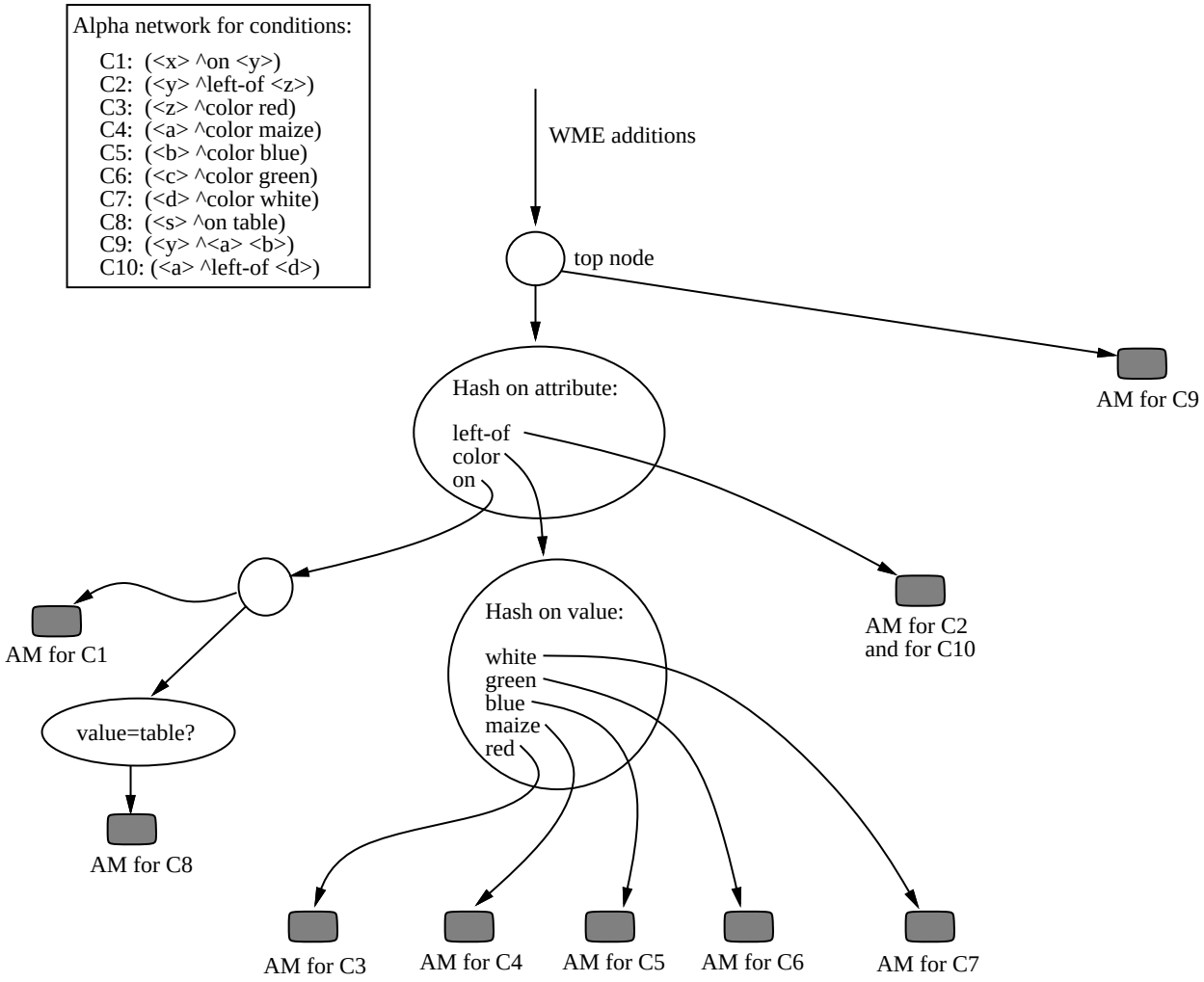


Figure 2.4: Example alpha network with hashing.

"not-equal-to," or other special tests. We can then make the following observation: *For any given WME, there are at most eight alpha memories that WME can go into.* This is because every alpha memory has the form `(test-1 ^test-2 test-3)`, where each of the three tests is either a test for equality with a specific constant symbol, or a "don't care," which we will denote by "$*$". If a WME $w =$`(v1 ^v2 v3)` goes into an alpha memory a, then a must have one of the following eight forms:

```
(*  ^*   *)          (v1  ^*   *)
(*  ^*   v3)         (v1  ^*   v3)
(*  ^v2  *)          (v1  ^v2  *)
(*  ^v2  v3)         (v1  ^v2  v3)
```

These are the only eight ways to write a condition which `(v1 ^v2 v3)` will match. Thus, given a WME w, to determine which alpha memories w should be added to, we need only check whether

any of these eight possibilities is actually present in the system. (Some might not be present, since there might not be any alpha memory corresponding to that particular combination of tests and $*$'s.) We store pointers to all the system's alpha memories in a hash table, indexed according to the particular values being tested. Executing the alpha network then becomes a simple matter of doing eight hash table lookups:

```
procedure add-wme (w: WME) {exhaustive hash table version}
    let v1, v2, and v3 be the symbols in the three fields of w
    alpha-mem ← lookup-in-hash-table (v1,v2,v3)
    if alpha-mem ≠ "not-found" then alpha-memory-activation (alpha-mem, w)
    alpha-mem ← lookup-in-hash-table (v1,v2,*)
    if alpha-mem ≠ "not-found" then alpha-memory-activation (alpha-mem, w)
    alpha-mem ← lookup-in-hash-table (v1,*,v3)
    if alpha-mem ≠ "not-found" then alpha-memory-activation (alpha-mem, w)
    ⋮
    alpha-mem ← lookup-in-hash-table (*,*,*)
    if alpha-mem ≠ "not-found" then alpha-memory-activation (alpha-mem, w)
end
```

The above algorithm, while elegant, relies on two assumptions: the three-tuple form of WMEs and the lack of non-equality tests. The first assumption can be relaxed somewhat: to handle WMEs that are r-tuples, we can use 2^r hash table lookups. Of course, this only works well if r is quite small. There are two ways to eliminate the second assumption:

- The result of the hash table lookups, instead of being an alpha memory, can be a small dataflow network of the form described above in Section 2.2.1. All tests for equality with a constant are thus handled by hashing, while other tests are handled by constant-test nodes as before. Essentially, this amounts to taking the dataflow network of Section 2.2.1, moving all equality tests up to the top half and all other tests down to the bottom half, and then replacing the top half by eight hash table lookups.

- The non-equality tests can be handled by the beta part of the network instead of the alpha part. Of course, it is possible for an implementation of Rete to perform some of the constant (or intra-condition) tests in the beta rather than the alpha network. This leaves the alpha network almost trivial, and turns out to not increase the complexity of the beta network much, if at all. However, it reduces the potential for these tests to be shared when different productions have a common condition. (This is the approach taken in the implementation used in this thesis.)

With either the dataflow-plus-hashing implementation of Section 2.2.2 or the exhaustive-hash-table-lookup implementation of Section 2.2.3, the alpha network is very efficient, running in approximately constant time per change to working memory.[6] The beta part of the network

[6] If non-equality tests are not used, we have an $O(1)$ bound on time per WM change, from the exhaustive-hash-table-lookup algorithm (assuming the hash function produces a reasonable distribution of items to hash

accounts for most of the match cost in Soar systems, and previous studies have shown this to be true of OPS5 (Forgy, 1981) systems as well (Gupta, 1987). Most of this chapter (and most of this thesis) will therefore deal with the beta part of the network.

2.3 Memory Node Implementation

We now discuss the implementation of alpha and beta memory nodes. Recall that alpha memories store sets of WMEs, and beta memories store sets of tokens, each token representing a sequence of WMEs — specifically, a sequence of k WMEs (for some k) satisfying the first k conditions (with consistent variable bindings) of some production.

There are various ways of implementing memory nodes. We can classify implementations according to two criteria:

- How are the sets (of WMEs or tokens) structured?
- How is a token — a *sequence* of WMEs — represented?

For the first question, the simplest implementation imposes no special structure on the sets — they are represented simply as lists, without the items being in any particular order. However, we can often gain efficiency in the join operations by imposing an indexing structure on the memories. To see how, consider our earlier example:

```
(find-stack-of-two-blocks-to-the-left-of-a-red-block
  (<x> ^on <y>)           /* C1 */
  (<y> ^left-of <z>)      /* C2 */
  (<z> ^color red)        /* C3 */
 -->
  ... RHS ...
)
```

The Rete net for this single production is shown in Figure 2.2 (b) on page 10. When a new WME `(B7 ^color red)` is added to working memory, it will be added to the alpha memory for `C3`, and then the lowest join node will be right-activated. The join node must check whether there are any matches for the first two conditions which have the right value of `<z>`. So it searches its beta memory (i.e., the beta memory that feeds into it) for any tokens which have `<z>` bound to `B7`. Without any indexing, this search requires iterating over all the tokens in the beta memory; however, it can be done much faster if the tokens in the beta memory are indexed according to their bindings for `<z>`. Similarly, whenever a new token is added to the beta memory, the join node will be left-activated and will have to search its alpha memory for WMEs having the

buckets). If non-equality tests are used pervasively, however, no nice bound can be given, because a single WME can go into a huge number of alpha memories. For example, if we have conditions `(a ^b <7)`, `(a ^b <8)`, `(a ^b <9)`, ..., `(a ^b <1,000,000)`, then a new WME `(a ^b 6)` must be added to 999,994 different alpha memories. Fortunately, such pervasive use of non-equality tests does not appear to arise in practice.

appropriate value of <z>; this search can be sped up by having the alpha memory indexed as well.

The most common indexing method for memory nodes is a hash table. (Trees are another possibility, but they do not seem to have been used in many Rete implementations, and (Barachini, 1991) found that hash tables generally outperformed (binary unbalanced) trees.) However, putting a hash table in every memory node leads to a dilemma: How should the hash table size be chosen? The number of tokens or WMEs at a node can vary greatly over time. If the hash table is always small, we will have a lot of hash collisions, reducing the effectiveness of hashing. If the hash table is always large, we will waste a lot of memory at times when there are few or no tokens or WMEs in the node. We could dynamically resize the hash table as items are added and removed, but this entails putting a potentially expensive piece of code inside what would otherwise be a very simple procedure.

The usual solution to this dilemma is to index all tokens from all beta memories in one big global hash table, and all WMEs from all alpha memories in another big global hash table, instead of using separate hash tables at each node (Gupta, 1987). The sizes of these global hash tables are chosen ahead of time to be very large (usually several thousand buckets). The hash function is a function of both the appropriate variable binding from the token or WME (e.g., the value of <z> above) and the node itself (e.g., an identification number unique to the particular memory node containing the token or WME, or the virtual memory address of that node). The idea is that to minimize collisions, (1) if we fix the memory node and vary the variable bindings, the hash function results should be spread out evenly among the buckets; and (2) if we fix the variable bindings but vary the memory node, the results should again be spread out evenly among the buckets.

As mentioned above, using indexed memories can greatly speed up the matcher by reducing the time spent computing joins. It does have two disadvantages, though. First, it increases the time needed to add or remove an item to or from a memory node, since the index must be updated. Second, it can reduce sharing: sometimes "duplicate" memory nodes have to be constructed, each one storing the same information but indexing it in a different way. For example, consider what would happen if we added a second, slightly different, production to the aforementioned one:

```
(slightly-modified-version-of-previous-production
  (<x> ^on <y>)           /* C1 */
  (<y> ^left-of <z>)      /* C2 */
  (<y> ^color red)        /* C3, but tests <y> instead of <z> */
 -->
  ... RHS ...
)
```

For this production, we want the tokens representing matches for the first two conditions to be indexed according to their bindings for <y>, rather than their bindings for <z>. If memory nodes are not indexed at all, but instead are simple unordered lists, then both productions can share a single memory node to store matches for the first two conditions. With indexing, we need two memory nodes instead of one (or one node that maintains two different indices). Because of

these two small costs, indexing may not be worthwhile in some systems, particularly ones where memories never contain very many items.

In spite of these potential drawbacks, empirical results indicate that in practice, hashed memories are an improvement over unindexed memories. (Gupta et al., 1988) found that their use sped up the match algorithm by a factor of 1.2–3.5 in several OPS5 systems, and (Scales, 1986) reports a factor of 1.2–1.3 for Soar systems.

Note that we cannot always find a variable whose binding we should index a memory on. For example, sometimes one condition is completely independent of all the earlier conditions:

```
(<x> ^on <y>)           /* C1 */
(<a> ^left-of <b>)      /* C2 */
...
```

In this example, there is no point in indexing the beta memory preceding the join node for `C2`, since `C2` doesn't test any variables that are used in `C1`. In cases like this, we simply use an unindexed memory node.

Turning now to the second question — How is a token (sequence of WMEs) represented? — there are two main possibilities. A sequence can be represented either by an array (yielding *array-form tokens*) or by a list (yielding *list-form tokens*). Using an array would seem the obvious choice, since it offers the advantage of direct access to all the elements in the sequence — given i, we can find the i^{th} element in constant time — whereas a list requires a loop over the first $i-1$ elements to get to the i^{th} one.

However, array-form tokens can result in a lot of redundant information storage and hence much more space usage. To see why, note that for every beta memory node storing matches for the first $i > 1$ conditions in a production, there is another beta memory node — namely, its grandparent (skipping over the intervening join node) — storing matches for the first $i-1$ conditions. And if $\langle w_1, \ldots, w_i \rangle$ is a match for the first i conditions, then $\langle w_1, \ldots, w_{i-1} \rangle$ must be a match for the first $i-1$ conditions. This means any token in the lower beta memory can be represented succinctly as a pair $\langle \mathit{parent}, w_i \rangle$, where *parent* is (a pointer to) the token in the upper beta memory representing the first $i-1$ WMEs. If we use this technique at all beta memory nodes, then each token effectively becomes a linked list, connected by *parent* pointers, representing a sequence of WMEs in reverse order, with w_i at the head of the list and w_1 at the tail. For uniformity, we make tokens in the uppermost beta memories (which represent sequences of just one WME) have their *parent* point to a *dummy top token*, which represents the null sequence $\langle\rangle$. Note that the set of all tokens now forms a tree, with links pointing from children to their parents, and with the dummy top token at the root.

With array-form tokens, a token for the first i conditions takes $O(i)$ space, whereas with list-form tokens, every token takes just $O(1)$ space. This can result in substantial space savings, especially if productions typically have a large number of conditions. If a production with C conditions has one complete match, array-form tokens will use at least $1 + 2 + \ldots + C = O(C^2)$ space, whereas list-form tokens will only require $O(C)$ space. Of course, using more space implies using more time to fill up that space. Every time a beta memory node is activated, it creates and stores a new token. With array-form tokens, this requires a loop which copies the $i-1$

elements from above, before adding in the new i^{th} element. With list-form tokens, creating a token does not require this loop.

To summarize, using array-form tokens requires more space than using list-form tokens, and requires more time to create each token on each beta memory node activation. However, it affords faster access to a given element of the sequence than using list-form tokens does. Access to arbitrary elements is often required during join node activations, in order to perform variable binding consistency checks. So we have a tradeoff. Neither representation is clearly better for all systems. In some systems, using array-form tokens is infeasible for space reasons — e.g., if productions have a huge number of conditions. In general, the choice depends on how high the cost of access to arbitrary elements is with list-form tokens. The more variable binding consistency checks a system uses, and the farther apart the variables involved are (i.e., the two variables occur in the very first and last conditions in a production, versus occurring in conditions c_i and c_{i+1}), the greater the access cost will be, and the more likely it is that array-form tokens will be faster.

We now turn to our pseudocode. To keep things as simple as possible, we use list-form tokens and unindexed memory nodes in this pseudocode; as we go along, we will note places where this makes a significant difference. (The actual implementation used in this thesis uses list-form tokens and hashed memory nodes.)

2.3.1 Alpha Memory Implementation

A WME simply contains its three fields:

```
structure WME:
    fields: array [1..3] of symbol
end
```

An alpha memory stores a list of the WMEs it contains, plus a list of its successors (join nodes attached to it):

```
structure alpha-memory:
    items: list of WME
    successors: list of rete-node
end
```

Whenever a new WME is filtered through the alpha network and reaches an alpha memory, we simply add it to the list of other WMEs in that memory, and inform each of the attached join nodes:

```
procedure alpha-memory-activation (node: alpha-memory, w: WME)
    insert w at the head of node.items
    for each child in node.successors do right-activation (child, w)
end
```

2.3.2 Beta Memory Implementation

With list-form tokens, as noted above, a token is just a pair:

structure token:
    parent: token {*points to the higher token, for items 1...i-1*}
    wme: WME {*gives item i*}
end

A beta memory node stores a list of the tokens it contains, plus a list of its children (other nodes in the beta part of the network). Before we give its data structure, though, recall that we were going to do our procedure calls for left and right activations through a `switch` or `case` statement or a jumptable indexed according to the type of node being activated. Thus, given a (pointer to a) node, we need to be able to determine its type. This is straightforward if we use *variant records* to represent nodes. (A variant record is a record that can contain any one of several different sets of fields.) Each node in the beta part of the net will be represented by a *rete-node* structure:

structure rete-node:
    type: "beta-memory", "join-node", or "p-node" {*or other node types we'll see later*}
    children: **list** of rete-node
    parent: rete-node {*we'll need this "back-link" later*}
    ...(variant part — other data depending on node type) ...
end

As we describe each particular type of node from now on, the data structure we give for it will list only the extra information for that type of node; remember that all nodes in the beta part of the network also have *type*, *children*, and *parent* fields. Also, we will simply write *left-activation* or *right-activation* as shorthand for the appropriate `switch` or `case` statement or jumptable usage.

Returning to beta memory nodes now, the only extra data a beta memory node stores is a list of the tokens it contains:

structure beta-memory:
    items: **list** of token
end

Whenever a beta memory is informed of a new match (consisting of an existing token and some WME), we build a token, add it to the list in the beta memory, and inform each of the beta memory's children:

```
procedure beta-memory-left-activation (node: beta-memory, t: token, w: WME)
    new-token ← allocate-memory()
    new-token.parent ← t
    new-token.wme ← w
    insert new-token at the head of node.items
    for each child in node.children do left-activation (child, new-token)
end
```

2.3.3 P-Node Implementation

We mention the implementation of production nodes (p-nodes) here because it is often similar to that of memory nodes in some respects. The implementation of p-nodes tends to vary from one system to another, so our discussion here will be rather general and we will not give pseudocode. A p-node may store tokens, just as beta memories do; these tokens represent complete matches for the production's conditions. (In traditional production systems, the set of all tokens at all p-nodes represents the *conflict set*.) On a left activation, a p-node will build a new token, or some similar representation of the newly found complete match. It then signals the new match in some appropriate (system-dependent) way.

In general, a p-node also contains a specification of what production it corresponds to — the name of the production, its right-hand-side actions, etc. A p-node may also contain information about the names of the variables that occur in the production. Note that variable names are not mentioned in any of the Rete node data structures we describe in this chapter. This is intentional — it enables nodes to be shared when two productions have conditions with the same basic form, but with different variable names. If variable names are recorded somewhere, it is possible to reconstruct the LHS of a production by looking at (its portion of) the Rete network together with the variable name information. The ability to reconstruct the LHS eliminates the need to save an "original copy" of the LHS in case we need to examine the production later.

2.4 Join Node Implementation

As mentioned in the overview, a join node can incur a right activation when a WME is added to its alpha memory, or a left activation when a token is added to its beta memory. In either case, the node's *other* memory is searched for items having variable bindings consistent with the new item; if any are found, they are passed on to the join node's children.

The data structure for a join node, therefore, must contain pointers to its two memory nodes (so they can be searched), a specification of any variable binding consistency tests to be performed, and a list of the node's children. From the data common to all nodes (the *rete-node* structure on page 22), we already have the *children*; also, the *parent* field automatically gives us a pointer to the join node's beta memory (the beta memory is always its parent). We need two extra fields for a join node:

structure join-node:
    amem: alpha-memory {*points to the alpha memory this node is attached to*}
    tests: **list** of test-at-join-node
end

The *test-at-join-node* structure specifies the locations of the two fields whose values must be equal in order for some variable to be bound consistently:

structure test-at-join-node:
    field-of-arg1: "identifier", "attribute", or "value"
    condition-number-of-arg2: **integer**
    field-of-arg2: "identifier", "attribute", or "value"
end

Arg1 is one of the three fields in the WME (in the alpha memory), while *arg2* is a field from a WME that matched some earlier condition in the production (i.e., part of the token in the beta memory). For example, in our example production

```
(find-stack-of-two-blocks-to-the-left-of-a-red-block
  (<x> ^on <y>)           /* C1 */
  (<y> ^left-of <z>)      /* C2 */
  (<z> ^color red)        /* C3 */
 -->
  ... RHS ...
),
```

the join node for `C3`, checking for consistent bindings of `<z>`, would have *field-of-arg1* = *"identifier"*, *condition-number-of-arg2* = *2*, and *field-of-arg2* = *"value"*, since the contents of the id field of the WME from the join node's alpha memory must be equal to the contents of the value field of the WME that matched the second condition.[7]

Upon a right activation (when a new WME w is added to the alpha memory), we look through the beta memory and find any token(s) t for which all these t-versus-w tests succeed. Any successful $\langle t, w \rangle$ combinations are passed on to the join node's children. Similarly, upon a left activation (when a new token t is added to the beta memory), we look through the alpha memory and find any WME(s) w for which all these t-versus-w tests succeed. Again, any successful $\langle t, w \rangle$ combinations are passed on to the node's children:

procedure join-node-right-activation (node: join-node, w: WME)
    for each t **in** node.parent.items **do** {*"parent" is the beta memory node*}
        if perform-join-tests (node.tests, t, w) **then**
            for each child **in** node.children **do** left-activation (child, t, w)
end

[7] Actually, with list-form tokens, it is convenient to have *condition-number-of-arg2* specify the *relative* condition number, i.e., the number of conditions in between the one containing *arg1* and the one containing *arg2*.

```
procedure join-node-left-activation (node: join-node, t: token)
    for each w in node.amem.items do
        if perform-join-tests (node.tests, t, w) then
            for each child in node.children do left-activation (child, t, w)
end

function perform-join-tests (tests: list of test-at-join-node, t: token, w: WME)
returning boolean
    for each this-test in tests do
        arg1 ← w.[this-test.field-of-arg1]
        {With list-form tokens, the following statement is really a loop[8]}
        wme2 ← the [this-test.condition-number-of-arg2]'th element in t
        arg2 ← wme2.[this-test.field-of-arg2]
        if arg1 ≠ arg2 then return false
    return true
end
```

We note a few things about the above procedures. First, in order to be able to use these procedures for the uppermost join nodes in the network — the ones that are children of the dummy top node, as in Figure 2.2 on page 10 — we need to have the dummy top node act as a beta memory for these join nodes. We always keep a single *dummy top token* in the dummy top node, just so there will be one thing to iterate over in the *join-node-right-activation* procedure. Second, this pseudocode assumes that all the tests are for *equality* between two fields. It is straightforward to extend the *test-at-join-node* structure and the *perform-join-tests* procedure to support other tests (e.g., tests requiring one field to be numerically less than another field). Most, if not all, implementations of Rete support such tests. Finally, this pseudocode assumes that the alpha and beta memories are not indexed in any way, as discussed above in Section 2.3. If indexed memories are used, then the activation procedures above would be modified to use the index rather than simply iterating over all tokens or WMEs in the memory node. For example, if memories are hashed, the procedures would iterate only over the tokens or WMEs in the appropriate hash bucket, not over all tokens or WMEs in the memory. This can significantly speed up the Rete algorithm.

2.4.1 Avoiding Duplicate Tokens

Whenever we add a WME to an alpha memory, we right-activate each join node attached to that alpha memory. In our discussions so far, one very important detail has been omitted. It turns out that the *order in which the join nodes are right-activated* can be crucial, because duplicate

[8] With list-form tokens, we need to follow the *parent* pointers on *token* data structures up a certain number of levels in order to find the *condition-number-of-arg2*'th element; the number of levels is equal to the number of the current condition (the one this join node is handling) minus *condition-number-of-arg2*. For convenience, we usually replace the *condition-number-of-arg2* field of the *test-at-join-node* structure with a *number-of-levels-up* field.

tokens (two or more tokens in the same beta memory representing the same sequence of WMEs) can be generated if the join nodes are activated in the wrong order.

To see how this can happen, consider a production whose LHS starts with the following three conditions:

```
(<x> ^self <y>)    /* C1 */
(<x> ^color red)   /* C2 */
(<y> ^color red)   /* C3 */
...                /* other conditions */
```

Suppose working memory initially contains just one WME, matching the first condition:

```
w1: (B1 ^self B1)
```

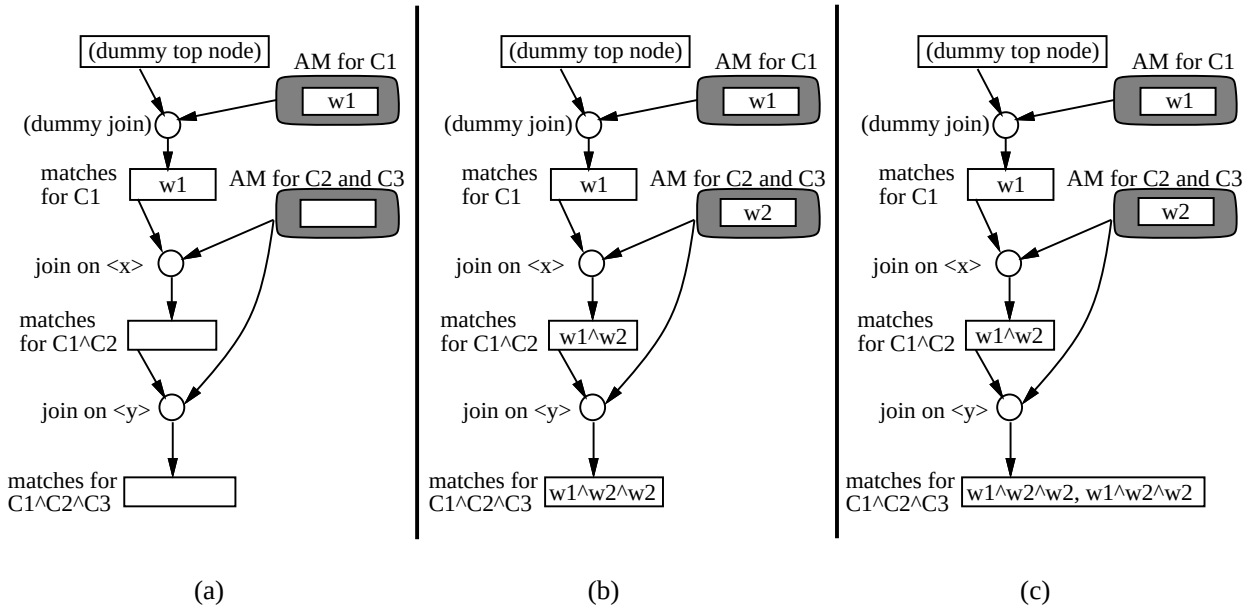
Figure 2.5 (a) shows the Rete net for this situation, including the contents of all the memory nodes. Notice that the lower alpha memory is used for two different conditions in this production, `C2` and `C3`. Now suppose another WME is added to working memory:

```
w2: (B1 ^color red)
```

This WME gets filtered through the alpha network and the *alpha-memory-activation* procedure is called to add `w2` to the lower alpha memory. We add `w2` to the memory's *items* list, and we then have to right-activate two join nodes (the one that joins on the values of `<x>`, and the one that joins on the values of `<y>`). Suppose we right-activate the higher one first (the one for `<x>`). It searches its beta memory for appropriate tokens, finds one, and passes this new match $\langle w1, w2 \rangle$ on to its child (the beta memory containing matches for `C1^C2`). There it is added to the memory and passed on to the lower join node. Now this join node searches its alpha memory for appropriate WMEs — and finds `w2`, which was just added to the memory — and passes this new match $\langle w1, w2, w2 \rangle$ on to its child. This concludes the processing triggered by the right-activation of the higher join node; this situation is shown in Figure 2.5 (b). We still have to right-activate the lower join node. When we right-activate it, it searches its beta memory for appropriate tokens — and finds $\langle w1, w2 \rangle$, which was just added to the memory. So it then passes this "new" match $\langle w1, w2, w2 \rangle$ on to its child, without realizing that this is a duplicate of the match found just before. The final result is shown in Figure 2.5 (c); note that the bottom beta memory contains two copies of the same token.

One way to deal with this problem would be to have beta memory nodes check for duplicate tokens. Every time a beta memory was activated, it would check whether the "new" match was actually a duplicate of some token already in the memory; if so, it would be ignored (i.e., discarded). Unfortunately, this would significantly slow down the handling of beta memory activations.

A better approach which avoids this slowdown is to right-activate the join nodes in a different order. In the above example, if we right-activate the lower join node first, no duplicate tokens are generated. (The reader is invited to check this; the key is that when the lower join node is right-activated, its beta memory is still empty.) In general, the solution is to right-activate



(a) (b) (c)

Figure 2.5: Duplicate tokens arising when join nodes are activated from the same alpha memory in the wrong order: (a) before addition of `w2` to the alpha memory; (b) after right activation of the join on `<x>` and subsequent processing, but before right activation of the join on `<y>`; (c) after right activation of the join on `<y>`.

descendents before their ancestors; i.e., if we need to right-activate join nodes J_1 and J_2 from the same alpha memory, and J_1 is a descendent of J_2, then we right-activate J_1 before right-activating J_2. Our overall procedure for adding a WME to an alpha memory, then, is:

1. Add the WME to the alpha memory's *items* list.
2. Right-activate the attached join nodes, *descendents before ancestors.*

The dual approach — right-activate ancestors before descendents, and *then* add the WME to the memory's *items* list — also works. For a full discussion of this issue, see (Lee and Schor, 1992).[9]

Of course, it would be foolish to traverse the Rete network on every alpha memory activation just to look for ancestor-descendent relationships among join nodes. Instead, we check these ahead of time, when the network is built, and make sure that the alpha memory's list of join nodes is appropriately *ordered*: if J_1 and J_2 are on the list and J_1 is a descendent of J_2, then J_1 must come *earlier* in the list than J_2. By enforcing this ordering in advance, we avoid any extra processing during node activations later, and avoid generating any duplicate tokens.

We will return to this issue in Chapter 4 when we describe *right unlinking*, an optimization in which join nodes will be dynamically spliced into and out of the lists on alpha memories. We will need to ensure that the appropriate ordering is maintained as we do this splicing.

[9]Note that the discussion in Forgy's thesis (Forgy, 1979, pages 66–68) is incorrect — Forgy's solution works only if one of the conditions sharing the alpha memory happens to be the first condition in the production. As our example shows, the problem is more general than this.

2.5 Removals of WMEs

When a WME is removed from working memory, we need to update the items in alpha and beta memory nodes (and tokens in p-nodes) accordingly, by removing any entries involving that WME. There are several ways of accomplishing this.

In the original Rete algorithm, removals were handled in essentially the same way as additions. We will refer to this method as *rematch-based removal*. The basic idea is that each procedure in the interpreter takes an extra argument, called a *tag*, a flag indicating whether the current operation is an addition or a deletion. Upon being called for a deletion, the procedures for alpha and beta memory nodes simply delete the indicated WME or token from their memories instead of adding it. The procedures then call their successors, just as they would for an addition, only with the tag being `delete` rather than `add`. The procedures for join nodes handle additions and deletions exactly the same — in both cases, they look for items in the opposite memory with consistent variable bindings and pass any matches (along with the `add`/`delete` tag) on to their children. Because this rematch-based removal method handles deletions with largely the same interpreter procedures as additions, it is simple and elegant.

Unfortunately, it is also slow, at least relative to other possible methods. With rematch-based removal, the cost of removing a WME is the same as the cost of adding a WME, since the same procedures get called and each does roughly the same amount of work. The problem is that none of the information obtained during the addition of a new WME is utilized during the later removal of that WME. There are at least three ways of handling removals which make use of such information.

In *scan-based removal*, we dispense with redoing variable binding consistency checks in the join nodes, and instead simply scan their output memories (their child beta memories and/or p-nodes) looking for any entries involving the item being removed. When a join node is right-activated for the removal of a WME w, it simply passes w on to its output memory (or memories); the memory looks through its list of tokens for any tokens whose last element happens to be w, deletes those tokens, and sends those deletions on to its children. Similarly, when a join node is left-activated for a removal of a token t, it passes t on to its output memory; the memory looks through its list of tokens for any whose parent happens to be t, deletes those tokens, and sends those deletions on to its children. Note that part of this procedure — looking for tokens whose parent is t — can be done efficiently only if list-form tokens are used rather than array-form tokens. (Scales, 1986) obtained a 28% speedup by replacing rematch-based removal with this scan-based removal technique in Soar; (Barachini, 1991) reports a 10% speedup from replacing rematch-based removal with a slight variant of this technique (Barachini and Theuretzbacher, 1988).

Probably the fastest way to handle a removal is to have written down ahead of time precisely which things need to be deleted. This straightforward idea is the basis of both *list-based removal* and *tree-based removal*. The idea is to keep some extra pointers on the data structures for WMEs and/or tokens so that when a WME is removed, we can find *all* the tokens that need to be deleted — and *only* those ones that need to be deleted — just by following pointers.

In list-based removal, proposed by (Scales, 1986), we store on every WME w a list of all the tokens involving w. Then when w is removed, we simply iterate over this list and delete

each token on it. The drawbacks to this are the large amount of extra space it uses and the potentially large amount of extra time it takes to create a token: a new token $\langle w_1, \ldots, w_i \rangle$ must be added to the lists on each of the i WMEs. However, if tokens are represented by arrays rather than lists, then both the space and time would be just a constant factor overhead, since creating this new token would require creating a new i-element array anyway. Thus, list-based removal might be acceptable in such implementations. There are no empirical results for list-based removal reported in the literature, so it is unclear whether it would work well in practice (or even whether it has ever been implemented).

In tree-based removal, on every WME w, we keep a list of all the tokens for which w is the last element. On every token t, we keep a list of all the children of t. These pointers to children allow us to find all of t's descendents in lower beta memories and production nodes. (Recall from Section 2.3 that with list-form tokens, the set of all tokens forms a tree.) Now when w is removed, we simply traverse a bunch of subtrees (of "root" tokens and their descendents), deleting everything in them. Of course, all these extra pointers mean more memory usage, plus extra time spent setting up these pointers ahead of time when WMEs are added or tokens are created. Empirically, however, the time savings during WME removals more than makes up for the extra time spent setting up the pointers beforehand. When the author replaced rematch-based removal with tree-based removal in Soar, it sped up the matcher by a factor of about 1.3; (Barachini, 1991) estimated a speedup factor of 1.25 for an OPS5-like system.

To implement tree-based removal, we revise the data structure for each WME to have it include a list of all the alpha memories containing the WME, and a list of all the tokens having the WME as the last element:

```
structure WME {revised from version on page 21}
    fields: array [1..3] of symbol
    alpha-mems: list of alpha-memory {the ones containing this WME}
    tokens: list of token {the ones with wme=this WME}
end
```

The data structure for a token is expanded to contain a pointer to the memory node it resides in (we'll use this in the *delete-token-and-descendents* procedure below), and a list of its children:

```
structure token {revised from version on page 22}
    parent: token {points to the higher token, for items 1...i-1}
    wme: WME {gives item i}
    node: rete-node {points to the memory this token is in}
    children: list of token {the ones with parent=this token}
end
```

We now modify the *alpha-memory-activation* and *beta-memory-left-activation* procedures so they set up these lists ahead of time. Whenever a WME w is added to an alpha memory a, we add a to w.*alpha-mems*.

```
procedure alpha-memory-activation (node: alpha-memory, w: WME)
{revised from version on page 21}
    insert w at the head of node.items
    insert node at the head of w.alpha-mems {for tree-based removal}
    for each child in node.successors do right-activation (child, w)
end
```

Similarly, whenever a new token $tok = \langle t, w \rangle$ is added to a beta memory, we add *tok* to *t.children* and to *w.tokens*. We also fill in the new *node* field on the token. To simplify our pseudocode, it is convenient to define a "helper" function *make-token* which builds a new token and initializes its various fields as necessary for tree-based removal. Although we write this as a separate function, it would normally be coded "inline" for efficiency.[10]

```
function make-token (node: rete-node, parent: token, w: wme)
returning token
    tok ← allocate-memory()
    tok.parent ← parent
    tok.wme ← w
    tok.node ← node {for tree-based removal}
    tok.children = nil {for tree-based removal}
    insert tok at the head of parent.children {for tree-based removal}
    insert tok at the head of w.tokens {for tree-based removal}
    return tok
end

procedure beta-memory-left-activation (node: beta-memory, t: token, w: WME)
{revised from version on page 23}
    new-token ← make-token (node, t, w)
    insert new-token at the head of node.items
    for each child in node.children do left-activation (child, new-token)
end
```

Now, to remove a WME, we just remove it from each alpha memory containing it (these alpha memories are now conveniently on a list) and call the helper routine *delete-token-and-descendents* to delete all the tokens involving it (all the necessary "root" tokens involving it are also conveniently on a list):

```
procedure remove-wme (w: WME)
    for each am in w.alpha-mems do remove w from the list am.items
    while w.tokens ≠ nil do
        delete-token-and-descendents (the first item on w.tokens)
end
```

[10] "Inline" means that instead of creating a separate procedure, the body of the function is simply inserted everywhere the function would be called. This avoids incurring a procedure call overhead.

Note the use of a "while" loop over *w.tokens* rather than a "for" loop. A "for" loop would be unsafe here, because each call to *delete-token-and-descendents* destructively modifies the *w.tokens* list as it deallocates the memory used for *token* data structures. A "for" loop would maintain a pointer into the middle of this list as it ran, and the pointer could become invalid due to this destructive modification.

The helper routine *delete-token-and-descendents* removes a token together with its entire tree of descendents. For simplicity, the pseudocode here is recursive; an actual implementation may be made slightly faster by using a nonrecursive tree traversal method.

```
procedure delete-token-and-descendents (tok: token)
    while tok.children ≠ nil do
        delete-token-and-descendents (the first item on tok.children)
    remove tok from the list tok.node.items
    remove tok from the list tok.wme.tokens
    remove tok from the list tok.parent.children
    deallocate memory for tok
end
```

2.5.1 On the Implementation of Lists

In many of the data structures here, we have one or more fields which hold a "**list** of such-and-such." For some of these, the representation of the list is very important. Take the list of tokens in a beta memory, for example, and consider the following line from the *delete-token-and-descendents* procedure above:

remove tok from the list tok.node.items

This says to splice *tok* out of the list containing it. If this list is singly-linked, then splicing out *tok* requires iterating over all the earlier tokens in the list in order to find the token whose *next* field should be modified. If the list is doubly-linked instead, then it is straightforward to splice out *tok* without using a loop.

In general, most if not all of the lists we use in Rete will be doubly-linked. The *next* and *previous* fields will, whenever possible, be contained in the data structures representing the items in the list. For example, since every token is on three lists (a memory node's *items* list, a WME's *tokens* list, and its parent token's *children* list), the data structure for a token will have six link fields: *next-in-this-memory*, *previous-in-this-memory*, *next-from-this-wme*, *previous-from-this-wme*, *next-from-parent*, and *previous-from-parent*. This may seem like it makes the *token* data structure unduly large, but there is no better alternative — every token is on these three lists, and for efficiency, each list must be doubly linked, so a minimum of six pointers per token are required. To keep our pseudocode simple, we will omit these extra link fields from our data structure declarations, and will continue to write statements like the one above with the understanding that this is to be implemented by splicing the item into or out of a doubly-linked list.

Sometimes we cannot know in advance how many different lists an item will be on at once. For example, a given WME *w* could be in many different alpha memories, hence could be on many different alpha memory nodes' *items* lists. So we can't reserve space in *w*'s data structure for *next* and *previous* pointers. Yet for tree-based removal, we need to be able to quickly splice *w* out of each of these lists. The obvious solution is to represent each alpha memory's *items* list using a doubly-linked set of auxiliary data structures:

```
structure alpha-memory {revised from version on page 21}
    items: list of item-in-alpha-memory
    successors: list of rete-node
end

structure item-in-alpha-memory
    wme: WME {the WME that's in the memory}
    amem: alpha-memory {points to the alpha memory node}
    (also: next, previous: pointer to item-in-alpha-memory {for the doubly-linked list})
end
```

Then on each WME, rather than having *alpha-mems* be a list of the memories containing it, we instead have a list of the *item-in-alpha-memory* structures involving it:

```
structure WME {revised from version on page 29}
    fields: array [1..3] of symbol
    alpha-mem-items: list of item-in-alpha-memory {the ones with wme=this WME}
    tokens: list of token {the ones with wme=this WME}
end
```

Finally, we modify various procedures that access alpha memories' *items* lists, so they handle the new format of *items* lists correctly.

```
procedure join-node-left-activation (node: join-node, t: token)
{revised from version on page 25}
    for each item in node.amem.items do
        if perform-join-tests (node.tests, t, item.wme) then
            for each child in node.children do left-activation (child, t, item.wme)
end

procedure alpha-memory-activation (node: alpha-memory, w: WME)
{revised from version on page 30}
    new-item ← allocate-memory()
    new-item.wme ← w; new-item.amem ← node;
    insert new-item at the head of node.items
    insert new-item at the head of w.alpha-mem-items
    for each child in node.successors do right-activation (child, w)
end
```

```
procedure remove-wme (w: WME) {revised from version on page 30}
    for each item in w.alpha-mem-items do
        remove item from the list item.amem.items
        deallocate memory for item
    while w.tokens ≠ nil do
        delete-token-and-descendents (the first item on w.tokens)
end
```

2.6 Adding and Removing Productions

It is sometimes mistakenly claimed that Rete does not allow the run-time addition or deletion of productions, or that it requires recompilation of the entire network. This misunderstanding may have arisen from people's experience with certain implementations that did not support run-time production addition or deletion — OPS5, for instance, only partially supports this,[11] and most compiled versions of Rete do not support this at all (though (Tambe et al., 1988) is an exception). The basic Rete algorithm does not preclude run-time additions and deletions, though; in fact, it is quite straightforward to modify the network "on the fly" when productions are added or deleted.

The basic method for adding a production with conditions $c_1, \ldots, c_k$ is to start at the top of the beta network and work our way down, building new memory and join nodes (or finding existing ones to share, if possible) for $c_1, \ldots, c_k$, in that order. We assume that the ordering of the conditions is given to us in advance. At a very high level, the procedure looks like this:

```
M_1 ← dummy-top-node
build/share J_1 (a child of M_1), the join node for c_1
for i = 2 to k do
    build/share M_i (a child of J_{i-1}), a beta memory node
    build/share J_i (a child of M_i), the join node for c_i
make P (a child of J_k), the production node
```

This procedure handles only the beta part of the net; we will also need to build or share an alpha memory for each condition as we go along.

We will use several helper functions to make the main *add-production* procedure simpler. The first one, *build-or-share-beta-memory-node*, looks for an existing beta memory node that is a child of the given *parent* node. If there is one, it returns it so it can be shared by the new production; otherwise the function builds a new one and returns it. This pseudocode assumes that beta memories are not indexed; if indexing is used, the procedure would take an extra argument specifying which field(s) the memory must be indexed on.

[11] Lisp-based versions of OPS5 allow productions to be added at run-time by using the `build` RHS action (Forgy, 1981). However, newly added productions are not matched against *existing* WMEs, only against WMEs added thereafter (Schor et al., 1986).

```
function build-or-share-beta-memory-node (parent: rete-node)
returning rete-node
    for each child in parent.children do {look for an existing node to share}
        if child is a beta memory node then return child
    new ← allocate-memory()
    new.type ← "beta-memory"
    new.parent ← parent; insert new at the head of the list parent.children
    new.children ← nil
    new.items ← nil
    update-new-node-with-matches-from-above (new) {see page 38}
    return new
end
```

The *update-new-node-with-matches-from-above* procedure initializes the memory node to store tokens for any existing matches for the earlier conditions.

The next helper function is similar, except it handles join nodes rather than beta memory nodes. The two additional arguments specify the alpha memory to which the join node must be attached and the variable binding consistency checks it must perform. Note that there is no need to call *update-new-node-with-matches-from-above* in this case, because a join node does not store any tokens, and a newly created join node has no children onto which join results should be passed.

```
function build-or-share-join-node (parent: rete-node, am: alpha-memory,
                                   tests: list of test-at-join-node)
returning rete-node
    for each child in parent.children do {look for an existing node to share}
        if child is a join node and child.amem=am and child.tests=tests then
            return child
    new ← allocate-memory()
    new.type ← "join"
    new.parent ← parent; insert new at the head of the list parent.children
    new.children ← nil
    new.tests ← tests; new.amem ← am
    insert new at the head of the list am.successors
    return new
end
```

Our next helper function, *get-join-tests-from-condition*, takes a condition and builds a list of all the variable binding consistency tests that need to be performed by its join node. To do this, it needs to know what all the earlier conditions are, so it can determine whether a given variable appeared in them — in which case its occurrence in the current condition means a consistency test is needed — or whether it is simply a new (not previously seen) variable — in which case no test is needed. If a variable v has more than one previous occurrence, we still only need one consistency test for it — join nodes for earlier conditions will ensure that all the previous

occurrences are equal, so the current join node just has to make sure the current WME has the same value for it as any one of the previous occurrences. The pseudocode below always chooses the *nearest* (i.e., most recent) occurrence for the test, because with list-form tokens, the nearest occurrence is the cheapest to access. With array-form tokens, this choice does not matter.

```
function get-join-tests-from-condition (c: condition, earlier-conds: list of condition)
returning list of test-at-join-node
    result ← nil
    for each occurrence of a variable v in a field f of c do
        if v occurs anywhere in earlier-conds then
            let i be the largest i such that the i^th condition in earlier-conds contains
                    a field f2 in which v occurs
            this-test ← allocate-memory()
            this-test.field-of-arg1 ← f
            this-test.condition-number-of-arg2 ← i
            this-test.field-of-arg2 ← f2
            append this-test to result
    return result
end
```

Finally, we have a helper function for creating a new alpha memory for a given condition, or finding an existing one to share. The implementation of this function depends on what type of alpha net implementation is used. If we use a traditional dataflow network, as described in Section 2.2.1, then we simply start at the top of the alpha network and work our way down, sharing or building new constant test nodes:

```
function build-or-share-alpha-memory (c: condition) {dataflow network version}
returning alpha-memory
    current-node ← top-node-of-alpha-network
    for each constant test in each field of c do
        let sym be the symbol tested for, and f be the field
        current-node ← build-or-share-constant-test-node (current-node, f, sym)
    if current-node.output-memory ≠ nil then return current-node.output-memory
    am ← allocate-memory()
    current-node.output-memory ← am
    am.successors ← nil ; am.items ← nil
    {initialize am with any current WMEs}
    for each WME w in working memory do
        if w passes all the constant tests in c then alpha-memory-activation (am, w)
    return am
end
```

```
function build-or-share-constant-test-node (parent: constant-test-node, f: field,
                                             sym: symbol )
returning constant-test-node
    {look for an existing node we can share}
    for each child in parent.children do
        if child.field-to-test = f and child.thing-the-field-must-equal = sym
            then return child
    {couldn't find a node to share, so build a new one}
    new ← allocate-memory()
    add new to the list parent.children
    new.field-to-test ← f; new.thing-the-field-must-equal ← sym
    new.output-memory ← nil ; new.children ← nil
    return new
end
```

The reader can easily extend the above procedure to handle the dataflow-network-plus-hashing implementation of the alpha network described in Section 2.2.2. For the exhaustive-hash-table-lookup implementation described in Section 2.2.3, the procedure is much simpler, as there is no network and all we have to deal with is a hash table:

```
function build-or-share-alpha-memory (c: condition) {exhaustive table lookup version}
returning alpha-memory
    {figure out what the memory should look like}
    id-test ← nil ; attr-test ← nil ; value-test ← nil
    if a constant test t occurs in the "id" field of c then id-test ← t
    if a constant test t occurs in the "attribute" field of c then attr-test ← t
    if a constant test t occurs in the "value" field of c then value-test ← t
    {is there an existing memory like this?}
    am ← lookup-in-hash-table (id-test, attr-test, value-test)
    if am ≠ nil then return am
    {no existing memory, so make a new one}
    am ← allocate-memory()
    add am to the hash table for alpha memories
    am.successors ← nil ; am.items ← nil
    {initialize am with any current WMEs}
    for each WME w in working memory do
        if w passes all the constant tests in c then alpha-memory-activation (am ,w)
    return am
end
```

One final note here: whenever we create a new alpha memory, we initialize it by adding to it any appropriate WMEs in the current working memory. The pseudocode above does this by iterating over the entire WM and checking each WME. It is often possible to do this much faster. If there is already another alpha memory that stores a superset of the WMEs the new

one should, we can iterate just over that other alpha memory's contents, rather than over all of WM. For instance, if we are creating a new alpha memory for `(* ^color red)`, we could just check the WMEs in the alpha memory for `(* ^color *)` (if such an alpha memory can be found).

We are now ready to give the *add-production* procedure, which takes a production (actually, just the conditions of the production — the match algorithm doesn't care what the actions are) and adds it to the network. It follows the basic procedure given at the beginning of this section, and uses the helper functions we have just defined. The last two lines of the procedure are a bit vague, because the implementation of production nodes tends to vary from one system to another. It is important to note that we build the net *top-down*, and each time we build a new join node, we insert it at the *head* of its alpha memory's list of *successors*; these two facts guarantee that descendents are on each alpha memory's list before any of their ancestors, just as we required in Section 2.4.1 in order to avoid duplicate tokens.

```
procedure add-production (lhs: list of conditions)
    let the lhs conditions be denoted by c1, ..., ck
    current-node ← dummy-top-node
    earlier-conditions ← nil

    tests ← get-join-tests-from-condition (c1, earlier-conditions)
    am ← build-or-share-alpha-memory (c1)
    current-node ← build-or-share-join-node (current-node, am, tests)

    for i = 2 to k do
        {get the beta memory node Mi }
        current-node ← build-or-share-beta-memory-node (current-node)
        {get the join node Ji for condition ci }
        append ci-1 to earlier-conditions
        tests = get-join-tests-from-condition (ci, earlier-conditions)
        am ← build-or-share-alpha-memory (ci)
        current-node ← build-or-share-join-node (current-node, am, tests)

    build a new production node, make it a child of current-node
    update-new-node-with-matches-from-above (the new production node)
end
```

Finally, we give the *update-new-node-with-matches-from-above* procedure. This is needed to ensure that newly added productions are immediately matched against the current working memory. The procedure's job is to ensure that the given *new-node*'s left-activation procedure is called with all the existing matches for the previous conditions, so that the *new-node* can take any appropriate actions (e.g., a beta memory stores the matches as new tokens, and a p-node signals new complete matches for the production). How *update-new-node-with-matches-from-above* achieves this depends on what kind of node the *new-node*'s parent is. If the parent is a beta memory (or a node for a negated condition, as we will discuss later), this is straightforward,

since the parent has a list (*items*) of exactly the matches we want. But if the parent node is a join node, we want to find the matches satisfying the join tests, and these may not be recorded anywhere. To find these matches, we iterate over the WMEs and tokens in the join node's alpha and beta memories and perform the join tests on each pair. The pseudocode below uses a trick to do this: while temporarily pretending the *new-node* is the only child of the join node, it runs the join node's right-activation procedure for all the WMEs in its alpha memory; any new matches will automatically be propagated to the *new-node*. For a variation of this implementation, see (Tambe et al., 1988); for a general discussion, see (Lee and Schor, 1992).

```
procedure update-new-node-with-matches-from-above (new-node: rete-node)
    parent ← new-node.parent
    case parent.type of
        "beta-memory":
            for each tok in parent.items do left-activation (new-node, tok)
        "join":
            saved-list-of-children ← parent.children
            parent.children ← [new-node] {list consisting of just new-node}
            for each item in parent.amem.items do
                right-activation (parent, item.wme)
            parent.children ← saved-list-of-children
end
```

To remove an existing production from the network, we start down at the bottom of the beta network, at the p-node for that production. The basic idea is to start walking from there up to the top of the net. At each node, we clean up any tokens it contains, and then get rid of the node — i.e., remove it from the *children* or *successors* lists on its predecessors (its parent and, for some nodes, its alpha memory as well), and deallocate it. We then move up to the predecessors. If the alpha memory is not being shared by another production, we deallocate it too. If the parent is not being shared by another production, then we apply the same procedure to it — clean up its tokens, etc. — and repeat this until we reach either a node being shared by some other production, or the top of the beta network.

```
procedure remove-production (prod: production)
    delete-node-and-any-unused-ancestors (the p-node for prod)
end
```

```
procedure delete-node-and-any-unused-ancestors (node: rete-node)
    if node is a join node then {for join nodes, deal with the alpha memory}
        remove node from the list node.amem.successors
        if node.amem.successors=nil then delete-alpha-memory (node.amem)
    else {for non-join nodes, clean up any tokens it contains}
        while node.items ≠ nil do
            delete-token-and-descendents (first item on node.items)
    remove node from the list node.parent.children
    if node.parent.children=nil then
        delete-node-and-any-unused-ancestors (node.parent)
    deallocate memory for node
end
```

The *delete-alpha-memory* procedure cleans up and deletes a given alpha memory (together with any now-unused constant test nodes, if a dataflow implementation is used for the alpha network). For brevity, we do not give pseudocode for this here; the procedure for this alpha network cleanup is straightforward and is analogous to the procedure just given for beta network cleanup.

2.7 Negated Conditions

So far we have been discussing conditions that test for the *presence* of a WME in working memory. We now move on to discuss conditions testing for the *absence* of items in working memory. In this section, we discuss *negated conditions*, which test for the absence of a certain WME; in Section 2.8, we discuss *negated conjunctive conditions*, which test for the absence of a certain combination of WMEs.

Consider our earlier example production, only with its last condition negated (indicated by the "-" sign preceding it):

```
 (<x> ^on <y>)           /* C1 */
 (<y> ^left-of <z>)      /* C2 */
-(<z> ^color red)        /* C3 */
```

This production matches if there is a stack of (at least) two blocks (designated by `<x>` and `<y>`) to the left of some block (designated by `<z>`) which is *not* known to be red. To implement this, we need a node for `C3` which will take a match $\langle w_1, w_2 \rangle$ for the first two conditions, and will propagate it further down the beta network *if and only if* there is no WME w_3 whose id field contains the same thing `<z>` is bound to in $\langle w_1, w_2 \rangle$, whose attribute field contains `color`, and whose value field contains `red`.

The standard way of doing this is to use a different type of Rete node, called a *negative node* or *not node*, for the negated condition, as illustrated in Figure 2.6. The negative node for a condition c_i stores all the matches (tokens) for the earlier conditions, just as a beta memory node would; it is linked into the network as a child of the join node for c_{i-1}, just as a beta memory

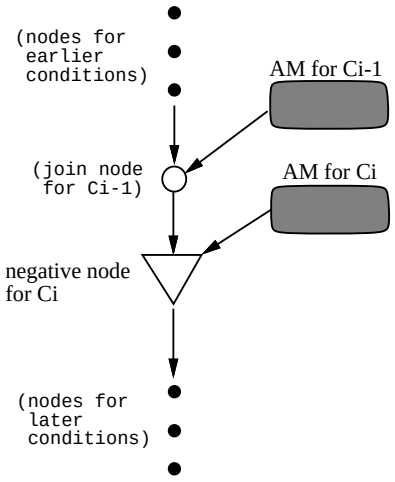


Figure 2.6: Rete network for a negative condition.

node would be. The negative node is also linked to an alpha memory, just as if c_i were a positive condition. In the example above, the negative node for `C3` would be linked to the alpha memory containing all WMEs of the form `(* ^color red)`. So far, this sounds like the negative node is just a combination of a beta memory and a join node. The negative node performs a join between the WMEs in its alpha memory and each token from the earlier conditions, just as a (positive) join node would; however, instead of propagating the results of the join down the network, it stores them in a local memory on each token. A token is then propagated down the network if and only if its local result memory is empty, since this indicates the absence of any such WME in working memory.[12]

Note that whereas the beta memories and join nodes for positive conditions are separate nodes, the memory and join functions for negative conditions are combined in a single node. This is done simply to save space. For positive conditions, these functions are separated in order to enable multiple join nodes (for different positive conditions) to share the same beta memory node. We cannot share the memory portion of a negative node with any other nodes, because we need to store local join results on each token in the memory, and these local join results can be different for different (negative) conditions.

To implement all this, we first add a *join-results* field to the data structure for a token; this field will store a list of *negative-join-result* data structures. The need for a list of these structures rather than just a list of WMEs will become clear when we discuss WME removals shortly. The extra *join-results* field will not be used for tokens in beta memories.

[12] The description here applies to Rete versions that use tree-based removal. If rematch-based removal is used, then we save on each token just a *count* of the number of join results; the results themselves are not stored anywhere. Then whenever a WME w is added to or removed from the alpha memory, the negative node is right-activated and we update the count on any token consistent with w. If the count on a token changes from zero to one or vice-versa, we pass the change on to the node's children.

structure token {*revised from version on page 29*}
    parent: token {*points to the higher token, for items 1...i-1*}
    wme: WME {*gives item i*}
    node: rete-node {*points to the node this token is in*}
    children: **list** of token {*the ones with* `parent`*=this token*}
    join-results: **list** of negative-join-result {*used only on tokens in negative nodes*}
end

A *negative-join-result* just specifies the $\langle t, w \rangle$ pair that is the result of the join:

structure negative-join-result
    owner: token {*the token in whose local memory this result resides*}
    wme: WME {*the WME that matches* `owner` }
end

We also add a field *negative-join-results* to the data structure for each WME, to store a list of all the *negative-join-result* structures involving the WME. This will be used for handling WME removals, as we discuss below.

structure WME {*revised from version on page 32*}
    fields: **array** [1..3] of **symbol**
    alpha-mem-items: **list** of item-in-alpha-memory {*the ones with* `wme`*=this WME*}
    tokens: **list** of token {*the ones with* `wme`*=this WME*}
    negative-join-results: **list** of negative-join-result
end

The data structure for a negative node looks like a combination of those for a beta memory and a join node:

structure negative-node:
    {*just like for a beta memory*}
    items: **list** of token
    {*just like for a join node*}
    amem: alpha-memory {*points to the alpha memory this node is attached to*}
    tests: **list** of test-at-join-node
end

Next, we have procedures for left and right activations of negative nodes. On a left activation (when there is a new match for all the earlier conditions), we build and store a new token, perform a join for the token, store the join results in the token structure, and pass the token onto any successor nodes if there were no join results.

```
procedure negative-node-left-activation (node: negative-node, t: token, w: WME)
    {build and store a new token, just like a beta memory would (page 30)}
    new-token ← make-token (node, t, w)
    insert new-token at the head of node.items

    {compute the join results}
    new-token.join-results ← nil
    for each item in node.amem.items do
        if perform-join-tests (node.tests, new-token, item.wme) then
            jr ← allocate-memory()
            jr.owner ← new-token; jr.wme ← w
            insert jr at the head of the list new-token.join-results
            insert jr at the head of the list w.negative-join-results

    {If join results is empty, then inform children}
    if new-token.join-results=nil then
        for each child in node.children do left-activation (child, new-token, nil )
end
```

Note that in the last line of the procedure, when we left-activate the children, we pass *nil* in place of the usual WME argument, since no actual WME was matched.[13] This means we need to revise our *make-token* and *delete-token-and-descendents* procedures to handle this properly. (An alternative implementation that avoids the special check in these two procedures is to pass a special dummy WME for negated conditions instead of nil.)

```
function make-token (node: rete-node, parent: token, w: wme)
returning token {revised from version on page 30}
    tok ← allocate-memory()
    tok.parent ← parent
    tok.wme ← w
    tok.node ← node {for tree-based removal}
    tok.children = nil {for tree-based removal}
    insert tok at the head of parent.children {for tree-based removal}
    if w ≠ nil then {we need this check for negative conditions}
        insert tok at the head of w.tokens {for tree-based removal}
    return tok
end
```

[13] If we use rematch-based removal, then in the *negative-node-left-activation* procedure above, we can just "skip a level" and pass $\langle t, w \rangle$ on to the children instead of $\langle$*new-token*, **nil**$\rangle$. If we use tree-based removal, though, we can't skip a level like this, since we need to be able to quickly locate the descendents of *new-token*; skipping a level would result in *new-token* always having *children=nil*.

```
procedure delete-token-and-descendents (tok: token)
{revised from version on page 31}
    while tok.children ≠ nil do
        delete-token-and-descendents (the first item on tok.children)
    remove tok from the list tok.node.items
    if tok.wme ≠ nil then {we need this check for negative conditions}
        remove tok from the list tok.wme.tokens
    remove tok from the list tok.parent.children
    deallocate memory for tok
end
```

On a right activation of a negative node (when a WME is added to its alpha memory), we look for any tokens in its memory consistent with the WME; for each such token, we add this WME to its local result memory. Also, if the number of results changes from zero to one — indicating that the negated condition was previously true but is now false — then we call the *delete-descendents-of-token* helper function to delete any tokens lower in the network that depend on this token.

```
procedure negative-node-right-activation (node: negative-node, w: WME)
    for each t in node.items do
        if perform-join-tests (node.tests, t, w) then
            if t.join-results=nil then delete-descendents-of-token (t)
            jr ← allocate-memory()
            jr.owner ← t; jr.wme ← w
            insert jr at the head of the list t.join-results
            insert jr at the head of the list w.negative-join-results
end


procedure delete-descendents-of-token (t: token)
    while t.children ≠ nil do
        delete-token-and-descendents (the first item on t.children)
end
```

The procedures given so far handle WME and token additions for negative nodes. We now discuss how removals are handled. When a WME is removed from working memory, we need to update the local join results on any tokens in negative nodes accordingly. We can efficiently locate all the join results that need to be updated by looking at the WME's *negative-join-results* field. We delete each such join result; as we are doing this, if the number of results in any token's local memory changes from one to zero — indicating that the negated condition was previously false but is now true — then we inform any children of the negative node containing the token.

```
procedure remove-wme (w: WME) {revised from version on page 33}
    for each item in w.alpha-mem-items do
        remove item from the list item.amem.items
        deallocate memory for item
    while w.tokens ≠ nil do
        delete-token-and-descendents (the first item on w.tokens)
    for each jr in w.negative-join-results do
        remove jr from the list jr.owner.join-results
        if jr.owner.join-results=nil then
            for each child in jr.owner.node.children do
                left-activation (child, jr.owner, nil )
        deallocate memory for jr
end
```

We also need to modify the *delete-token-and-descendents* procedure so it can correctly handle tokens in negative nodes — these require some extra cleanup code to deallocate all the local join results.

```
procedure delete-token-and-descendents (tok: token)
{revised from version on page 43}
    while tok.children ≠ nil do
        delete-token-and-descendents (the first item on tok.children)
    remove tok from the list tok.node.items
    if tok.wme ≠ nil then remove tok from the list tok.wme.tokens
    remove tok from the list tok.parent.children
    if tok.node is a negative node then
        for each jr in tok.join-results do
            remove jr from the list jr.w.negative-join-results
            deallocate memory for jr
    deallocate memory for tok
end
```

Finally, we need to extend our *add-production* and *remove-production* handling (Section 2.6) so it can accept negative conditions and create and delete negative nodes. This is straightforward and will not be presented here, but see Appendix A.

2.8 Conjunctive Negations

In this section, we show how to implement *negated conjunctive conditions* (NCC's), also called *conjunctive negations*, which test for the absence of a certain combination of WMEs. (Conjunctive negations with only one conjunct are semantically equivalent to the negated conditions discussed in the previous section.) Although the implementation of Rete used in this thesis

is not the first to support NCC's (Forgy, 1979; Allen, 1982; Schor et al., 1986; Tambe et al., 1988), to our knowledge no detailed description of their implementation has yet been given in the literature.

A conjunctive negation is usually written surrounded by set braces preceded by a minus sign:

```
(<x> ^on <y>)              /* C1 */
(<y> ^left-of <z>)         /* C2 */
-{ (<z> ^color red)        /* NCC comprising C3... */
  (<z> ^on <w>) }          /* ... and C4 */
```

This production matches if there is a stack of (at least) two blocks (designated by `<x>` and `<y>`) to the left of some block (designated by `<z>`) which is *not both* red and on some other block. Note that the occurrences of `<z>` *inside* the NCC refer to the binding of `<z>` from *outside* the NCC (in `C2`), while `<w>` is simply a new variable inside the NCC, since it does not occur anywhere in the (positive) conditions outside the NCC.

We will refer to the conjuncts in an NCC as its *subconditions*. There are no special restrictions about what can be a subcondition; they may be positive or negative conditions, or even NCC's — conjunctive negations may be nested to any depth. Importantly, the ability to nest conjunctive negations allows us to support conditions containing arbitrary combinations of $\forall$ and $\exists$ quantifiers: informally, ordinary (positive) conditions have the semantics $\exists x P(x)$, while $\forall x P(x)$ may be rewritten using a conjunctive negation as $\neg\exists x \neg P(x)$. For example, a check for whether *every red block has a blue block on top of it* can be rewritten as a check that *there is no red block that does not have a blue block on top of it*, i.e.,

```
-{ (<x> ^color red)
   -{ (<y> ^on <x>)
      (<y> ^color blue) } }
```

The basic idea behind the implementation of NCC's is the same as for the negated single conditions discussed in the previous section. With negated single conditions, for each incoming token (each match for the earlier conditions) we computed the results of a join as if the condition were not negated, saved the results in a local memory on the incoming token, and passed the dataflow on down the network if and only if there were no join results. We use a similar method for an NCC, except that instead of using a single node to compute the results of a single join, we use a subnetwork to compute the results of the subconditions inside the NCC. This is illustrated in Figure 2.7. Underneath the join node for the condition immediately preceding the NCC, we have a sequence of Rete nodes to handle the sequence of subconditions. The figure shows this as just beta memories and join nodes, but in general it could contain negative nodes or nodes for (nested) NCC's; the figure omits the alpha memories. We also use a pair of special nodes for the conjunctive negation. An *NCC node* is made a child of the join node for the condition preceding the NCC, and an *NCC partner node* is made a child of the bottom node in the subnetwork for the subconditions.[14] Note that when we have conjunctive negations, the beta part of the Rete net is no longer a tree as we stated in the overview. This fact will become important later, in Section 4.3.

[14]The reason for using *two* nodes is as follows. At this point in the network, we need to receive activations

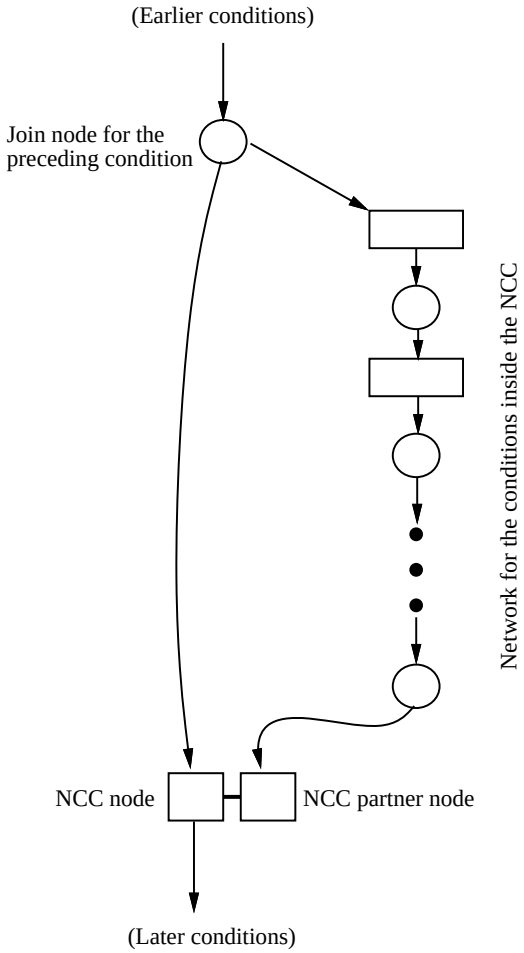


Figure 2.7: Rete network for a negated conjunctive condition.

Assume for the moment that we always order the preceding join node's *children* list so that the NCC node comes *before* the top node in the subnetwork; we will discuss node ordering issues shortly. The NCC node is left-activated whenever there is a new match for the earlier conditions; it stores this match as a new token and initializes the token's local result memory to be empty. The new token is then passed on to the NCC node's children (since the result memory is empty

from two sources: (1) the join node for the preceding condition, and (2) the bottom of the subnetwork. In both cases, (using our pseudocode) the activation will be executed via a *left-activation* (of such-and-such a node type) procedure call, but we need to handle these two kinds of activations in different ways. The solution we adopt here is to use a pair of nodes (and a pair of procedures: *ncc-node-left-activation* and *ncc-partner-node-left-activation*). Another possibility is to use a single NCC node that can be either left-activated or right-activated. However, this means join nodes must have two separate lists of children — one list of children to be left-activated and one list of children to be right-activated — so that the bottom join node in the subnetwork will right-activate the single NCC node while the join node preceding the subnetwork will left-activate it. This would require extra space on every join node. A third possible implementation saves a bit of space by making due with a single node and a single activation procedure — the first thing the procedure does is examine the incoming token and figure out "from which direction it came." This can be done, but it makes the code a bit more complex and a bit slower.

at this point). The NCC partner node serves as a collector of matches for the subconditions; each time a new match for them is found, the NCC partner node is left-activated. Its job is to add this new match to the local result memory of the appropriate token in the NCC node. (Each token in the NCC node must have its own local result memory, since the subconditions' results will vary depending on the variable bindings from the earlier conditions. For example, in the production above, different matches $\langle w_1, w_2 \rangle$ for the first two conditions may have different bindings for **<z>** and hence may yield different results from the NCC.) How do we figure out which token in the NCC node is the appropriate one? Consider the example production above. A match for the subconditions has the form $\langle w_1, w_2, w_3, w_4 \rangle$, where w_1 and w_2 match the first two conditions (above the NCC) and w_3 and w_4 match the two conditions inside the NCC. The appropriate token in the NCC node is the one representing $\langle w_1, w_2 \rangle$. So to find the appropriate token, the NCC partner node just strips off the last j WMEs from the subconditions' new match, where j is the number of conjuncts in the NCC, and finds the token corresponding to the remaining sequence of WMEs. (This can be done quickly if tokens are represented by linked lists, as we will see below; it can be more time-consuming if tokens are represented by arrays, though.) This token's local memory is then updated, and (if necessary) the NCC node's children are informed.

The astute reader may have noticed a possible efficiency problem in the above description. Whenever a new match for the earlier conditions was found, we first left-activated the NCC node, and propagated a new match to its children, possibly resulting in a great deal of activity in the network below. We then left-activate the top of the subnetwork and let any matches filter down to the NCC partner node. If any reach the NCC partner node, this indicates that the whole condition (the NCC) is false, so we then have to retract everything that just happened underneath the NCC node. This could result in a lot of wasted effort.

The source of the problem is that when the NCC node was activated, it did not "know" whether any results would emerge from the bottom of the subnetwork. It had to "play it safe" and assume that none would. (With the simple negative conditions of Section 2.7, we simply computed the join results for a new token "on demand," by scanning an alpha memory.) To avoid this possible inefficiency, we can order the preceding join node's *children* list so that the subnetwork gets activated *before* the NCC node — that way, when the NCC node gets activated, it can find out whether any results emerged from the bottom of the subnetwork just by looking over at its partner and checking whether any new results are waiting there.

Our pseudocode below assumes that this ordering is used.[15] Turning to the pseudocode now, an NCC node stores a list of tokens and a pointer to its partner node:

```
structure ncc-node
    items: list of token
    partner: rete-node {points to the corresponding NCC partner node}
end
```

[15] It is not hard to extend the code so it handles either ordering. The implementation used in this thesis does this. Note that our approach to handling conjunctive negations in Rete is not the only possible one. Though some earlier versions of Rete have supported conjunctive negations, this is the first implementation we are aware of that does *constrained* conjunctive negations — the matching activity in the subnetwork is constrained by the bindings of variables that occur outside the conjunctive negation.

We extend our *token* data structure to include two extra fields. *Ncc-results* will be the local result memory on a token in an NCC node. This is just like the *join-results* field on tokens in negative nodes, except that instead of each local result being specified by a *negative-join-result* structure, here each local result will be specified by a *token* structure. This is because the local results here are matches (tokens) emerging from the bottom of the subnetwork, rather than just WMEs (as was the case with negated single conditions). Since a token can now be a local result, we have to add an *owner* field to each token, just as we had an *owner* field in the *negative-join-result* structure (page 41).[16]

structure token {*revised from version on page 41*}
    parent: token {*points to the higher token, for items 1...i-1*}
    wme: WME {*gives item i*}
    node: rete-node {*points to the node this token is in*}
    children: **list** of token {*the ones with* `parent`*=this token*}
    join-results: **list** of negative-join-result {*used only on tokens in negative nodes*}
    ncc-results: **list** of token {*similar to* `join-results` *but for NCC nodes*}
    owner: token
        {*on tokens in NCC partners: token in whose local memory this result resides*}
end

An NCC partner node stores a pointer to the corresponding NCC node, plus a count of the number of conjuncts in the NCC (this is used for stripping off the last several WMEs from each subnetwork match, as discussed above). It also contains a *new-result-buffer*, which is used as a temporary buffer *in between* the time the subnetwork is activated with a new match for the preceding conditions and the time the NCC node is activated with that match. It stores the results (if there are any) from the subnetwork for that match.

structure ncc-partner-node
    ncc-node: rete-node {*points to the corresponding NCC node*}
    number-of-conjuncts: **integer** {*number of conjuncts in the NCC*}
    new-result-buffer: **list** of token
        {*results for the match the NCC node hasn't heard about*}
end

Our *ncc-node-left-activation* procedure is similar to the *negative-node-left-activation* procedure (page 42). In both cases, we need to find the join results for a new token. For negative nodes, we compute these join results by scanning the WMEs in an alpha memory and performing the join tests on them. For NCC nodes, the join results have already been computed by the subnetwork, so we simply look at the *new-result-buffer* in the NCC partner node to find them.

[16] Our *token* structure is getting rather big now; to save space, we can use a variant record structure for it, and allocate space for the last three fields, which are not used on ordinary tokens in beta memories, only when necessary.

```
procedure ncc-node-left-activation (node: ncc-node, t: token, w: WME)
    {build and store a new token, just like a beta memory would (page 30)}
    new-token ← make-token (node, t, w)
    insert new-token at the head of node.items

    {get initial ncc results}
    new-token.ncc-results ← nil
    for each result in node.partner.new-result-buffer do
        remove result from node.partner.new-result-buffer
        insert result at the head of new-token.ncc-results
        result.owner ← new-token

    {If no ncc results, then inform children}
    if new-token.ncc-results=nil then
        for each child in node.children do left-activation (child, new-token, nil )
end
```

To handle an NCC partner node (left) activation, we take the new match from the subnetwork and build a "result" token to store it. (The pseudocode for this is shown in Figure 2.8.) Next we *try* to find the appropriate owner token in the NCC node's memory. (There might be one there, if this is a new subconditions match for an *old* preceding-conditions match, or there might not be one there, if this is an initial subconditions match for a *new* preceding-conditions match.) If we find an appropriate owner token, then we add the new result token to its local memory; if the number of results in the local memory changes from zero to one — indicating that the NCC was previously true but is now false — then we call the *delete-descendents-of-token* helper function to delete any tokens lower in the network that depend on this owner token. (This is similar to the *negative-node-right-activation* procedure on page 43.) On the other hand, if there isn't an appropriate owner token already in the NCC node's memory, then this new result token is placed in the *new-result-buffer*. (The NCC node will soon be activated and collect any new results from the buffer.)

We now modify the *delete-token-and-descendents* procedure so it can correctly handle tokens in NCC and NCC partner nodes. (The modified pseudocode is shown in Figure 2.9.) Three things about such tokens require special handling. First, when a token in an NCC node is deleted, we need to clean up its local result memory by deleting all the result tokens in it. Second, when a token in an NCC partner node is deleted, instead of removing it from a *node.items* list, we remove it from its owner's *ncc-results* list. Third, when we do this, if the number of tokens on the owner's *ncc-results* list changes from one to zero — indicating that the NCC was previously false but is now true — then we inform any children of the NCC node.

Finally, we need to extend our *add-production* and *remove-production* handling (Section 2.6) so it can accept NCC's, creating and deleting NCC and NCC partner nodes. This is fairly straightforward and will not be presented here, but see Appendix A.

Note that (single) negative conditions are just a special case of conjunctive negations in which there is only one subcondition. They could therefore be implemented using the same mechanism as NCC's. However, this would take extra space and time, since it uses more nodes

```
procedure ncc-partner-node-left-activation (partner: rete-node, t:token, w:WME)
    ncc-node ← partner.ncc-node
    new-result ← make-token (partner, t, w) {build a result token ⟨t,w⟩}

    {To find the appropriate owner token (into whose local memory we should put this
    result), first figure out what pair ⟨owners-t,owners-w⟩ would represent the owner. To
    do this, we start with the ⟨t,w⟩ pair we just received, and walk up the right number
    of links to find the pair that emerged from the join node for the preceding condition.}
    owners-t ← t; owners-w ← w
    for i = 1 to partner.number-of-conjuncts do
        owners-w ← owners-t.wme; owners-t ← owners-t.parent

    {Look for this owner in the NCC node's memory. If we find it, add new-result to its
    local memory, and propagate (deletions) to the NCC node's children.}
    if there is already a token owner in ncc-node.items with parent=owners-t
            and wme=owners-w then
        add new-result to owner.ncc-results; new-result.owner ← owner
        delete-descendents-of-token (owner)
    else
        {We didn't find an appropriate owner token already in the NCC node's memory.
        This means the subnetwork has been activated for a new match for the preceding
        conditions, and this new result emerged from the bottom of the subnetwork, but
        the NCC node hasn't been activated for the new match yet. So, we just stuff the
        result in our temporary buffer.}
        insert new-result at the head of partner.new-result-buffer
end
```

Figure 2.8: Pseudocode for an NCC partner node left activation.

and node activations. Unless single negative conditions are quite rare, special-case handling of them is worthwhile.

2.9 Miscellaneous Implementation Notes

Having finished our discussion of the basics of Rete, we now make a few remarks about potentially important details or "tricks" of the implementation.

2.9.1 Garbage Collection

For a system to maintain good performance even with a hundred thousand or a million productions, we must avoid ever having to traverse the entire Rete network — such a traversal takes time directly proportional to the size of the knowledge base, so a system doing this would not

```
procedure delete-token-and-descendents (tok: token)
{revised from version on page 44}
    while tok.children ≠ nil do
        delete-token-and-descendents (the first item on tok.children)
    if tok.node is not an NCC partner node then
        remove tok from the list tok.node.items
    if tok.wme ≠ nil then remove tok from the list tok.wme.tokens
    remove tok from the list tok.parent.children
    if tok.node is a negative node then
        for each jr in tok.join-results do
            remove jr from the list jr.w.negative-join-results
            deallocate memory for jr
    if tok.node is an NCC node then
        for each result-tok in tok.ncc-results do
            remove result-tok from the list result-tok.wme.tokens
            remove result-tok from the list result-tok.parent.children
            deallocate memory for result-tok
    if tok.node is an NCC partner node then
        remove tok from the list tok.owner.ncc-results
        if tok.owner.ncc-results = nil then
            for each child in tok.node.ncc-node.children do
                left-activation (child, tok.owner, nil )
    deallocate memory for tok
end
```

Figure 2.9: Revised pseudocode for deleting a token and its descendents.

be very scalable. Unfortunately, traversing the entire Rete network is precisely what standard "stop-and-copy" garbage collectors do. This results in garbage collection pauses whose duration grows as the knowledge base grows, rendering the system unsuitable for any real-time tasks. To avoid this, we must either use a more sophisticated garbage collector, or handle memory allocation and deallocation within the Rete net ourselves.[17] The latter approach is used in the pseudocode given in this chapter, and is quite simple and straightforward, since we always know when things become garbage: tokens become garbage when they are deleted during a WME removal, as described in Section 2.5, and Rete nodes become garbage during a production removal, as described in Section 2.6.

[17]For efficiency, a small number of available space lists (Knuth, 1973a), each containing blocks of a fixed size (for a particular data structure), should be used. This is much faster than calling `malloc()` and `free()` (Miranker and Lofaso, 1991).

2.9.2 Merging Beta Memories and Join Nodes

It turns out that in typical systems, most beta memories have just a single child join node. An implementation of Rete may merge such a pair of nodes into one node. This saves space because it eliminates an extra set of pointers — *parent*, *child*, *next*, *previous*, etc. It also saves time, since now a single procedure call (*merged-node-left-activation*) replaces a sequence of two (*beta-memory-left-activation* and *join-node-left-activation*).

Note that if a memory node has $k > 1$ child join nodes, it is still possible to used merged nodes: k copies of the memory node may be used, one merged with each join node. This has been done in at least one implementation (for independent reasons) (Gupta et al., 1988). However, this can increase both the space and time usage. The space increases because for every token we previously had in the un-merged version, we now have k copies of it. The time usage may decrease, since what used to take $k+1$ procedure calls now takes only k, but may also increase, because of the extra time required to create $k-1$ extra tokens.

We thus want to use a merged node only when the memory has just one child. Of course, the number of children a given beta memory has can vary as productions are added to or removed from the Rete net. Therefore, we sometimes dynamically split a merged node into its two separate components as we are adding a new production to the net, or merge the two components into a single node as we are removing a production from the net. (To our knowledge, the Rete implementation used in this thesis is the first to do this.)

Another possibility is to merge nodes slightly differently: if a join node has just a single beta memory node as its only child, we can merge the two nodes. A similar analysis applies to this case.

2.9.3 Speeding Up the Addition of Productions

When we add new productions to the network, we always look for existing nodes to share before we build any new nodes. This search for appropriate existing nodes takes the form of a loop over all the existing children of a certain node (e.g., see the *build-or-share-join-node* procedure on page 34). In large learning systems, this loop may become problematic: there is no limit to how many children a given node can have, and as a system learns more and more productions, some Rete nodes may acquire more and more children. If this happens, we will incur more and more iterations of this loop, and it will take longer and longer to add new productions to the network.

To avoid this problem, we can use an auxiliary hash table which lets us quickly find any children of a given (parent) node that happen to have certain properties (e.g., being a node of a certain type, or using a certain alpha memory). Essentially, the hash function takes as input (a pointer to) the given parent node and a list of the properties of the child node. As output, we get a bucket in the hash table containing all existing children of the given parent node having exactly those properties. By using an auxiliary hash table this way, we can replace the potentially expensive loop over all the children with a simple hash table lookup.

2.9.4 Resizing Hash Tables

Of course, the discussion in the previous section assumes that the cost of the hash table lookup does not increase as the number of nodes in the network increases. But if we use a fixed-size hash table, then we will have more and more hash collisions as the number of nodes (and hence the number of entries in the hash table) increases. This would increase the cost of hash table lookups, defeating the whole purpose of using a hash table in the first place. A similar argument applies to any hash tables used as part of the alpha network, as described in Sections 2.2.2 and 2.2.3 Thus, these hash tables must be dynamically resized as items are added to them during the addition of a new production to the system (e.g., doubling the number of buckets each time the number of items doubles). Most versions of Lisp take care of this automatically, but in other languages, we may have to handle it ourselves.

2.10 Other Optimizations for Rete

In this section, we discuss some other ways that have been developed to improve the Rete algorithm. Three substantially different match algorithms have been developed as alternatives to Rete: Treat (Miranker, 1990), Match Box (Perlin and Debaud, 1989; Perlin, 1991a), and Tree (Bouaud, 1993). We postpone our discussion of these until after we have looked at some aspects of Rete's performance on our large testbed systems, so we can discuss how their performance would compare. Treat will be discussed in Sections 3.6, 5.8.4, and 5.8.5. Match Box will be discussed in Section 3.6. Tree will be discussed in Section 5.8.3.

The largest body of work aimed at speeding up production matching has dealt with parallel implementations of match algorithms, most notably (Gupta, 1987) and (Stolfo and Miranker, 1986); for a good survey of this work, see (Kuo and Moldovan, 1992). Of course, these are just parallel versions of existing sequential algorithms, so although they achieve faster overall speed, they do so only with special hardware and are of no use on ordinary sequential machines.

A number of other miscellaneous (potential) improvements to Rete have been developed over the years. We mention these briefly here, not because they are directly related to this thesis, but in order to provide the interested reader with an overview and some pointers to the literature.

- *Modify-in-place* (Schor et al., 1986) extends the Rete interpreter procedures to handle the *modification* of a WME directly. Our pseudocode requires a modification to be handled indirectly, via a removal of the original WME followed by the addition of the modified WME.

- *Scaffolding* (Perlin, 1990a) is useful when the same WMEs are repeatedly added to and removed from working memory. It works by marking WMEs and tokens as "inactive" instead of deleting them; when a WME is re-added later, inactive tokens are re-activated using a procedure similar to our tree-based removal method.

- Another way to implement the alpha network is to use a *decision tree* (Ghallab, 1981; Nishiyama, 1991). This works somewhat like the "dataflow-network-with-hashing" version we discussed in Section 2.2.2, but yields a nice time bound: it is guaranteed to run in time

linear in the depth of the decision tree. However, it makes run-time addition and deletion of productions more complicated, and the decision tree can require exponential space in the worst case.

- In systems using some form of *conflict resolution* (McDermott and Forgy, 1978), the matcher can often be sped up by incorporating knowledge about the particular conflict resolution strategy (*test*) into the matcher (*generator*). A conflict resolution strategy is a way of selecting a single complete production match and using it alone, instead of using all complete matches. The optimization idea is to change the definition of the matcher module so its job is to find just the one complete match that would get selected by the conflict resolution strategy, instead of finding *all* complete matches. This has been implemented in *lazy match* (Miranker et al., 1990) and also by (Tzvieli and Cunningham, 1989). A related idea is to incorporate Prolog-like *cuts* in the LHS, allowing only one token to propagate past a cut (Lichtman and Chester, 1989).

- In systems with very large working memories, the cost of individual JOIN operations can become expensive, due to large cross-products being generated. *Collection Rete* (Acharya and Tambe, 1992) is a way of reducing this cost by structuring the contents of beta memories into a set of *collections of tokens* rather than a set of *individual tokens*.

- Another way to reduce the cost of JOIN operations in individual productions is to add *factored arc consistency* (Perlin, 1992) to Rete. This adds an arc-consistency algorithm to Rete in order to "prune off" many WME/token combinations without checking them one by one. Although its overhead is so high that it often does not pay off in practice, it can help if used judiciously (e.g., only on particularly expensive productions).

- Still another way to avoid expensive JOIN operations is to restrict the contents of working memory by using *unique-attributes* (Tambe et al., 1990). The Rete algorithm can then be specialized to take advantage of certain properties of this restriction, yielding the *Uni-Rete* algorithm (Tambe et al., 1992). Uni-Rete is significantly faster than Rete, but is not as general-purpose, since it requires strict adherence to the unique-attributes restriction.

- In standard Rete, the set of beta network nodes for a production — the beta memories, join nodes, etc. for it — is linear (conjunctive negations notwithstanding). Researchers have investigated using *nonlinear topologies* for the beta network for individual productions (Schor et al., 1986; Scales, 1986; Gupta, 1987; Ishida, 1988; Tambe et al., 1991), with mixed results. If we can find the best topology for a production, we can often do better than the standard linear one. Unfortunately, there is no known efficient algorithm for finding the best one, and some nonlinear topologies turn out to be much worse than the standard linear one. Techniques for efficiently finding good topologies remain an area of current research (Hanson, 1993).

2.11 Discussion

Having finished our discussion of the implementation of the basic Rete algorithm, we now make a few remarks about its generality. Rete's range of applicability is sometimes underestimated, perhaps because its best-known uses — in production systems similar to OPS5 — all share certain features not really required by Rete. As mentioned at the beginning of this chapter, match algorithms like Rete can be treated as "black boxes" — modules performing a service as part of larger overall systems. Naturally, the properties and limitations of the overall systems are not necessarily imposed on them by the match algorithm, but may instead arise from other considerations.

For instance, currently most systems using Rete are production systems that employ some method of *conflict resolution*. In each cycle, Rete is used to find the set of complete production matches (the *conflict set*). A conflict resolution method is then used to select a single production to fire; its right-hand-side is executed, modifying the working memory, and then the cycle repeats. Conflict resolution has nothing to do with Rete. Indeed, Soar uses Rete but does not employ any conflict resolution at all: at each cycle, all matching productions are fired.

A second feature of many systems currently using Rete is that WMEs are represented using attributes and values. In this chapter we have been using WMEs of the form

```
(identifier ^attribute value).
```

However, the Rete implementation described in this chapter can easily be modified to support other representations. OPS5 and related systems also use attribute-value syntax, but allow a single WME to have more than one attribute-value pair:

```
(classname ^attr-1 value-1 ^attr-2 value-2 ... ^attr-n ^value-n)
```

Many machine learning systems use a simple predicate-with-arguments representation:

```
(predicate-name arg-1 ... arg-n).
```

Some implementations allow WMEs to be arbitrary list structures with nested sublists (Forgy, 1979; Allen, 1982), to contain records and arrays (Forgy, 1984), or to be arbitrary data structures from a high-level programming language (Cruise et al., 1987; Miranker et al., 1991).

Modifying Rete to support these other representations simply involves changing the way tests (constant tests and variable binding consistency tests) are specified in data structures and performed by node activation procedures. For example, if WMEs are arbitrary list structures, then a variable binding consistency test might be something like "the third element of the first WME must equal the second element of the first element (a nested sublist) of the second WME."

Of course, these other representations yield no additional representational power beyond that of id-attribute-value triples, since WMEs and conditions in the other representations can be mechanically transformed into equivalent (sets of) WMEs and conditions in id-attribute-value form. For example, the WME

```
(classname ^attr-1 value-1 ^attr-2 value-2 ... ^attr-n ^value-n)
```

can be represented by an equivalent set of id-attribute-value triples all having the same id:

```
(id-337 ^class classname)
(id-337 ^attr-1 value-1)
(id-337 ^attr-2 value-2)
...
(id-337 ^attr-n value-n)
```

(The choice of `id-337` is arbitrary: we would use a different "gensym" identifier for each original WME we transform in this way.) Similarly, the predicate-with-arguments representation can be transformed into

```
(id-294 ^predicate predicate-name)
(id-294 ^first-arg arg-1)
...
(id-294 ^nth-arg arg-n)
```

To transform arbitrary list structures into equivalent sets of id-attribute-value triples, we use a different identifier for each cons cell in the list structure, and then use `car` and `cdr` attributes to specify the structure.

Although the choice of notation for WMEs does not affect the representational power of the system, it may affect the performance of certain parts of the Rete algorithm. In general, the above transformations increase the number of WMEs needed to represent a given thing, and the number of conditions needed to test for it. After we apply these transformations, there will often be many conditions in a production with the same variable in their id field; thus, many variable binding consistency tests will be required with the id-attribute-value representation that would not be required in the more complex representations. This means the beta part of the network will have to do more work. On the other hand, the alpha part of the network may have to do more work with the more complex representations than with the simple id-attribute-value representation. In the simple representation, a given WME can go into at most eight alpha memories, as discussed in Section 2.2.3 (assuming all intra-condition tests are for equality with constants). In the more complex representations, a WME may go into many more — e.g., a predicate with r arguments could go into up to 2^{r+1} alpha memories — so the alpha network may no longer run as fast as it can with the id-attribute-value representation. Essentially, transforming the representation can shift work from one part of the network to another. Performing a careful study of the relative performance of the matcher using each of these different representations would require obtaining a collection of testbed systems, each encoded using each of several different representations; that is not the focus of this thesis, but is an interesting idea for future work.

Finally, although the basic Rete algorithm delivers only *complete matches* for productions, it can be extended to deliver the best (or near-best) *partial match* under some suitable metric for rating the quality of partial matches. A partial match is a set of WMEs which satisfy some but not all of the conditions of a production. Finding good partial matches is a requirement for most case-based reasoning systems, and although some systems have their own special-purpose

algorithms for this (Veloso, 1992), no fully general algorithm has been developed. To assess the quality of a partial match, we might assign (in advance) a numeric *penalty* to each condition; the quality of a given partial match would then be inversely related to the sum of the penalties for all the unsatisfied conditions in the production.

Within the Rete algorithm, a partial match could be represented by a token with certain elements set to *unmatched*. (Recall that a token represents a sequence of WMEs; we could just replace some of them with this special flag.) Each token would have an additional field, *accumulated-penalty*, indicating the total penalty incurred for all its *unmatched* entries. Join node activation procedures would be modified to create partial matches: on a left activation, given a new token t, a join node would signal a new match $\langle t, \mathit{unmatched} \rangle$ with an appropriately increased *accumulated-penalty*.

At this point, the algorithm would be virtually useless, since it would deliver (to the p-nodes at the bottom of the network) *all* partial matches, even $\langle \mathit{unmatched}, \ldots, \mathit{unmatched} \rangle$. To get only the partial matches of at least a certain quality, we could set a penalty threshold τ — the algorithm would be modified to generate only tokens with *accumulated-penalty* $\leq \tau$. For this threshold to be most effective, we would want the conditions in each production to be ordered so that those with high penalties appear near the top of the Rete network, and those with low penalties appear near the bottom. With this ordering, the threshold would cut off matching activity as early as possible (without changing the set of matches reaching the p-nodes — the ordering has no effect on this).

Even better, we can obtain just the partial match with the lowest possible penalty by modifying the order in which tokens are generated and node activations take place. In the standard Rete algorithm, the dataflow takes place *depth-first*: before the *second* child of a given node is activated, the *first* child of the node is activated, and all processing resulting from that activation is completed. Instead, we would like to perform processing of tokens in order of increasing penalty. So whenever a token is created, instead of passing it on to the node's children right away, we could store it in a priority queue sorted according to *accumulated-penalty*. The basic loop of the algorithm would be to remove from the queue the token with the smallest penalty, pass it on to the appropriate nodes, and add any resulting tokens from those nodes to the queue. A priority queue can be implemented with logarithmic cost, so the extra cost of processing tokens in this order is, at worst, a factor logarithmic in the total number of tokens generated.

The ideas above are just a sketch and are still untested; to our knowledge, this kind of partial matching has never been implemented using Rete.[18] Developing these ideas further and implementing them in a real system is beyond the scope of this thesis. However, the above discussion suggests that some variant of Rete might be useful in case-based reasoning systems

[18] At least one implementation has supported special "don't-care" symbols in WMEs (Rosenbloom, 1994). These symbols "match" any constant in any condition. This provides a different form of partial matching. (Perlin, 1991b, pages 78-79) proposes a partial matching scheme in which Rete generates matches that may fail up to some fixed number of constraints; this scheme is a finer-grained version of our use of *unmatched*, but generates many extra matches besides the best one. ACORN (Hayes-Roth and Mostow, 1975), a predecessor of Rete, implemented a limited form of partial matching using "fuzzy" equality tests, where "equality" was defined using a fixed threshold parameter (Mostow, 1994); this approach also yields many extra matches besides the best one.

or other systems which need to retrieve close partial matches. This remains an interesting area for future work.

Chapter 3

Basic Rete: Performance and Observations

Having described the basic Rete algorithm in the previous chapter, we now examine its performance on large learning systems. We first describe in Section 3.1 the various learning systems we will use as testbeds in this thesis. In Section 3.2, we discuss some issues related to the generators used to create training problems for these systems. We then present empirical results on the performance of the basic Rete algorithm on these testbed systems in Section 3.3 — in particular, we show that it slows down linearly in the number of rules in each system. In Section 3.4, we examine the fundamental cause of this slowdown, namely, the large number of productions affected by changes to working memory. Rete's sharing, one way to alleviate some of the effects of this fundamental cause, is examined in Section 3.5. Finally, Section 3.6 discusses the implications of our observations for other work on production match algorithms.

3.1 The Testbed Systems

Several large learning systems are used as testbeds for the empirical studies in this thesis. None of these systems was designed especially for match algorithm performance or for learning a large number of rules; in fact, many of them were designed before the author embarked on this thesis research. In this section we describe each system briefly, to provide the reader with a feel for the complexity and diversity of the systems. The interested reader should refer to the literature citations below for more information about any particular one of these systems. Sections 3.1.1 through 3.1.6 describe each of the testbed systems.

Each of the testbed systems is implemented in Soar (Laird et al., 1987; Rosenbloom et al., 1991), an integrated problem-solving and learning architecture based on formulating every task using *problem spaces*. Each basic step in Soar's problem-solving — e.g., the immediate application of an operator to a state to generate a new state — is called a *decision cycle*. The knowledge necessary to execute a decision cycle is obtained from Soar's knowledge base, which is implemented as a production system whose working memory represents problem spaces, states, and operators. If this knowledge is insufficient to reach a decision, an *impasse* occurs; Soar then uses

another problem space in a subgoal to obtain more knowledge. This process can recurse, leading to a stack of problem spaces. Soar learns by compiling the results of this subgoal processing into *chunks*, productions that immediately produce comparable results under analogous conditions (Laird et al., 1986). Chunking is a form of explanation-based learning (DeJong and Mooney, 1986; Mitchell et al., 1986; Rosenbloom and Laird, 1986).

The number of productions in a Soar system can be used as a crude measure of its complexity, somewhat like the number of lines of code in a program written in a conventional programming language. Perhaps a better measure of the complexity of a Soar system is the number of problem spaces it uses. This is roughly comparable to the number of procedures or modules in a conventional program. Encodings of typical AI "toy tasks" in Soar usually involve just two problem spaces.

Each testbed system starts out with a small to moderate number of rules. These initial (unlearned) rules are normally just hand-coded Soar productions. In three of the testbed systems, however, most of the initial rules were generated from source code written in TAQL (Yost and Newell, 1989; Yost, 1992), a higher-level language which compiles into Soar productions. The use of TAQL does affect the rules these systems learn — they often have extra conditions and actions that would not be present if the initial rules were hand-coded instead — but these extra conditions and actions do not appear to have a significant effect on the matcher.

For each testbed system, a problem generator was used to create a set of problems in that system's task domain; the system was then allowed to solve the sequence of problems, learning new rules as it went along. Each system learned at least 100,000 rules. (This was done using Rete/UL, the improved match algorithm described later in this thesis. As we will see in Section 3.3, with the basic Rete match algorithm, these testbed systems all become very slow as they learn more and more rules, making it infeasible to have many of them solve enough problems to learn so many rules in a reasonable amount of time.)

Table 3.1 summarizes the testbed systems. For each, it indicates the system's author, the task domain in which it functions, and the main problem-solving techniques it employs. Table 3.2 summarizes the systems' sizes, showing the number of problem spaces, initial rules, and learned rules for each one. These seven systems provide a good test suite because:

- They operate in a variety of task domains.
- They use a variety of problem-solving methods.
- They were written by a variety of people.
- They were written with different research interests in mind.
- They are at least moderately complex, not standard AI "toy problems."
- They were not designed especially for this research.

System	Author	Domain	Problem-solving technique(s)
Assembler	Mertz	circuit board assembly	procedural, recognition
Dispatcher	Doorenbos	message dispatching	search, procedural, memorization
Merle	Steier et al.	factory scheduling	procedural, generate & test
Radar	Papageorgiou	aircraft classification	episodic memory
SCA-Fixed	Miller	general concept acquis.	deliberate rule induction
SCA-Random	Miller	general concept acquis.	deliberate rule induction
Sched	Nerb & Krems	job-shop scheduling	advice-taking

Table 3.1: Domains and basic problem-solving techniques of the testbed systems.

System	Problem spaces	Initial rules	Learned rules
Assembler	6	293	105,015
Dispatcher	20	1,953	113,938
Merle	18	624	105,699
Radar	7	341	105,207
SCA-Fixed	1	48	154,251
SCA-Random	1	48	119,902
Sched	4	418	116,968

Table 3.2: Sizes of the testbed systems.

3.1.1 Assembler

Assembler (Mertz, 1992) is a cognitive model of a person assembling printed circuit boards. It was developed as part of a research project aimed at improving instructional design for training people to perform this task (Mertz, 1995). The system is presented with a printed circuit board with various labeled slots for electronic components; several bins, each containing electronic parts of a certain type (e.g., ten ohm resistors); and an instruction list indicating what type of part belongs in each slot. Assembler's approach to the task is basically procedural: it iterates over parts, slots, and bins, picking up each part from the appropriate bin and inserting it into the appropriate slot.

The first time it assembles a given kind of board, the system spends a great deal of time iterating over various slots on the board, trying to find the appropriate slot for a certain part. (This may not be the most efficient possible approach to the task, but Assembler is intended as a cognitive model, not an optimal-performance system.) It learns several types of rules; the most important rules are ones that speed up the assembly process on later trials by allowing it to recognize the appropriate slot quickly. These rules are specific to particular slots on particular boards, so to be able to assemble several different kinds of boards proficiently, the system needs a large number of rules.

Assembler uses six problem spaces and begins with 293 (unlearned) rules. Some of these rules were hand-coded; others were generated from source code written in TAQL. A problem generator was used to create 300 different printed circuit boards, each with 50 slots. Both the arrangement of slots on a board and the assignment of particular parts to particular slots are

arbitrary and have no effect on the system's problem solving. No two boards have a slot in common; i.e., the symbols used to designate particular slots are different on different boards. Over the course of assembling each of these boards once, the system learned 105,015 rules.

3.1.2 Dispatcher

Dispatcher (Doorenbos et al., 1992) grew out of a research project aimed at studying how AI systems could use databases in service of AI tasks. The system acts as a message dispatcher for a large organization. It makes queries to an external database containing information about the people in the organization, their properties, and their ability to intercommunicate. Given a specification of the desired set of message recipients (e.g., "everyone involved in marketing for project two"), the system must find a way to get the message to the right people. This problem is different from a simple network routing problem because both communication links and desired recipients are specified in terms of *properties* of people — for example, a communication link might be specified by "John can talk to all the marketing people at headquarters." Also, the data is available only indirectly in a database, which is external to the system rather than part of it. Whenever Dispatcher needs some piece of information from the database, it must formulate a query using the SQL database-query language, send the query off to a database system, and interpret the database system's response.

Dispatcher has three basic methods of performing its task. First, it can use the database to try to find someone in the organization who can talk to precisely the desired set of message recipients, and send the message to that person with instructions to forward it to all the desired recipients. Second, it can use the database to obtain a list of all the desired recipients, then send a separate copy of the message to each one. Third, it can use a divide-and-conquer approach, breaking the desired set of recipients down according to geographic region (or some other property), then recursively use any of the three basic methods on each subset. The system chooses among these three methods by using best-first search with a heuristic evaluation function. (The solutions Dispatcher finds are not necessarily optimal.) With any of these methods, the system eventually reaches a point where it needs to find a way to send a message to some particular person (say Fred). Dispatcher is only able to *directly* communicate with a few people in the organization. If it is unable to talk directly to Fred, it uses breadth-first search to find a sequence of intermediaries who can forward the message to Fred. The system makes queries to the database to find people who can talk to Fred, then more queries to find people who can talk to each of those people, and so on, until the system reaches someone it can talk to directly.

There are three main types of rules Dispatcher learns. First, it learns search-control rules that are used to choose among the three basic methods, avoiding the need to invoke the heuristic evaluation function to compare alternatives. Second, it learns rules which encapsulate the solutions to various subproblems — e.g., constructing the proper SQL query to retrieve a certain type of information from the database, or finding a sequence of intermediaries who can forward a message to a particular person. Third, it learns rules which contain memorized database query results. Whenever Dispatcher has to go out to the database to find some piece of information, it learns a rule (based on the memorization technique introduced in (Rosenbloom and Aasman,

1990)) that "memorizes" it. The rule can transfer to other problems, so that if the same information is needed again, the database will not have to be consulted a second time. Although the system does not manage to memorize the entire database, it sometimes "gets lucky" and solves a problem without consulting the database at all.

Dispatcher is implemented using twenty problem spaces, eight involved with the dispatching task and twelve involved with using the external database. It starts with 1,953 initial productions, most of which are compiled from source code written in TAQL. The database it uses describes an organization of 61 people; the details of this organization were created by a random generator, weighted to make the organization modestly realistic, e.g., people in the same region are more likely to have a communication link between them than people in different regions. A random problem generator was used to create 6,550 different dispatching problems for the system. The generator used weighted probabilities to skew the problem distribution in favor of modestly realistic problems, but the skew was mild enough that many less realistic problems were also included. Problems were selected from this distribution without replacement, so that the system never saw exactly the same problem twice. (This also reduces the effect of the skewed problem distribution.) Over the course of solving these 6,550 problems, the system learned 113,938 rules.

3.1.3 Merle

Merle (Prietula et al., 1993) is an expert system for scheduling tasks for an automobile windshield factory. It has been used to study learning in the context of a scheduling domain. The system is given a windshield order, specified by several parameters indicating the particular types of windshields to be produced and the desired quantity of each type, and the amount of time available on each machine in the factory. It then iterates over all the available slots in the schedule; for each one, it uses generate-and-test to try to find a task which can fill that slot. A task must satisfy various constraints in order to be scheduled in a given time slot.

The rules learned by Merle encapsulate solutions to various subproblems of the overall scheduling problem — e.g., checking constraints, determining the options available for filling a single schedule slot, or doing some simple bookkeeping operations. These rules transfer to other problems (or, as often happens in Merle, to later occasions within the same problem during which they were learned) where the same or similar subproblem arises again.

Merle uses eighteen problem spaces, and starts out with 624 hand-coded rules. Over the course of solving 160 different problems, created by a random problem generator, it learned 105,699 rules. Each problem is specified by thirty-six parameters; these are chosen independently from uniform (or, in a few cases, mildly skewed) distributions, except for six parameters whose values are restricted by those of other parameters (e.g., "start time" must be before "stop time").

3.1.4 Radar

Radar (Papageorgiou and Carley, 1993) is a cognitive model developed as part of a research project on procedures for training decision-making agents. The system examines descriptions of radar images of planes and learns to classify them as either friendly, neutral, or hostile. The

images are specified in terms of nine simple features, such as speed, altitude, and size; each feature has nine possible values. The system learns to correctly classify images by building up an episodic memory of previously seen images and their true classifications. The memory is indexed using a representation more abstract than the nine-feature one, so that generalization of individual training examples can occur. When presented with an example radar image, the system first maps the nine feature values into the abstract representation, then uses the most recent similar episode to classify the image. "Similar" is defined as "having the same abstract representation." If no similar episode is present, the system guesses randomly. The system is then given feedback, and it updates its episodic memory.

In this system, the purpose of the learning is not to speed up the system, but to increase its classification accuracy. Part of the system's episodic memory is implemented by learned rules. (Rules store the actual episodes, while the index of episodes is stored in a table implemented in some code which is not part of the Soar architecture.) As the system sees more and more training examples, the coverage of its episodic memory increases, and consequently, so does its classification accuracy. There is no noise in the training data or feedback.

Radar uses seven problem spaces and starts with 341 initial rules, some hand-coded, some compiled from source code written in TAQL. It learned 105,207 rules over the course of 7,500 randomly-generated training examples. The examples were selected, with replacement, from a uniform distribution over the 3^9 possible image specifications. A mathematical formula was used to determine the correct classification of each training example.

3.1.5 SCA-Fixed and SCA-Random

SCA (Symbolic Concept Acquisition) (Miller and Laird, 1991; Miller, 1993) is a cognitive model of traditional concept acquisition (i.e., in the style of ID3 (Quinlan, 1986)). While Radar is hard-coded for a particular domain, SCA is domain-independent. It performs incremental, noise-tolerant learning, and reproduces a number of regularities in human concept acquisition found in the psychological literature.

As in most concept acquisition systems, a training example for SCA consists of a list specifying the values of a number of features, plus an indication of the true classification of the example (sometimes subject to noise). From such an example, SCA normally learns a single classification rule. The rule's conditions include a subset of the given feature values, and its actions include the given true classification. For rules learned early on, the conditions include only a small subset of the feature values, so these rules yield significant generalization of the training examples. As the system sees more and more training examples, the rules' conditions include more and more features, so the rules become increasingly specific. To make a prediction (at test-time), SCA tries to apply more specific rules first, so that specific rules "override" more general ones.

Although concept acquisition can be a complex task, SCA's algorithm is quite straightforward to implement. SCA starts out with forty-eight hand-coded rules, and uses a single problem space. Two versions of SCA were used. SCA-Fixed always focuses its attention on the same subset of the features of each training example, whereas SCA-Random focuses on a different randomly

chosen subset on each training example.[1] SCA-Fixed models situations where the system has prior knowledge of the relative importance of different features, while SCA-Random models situations where the system has no such knowledge. SCA-Random learned 119,902 rules from 120,000 training examples. SCA-Fixed learned 154,251 rules from 200,000 training examples.[2]

The training examples were created by a random generator. Each example is specified by values for twelve features; each feature takes one of twelve possible values. The system must learn to classify each training example into one of twelve classes. The generator works as follows. For each class, for each feature, it creates a probability distribution over the twelve possible values of that feature. This distribution is chosen randomly but weighted so certain values have a high probability and others have a very low probability. Once these distributions have been created for all classes and features, the generator creates successive training examples by selecting a class (one of the twelve possible classes is chosen with equal probability) and then, for each of the twelve features, selecting a value for that feature using the previously created probability distribution for that class and feature.

3.1.6 Sched

Sched (Nerb et al., 1993) is a computational model of skill acquisition in job-shop scheduling. Its domain is a simple job shop with two machines. The system is given a list of jobs to be done. For a given job to be completed, that job must first occupy machine one for a certain amount of time (specified in the list of jobs) and then occupy machine two for some other amount of time. Each machine can process only one job at a time. The task is to schedule the jobs so as to minimize the total time to completion. The research focus here is not on obtaining an optimal schedule — there is a straightforward algorithm for that. Rather, the focus is on modeling the way human subjects acquire skill while learning to perform this task. (Subjects do not quickly acquire the aforementioned algorithm.)

The system performs the task by selecting jobs to schedule first, second, and so on. Whenever it is unsure which job to schedule next, it asks for advice from a human supervisor. It then reflects on why this advice applies to the current situation (here the system makes an inductive leap) and memorizes those aspects of the situation deemed relevant, e.g., amounts of time required by different jobs, or positions of jobs in the schedule. The system then recalls this memorized knowledge in similar situations later. (This can lead to positive or negative transfer, depending on whether the inductive leap was correct.) Most of the rules Sched learns are rather specific, because they incorporate particular numeric values in their conditions.

Sched uses four problem spaces and starts with 418 rules. Over the course of solving 3,000 different problems created by a random problem generator, it learned 116,968 rules. Each

[1] Strictly speaking, the subset depends on the *order* in which individual features are *selected* to be the focus of attention, and on the *number* of features which receive attention. In both SCA-Fixed and SCA-Random, the number gradually increases over time — learned rules grow increasingly specific. But in SCA-Fixed, the order of selection of features is held fixed across all training examples, whereas in SCA-Random, a different random order of selection is used on each training example.

[2] To demonstrate its feasibility, we also gave SCA-Fixed additional training examples until it learned over one million rules. With Rete/UL, the system showed no significant increase in match cost.

problem is specified by ten integer parameters in the range 100–999; these parameters were sampled independently from uniform distributions.

3.2 Problem Generators

For each testbed system, a random problem generator was used to create enough problems for the system to solve so that it learned at least 100,000 rules. In each case, the problems were selected from some probability distribution over a space of possible problems. Although a thorough study of the effects of different problem distributions on the match cost in these systems is beyond the scope of this thesis, we give a brief analysis here. For our purposes, the key question is not whether a different problem distribution would affect the results we obtain later in this thesis *at all* — it would almost certainly alter our quantitative results by at least some constant factor — but whether a different problem distribution would yield *qualitatively* different results.

In general, the problem distributions we use are fairly uniform — the skew, if any, is fairly mild. The problem distributions a system would encounter in "real life" situations are often more skewed: the same problems or very similar problems are often encountered repeatedly, while "outliers" show up only occasionally. What effect would such a highly skewed problem distribution have in these systems?

The problem distribution each of our testbed systems encounters affects both the set of rules it learns and its typical problem-solving behavior, which in turn affects the distribution of WMEs in working memory. These two things — the rules and the distribution of WMEs — together determine the match cost (given any particular match algorithm). Consider first the set of rules. In the long run, a system facing a problem distribution skewed toward some class of problems will encounter a subset of the problems encountered by a system facing a uniform problem distribution — with a uniform distribution, we eventually see everything in the skew class, plus we see some other problems outside that class. (How long "the long run" is in each of our testbed systems is an open question; the following analysis should hold to whatever extent running each system out to 100,000 rules yields a good approximation of "the long run.") Thus, the rules learned under a skewed distribution are a subset of those learned under a uniform distribution. This implies that for any given distribution of WMEs in working memory, the match cost with the rules from a skewed distribution of problems will be no higher than, and may be lower than, the match cost with the rules from a uniform distribution of problems.

To look at this another way, a highly skewed distribution would tend to lower the rate at which the systems learn rules. In most Soar systems, if the same problem (or subproblem) is encountered more than once, a new rule is only learned during the first encounter; on subsequent encounters, that rule is used to solve the problem (or subproblem) directly, and no additional rule is learned. The more skewed a problem distribution is, the more often identical problems or subproblems will arise, and hence the less often new rules will be learned. Later in this thesis, we develop improved matching techniques which allow most of these testbed systems to run for hours or days, learning 100,000 or more rules, without the match cost increasing significantly. If our problem distributions were more skewed, the systems would have to encounter more

training problems before they learned 100,000 rules. Thus, the techniques we develop to allow these systems to run for hours or days might allow systems facing "real life" distributions to run for weeks or months.

Now consider the second effect of a skewed problem distribution — the effect on the distribution of WMEs in working memory. This effect is harder to predict than the effect on the set of rules learned. Although we have been unable to conceive of a reason the WME distribution resulting from some skewed problem distribution would result in qualitatively different match behavior in these testbed systems, we cannot completely rule out this possibility. A thorough study would involve examining each testbed system and looking for particular problems which yield an abnormally high match cost,[3] finding characterizations of classes of such problems, and studying their effects on the matcher. However, such a detailed study is beyond the scope of this thesis.

In lieu of such a study, though, we note that if there are classes of problems whose WME distribution yields an abnormally high match cost, they would probably be reflected to some degree in the results we obtain using relatively uniform problem distributions. If enough problems are selected from a uniform distribution, there is a fairly high probability that at least one selected problem lies in one of the abnormal classes. Its abnormal effects would then show up in the results we obtain from a uniform distribution, although the magnitude of those effects would be reduced by a factor depending on the relative sizes of the skew class and the whole space of possible problems. For example, if there is a skewed problem distribution that would lead to a very large, linearly increasing match cost in one of the testbed systems, then with a uniform problem distribution, this would probably be reflected as a small, linearly increasing match cost.

In summary, the relatively uniform problem distributions we use in this thesis affect both the rules each system learns and the distribution of WMEs appearing in working memory. As far as the rules are concerned, the uniform distributions are likely to exacerbate the problem of increasing match cost, so these distributions provide a difficult test for match algorithms: matching techniques which work well here may work even better with other distributions. As far as the WME distributions are concerned, the effect of uniform problem distributions is difficult to predict, but it is unlikely that our use of them causes us to overlook any major effects on match cost.

3.3 Performance of Basic Rete on the Testbed Systems

Figure 3.1 shows the performance of the basic Rete algorithm on the testbed systems. For each system, it plots the average match cost per change to working memory as a function of the number of rules in the system.[4] It clearly demonstrates that as more and more rules are learned

[3] Or an abnormally low one, although such "good luck" is not really something to be concerned about.

[4] The times reported here and elsewhere in this thesis are for Soar version 6.0.6 (except for changes to the matcher), implemented in C, running on a DECstation 5000/260. Some previous studies of match algorithms have measured algorithm performance by counting the number of tokens they create (Tambe et al., 1990; Doorenbos et al., 1992) or the number of comparisons they require to perform variable binding consistency checks (Miranker, 1990; Bouaud, 1993). Although these metrics avoid the implementation-dependent nature of CPU

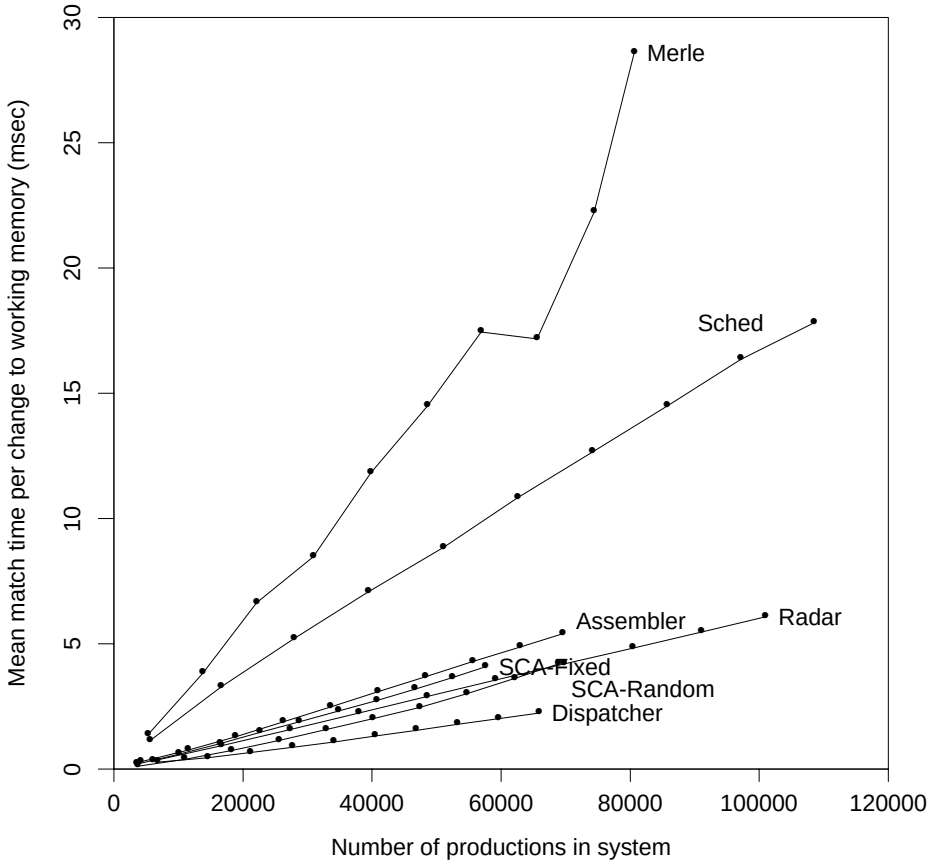


Figure 3.1: Match cost with the basic Rete algorithm.

in each system, the match cost increases significantly and approximately linearly in the number of rules. Thus, the standard Rete algorithm does not scale well with the number of learned rules in any of these systems. In fact, the problem is so severe that it is infeasible to have many of the systems solve enough problems to learn 100,000 rules, because they simply become too slow. For the interested reader, Appendix B provides some additional measurements of various characteristics of these testbeds using the basic Rete algorithm.

time measurements, they ignore the costs of other aspects of the match algorithm. This is acceptable provided that those costs are insignificant, but in our testbed systems, they turn out to be very significant, as we shall see. Of course, each absolute CPU time measurement we give here is machine- and implementation-dependent. However, *comparisons* of CPU times are not machine- or implementation-dependent, since we run the same implementation, modulo changes we develop for the match algorithm, on the same machine.

3.3.1 Holding WM Changes Fixed

At this point we cannot be certain that the increase in match cost shown in Figure 3.1 is caused by the increase in the number of rules, since there is an additional factor changing over time: the distribution of working memory elements. If a learned rule influences the behavior (external actions or internal problem-solving) of the system, it will almost certainly alter the likelihood that certain WMEs will be present in working memory. For example, if a learned rule prevents a "blocks world" system from considering using an "unstack" operator, then that rule's presence would probably lessen the frequency of occurrence of WMEs representing "unstack" operators.

Thus, there are really two things changing over time: the number of rules and the distribution of WMEs. It might be that the increase in match cost is really due to the changing distribution of WMEs, not the number of rules. To discount this possibility, we introduce a different experimental methodology in which the distribution of WMEs is held fixed. For each system, we use a single sequence of changes to working memory — additions and removals of WMEs — and hold that sequence fixed, while varying the number of rules. We will continue to use this methodology for the rest of the experiments dealing with match cost in this thesis. In Chapter 7, when we examine overall system performance, we will use a different methodology.

How should we choose the working memory change sequence to use for a system? We want a sequence that is *representative* — specifically, representative of the typical distribution of WMEs in the system *in the long term.* For these testbed systems, the distribution of WMEs at the end of the run (when the system has already learned around 100,000 rules) is expected to approximate the long-term distribution. The reasons for this are specific to Soar.[5] For each system, we obtained a representative sequence by recording the last 100,000 WME additions and removals in the runs of Section 3.3 above.[6] We then fed this fixed sequence of working memory changes to the matcher repeatedly, each time with a different number of rules in the system.

Figure 3.2 shows the results of this on the testbed systems.[7] As the figure shows, in each system, the match cost is still increasing significantly and linearly in the number of rules — as the number of rules increases from 10,000 to 100,000, the match cost increases by approximately

[5] It is common for Soar systems to have certain problem spaces that are used frequently at the beginning of the run, but less and less as the run proceeds. This is because as the run proceeds, the system learns more and more rules (chunks) which in more and more situations eliminate the need to use that problem space: in situations where that space previously would have been used, some chunk is used instead. Eventually, the system may learn enough rules to avoid ever using the space again — the original, deliberate problem-solving in that problem space has been completely "compiled" into chunks. As a result, WMEs used by such spaces will tend to occur frequently early on, but infrequently in the long run. Thus, at the beginning of the run, the distribution of WMEs may be quite different from the long-term distribution, but as the system learns more and more rules, its current distribution gradually approaches the long-term distribution.

[6] Recording just the last 100,000 changes yields a sequence that may contain removals of WMEs that were added before the sequence started. To avoid these nonsensical removals, we prepended corresponding WME additions to the sequence.

[7] In contrast to Figure 3.1, Figure 3.2 shows each line all the way out to 100,000 rules. The rules were obtained by running each system using Rete/UL and saving the learned rules in a file. To get the data for Figure 3.2 and several other figures later in this thesis, the 100,000 rules were reloaded into a system which used the basic Rete algorithm.

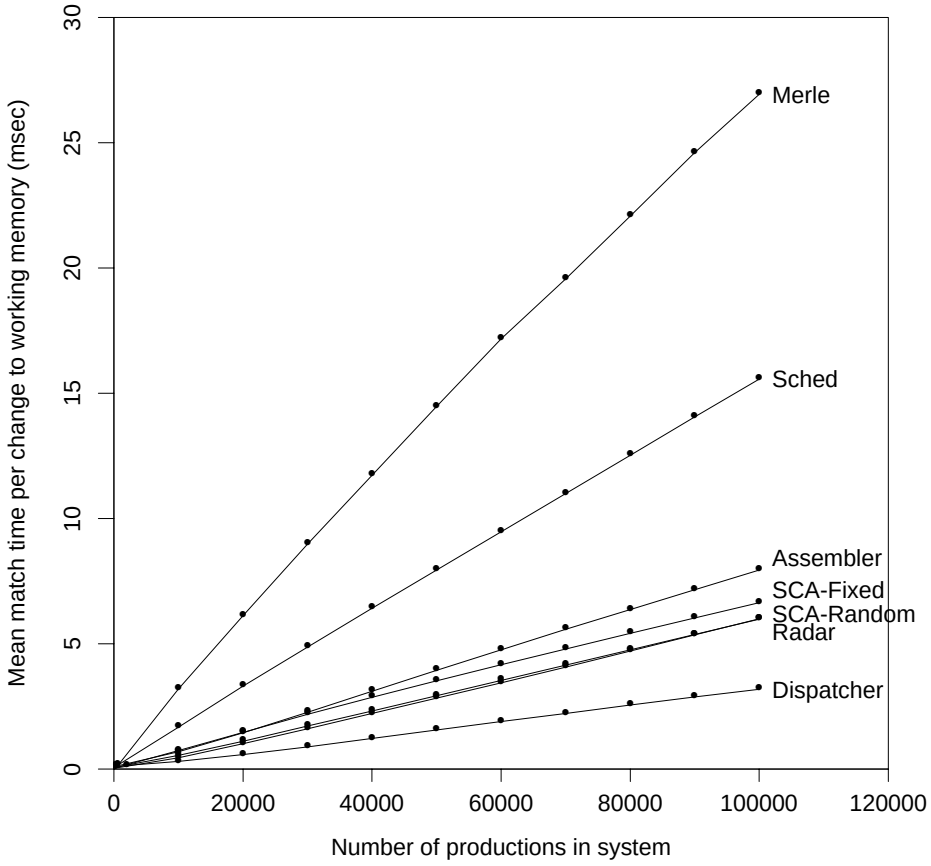


Figure 3.2: Match time with basic Rete, while holding the distribution of WMEs constant.

an order of magnitude in each system — even though we are holding the distribution of WMEs constant. We therefore conclude that the increasing match cost is (at least mostly) due to the increasing number of rules in each system, not the changing distribution of WMEs.

3.4 Number of Productions Affected by WMEs

The data shown in Figures 3.1 and 3.2 may come as a surprise to readers familiar with research on parallelism in production systems. This research has suggested that the match cost of a production system is limited, *independent* of the number of rules. This stems from several studies of OPS5 systems in which it was observed that only a few productions were affected by a change to working memory (Oflazer, 1984; Gupta et al., 1989). A production is said to be *affected* by a WME if that WME matches (the constant tests of) one of its conditions — i.e., if the WME goes into one of the alpha memories used for that production. The *affect set* of a WME is the set of productions affected by it. Quite small affect sets, containing just 20–30

productions on average, were observed in systems ranging from ~100 to ~1000 productions.

That only a few productions are affected by a WME is important because it means the match effort in the beta part of the Rete network is limited to just the nodes used to represent those few productions. This is a consequence of Rete's dataflow operation — the only beta nodes that get *activated* during Rete's handling of a WME change are nodes used to represent affected productions.[8] In essence, if only a few productions are affected by a WME, then as far as the match cost with the basic Rete algorithm is concerned, it is as if those few productions are the only productions in the system. The activity in the beta network is limited to the affected productions, and the alpha network can be implemented to run in approximately constant time per change to working memory (see Section 2.2). So based on the small affect sets observed in previous studies, one might conclude that Rete's performance ought to be roughly independent of the number of rules in the system.

However, these earlier studies were done on relatively small (~100 to ~1000 productions), hand-coded, non-learning systems only. To our knowledge, this thesis is the first study of the behavior of Rete on large rule sets generated by a machine learning algorithm. As we shall see, Rete's behavior looks quite different on these systems.

On small, hand-coded, non-learning systems, the affect set is never very large, and usually only 20–30 productions. This result does not hold for any of our large testbed systems. Figure 3.3 shows the average number of productions affected per change to working memory for each of the systems, plotted as a function of the number of productions. (The sequence of changes to working memory is held constant, as described in Section 3.3.1.) In all the systems, the average size of the affect set increases fairly linearly with the total number of productions in the system. With 100,000 productions in each system, the affect sets contain on the order of 10,000 productions on average. Appendix C shows the distribution of sizes.

It is important for readers familiar with previous research on production systems to keep this phenomenon in mind while reading the rest of this thesis. The occurrence of large affect sets is the fundamental phenomenon that sets these large learning systems apart from the smaller systems that have been previously studied. This has important consequences for match algorithm design and performance, as we shall see. Previous research on match algorithms has been aimed at addressing the efficiency problems caused by other phenomena, but not this phenomenon.

Given the importance of this phenomenon, three questions must be answered. Why do the affect sets become so large in these systems? Why do they tend to remain small in the previously

[8] In more detail, let us ask what nodes in the beta part of the network will be activated during Rete's handling of a WME change. (We'll consider a WME addition here; the argument for a WME removal is similar.) For each alpha memory the WME goes into, each of its successor join nodes will be activated. We'll refer to these join nodes as *primary* nodes. These primary nodes are used to represent (particular conditions of) productions which use that alpha memory; hence, they are used to represent affected productions (from the definition of "affected" above). Now, from each of those primary join nodes, dataflow may propagate down the network, activating other *secondary* nodes. Recall from Chapter 2 that the beta part of the network forms a tree, and each production is represented by a path from the root (top node) to a leaf (production node). Since each secondary node is a descendent of some primary node, any production the secondary node represents is also represented by that primary node, which means it must be an affected production. Since these (primary and secondary) are the only two ways nodes can be activated as a result of this WME addition, we conclude that the match effort in the beta part of the Rete network is limited to just the nodes used to represent affected productions.

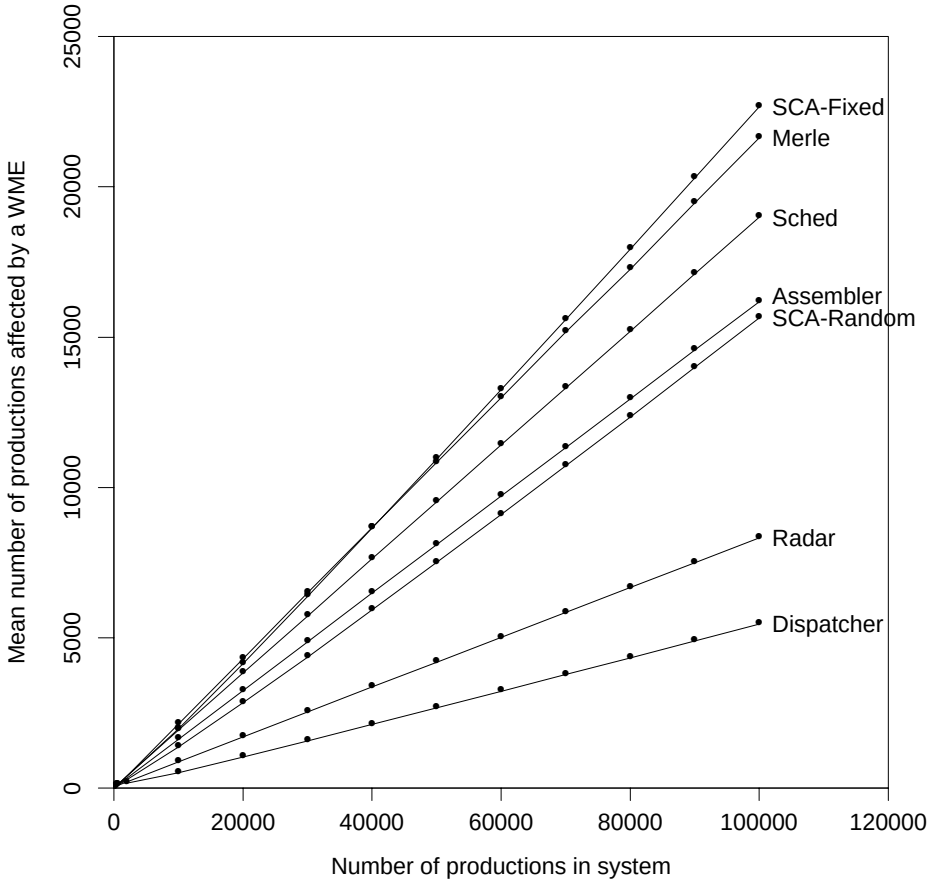


Figure 3.3: Average number of productions affected by each WME.

studied OPS5 systems? And are other large learning systems (besides these testbeds) likely to exhibit large affect sets as well, or are they more likely to resemble the OPS5 systems? We will take up each of these questions in turn.

Why are so many productions affected by individual WMEs in these systems? This is best explained by looking at an example rule from one of them. Figure 3.4 shows one of the rules learned by the SCA-Fixed system. (This is a printout from the system, edited slightly for clarity.) The rule can be roughly translated into English as, "If you are trying to predict the classification of an object, and your description of the object has six features, and the value of feature five is ten, and the value of feature three is six, ..., and the value of feature four is eight, then predict that the object is in class number seven." The rules learned by SCA-Fixed all have this same basic form. They differ in the number of features present on the object — `(<s> ^count 6)` in the figure — and in the particular features and values they test — the last twelve conditions in the figure.

Now, consider what happens when the system learns a large number of rules of this form. A WME `(G37 ^problem-space P29)` will affect every learned rule, since every one has a condition

```
(chunk-4253
  (<g> ^problem-space <p>) ; if the problem space used for the current goal ...
  (<p> ^name predict)      ;   ... is named ''predict''
  (<g> ^state <s>)         ; and the current state ...
  (<s> ^task predict)      ;   ... says the task is to make a prediction
  (<s> ^count 6)           ; and there are six features in the ...
  (<s> ^object <o>)        ;   ... description of the object being examined
  (<o> ^description <d>)
  (<d> ^f5 <f5>)           ; and feature #5 has value 10
  (<f5> ^value 10)
  (<d> ^f3 <f3>)           ; and feature #3 has value 6
  (<f3> ^value 6)
  (<d> ^f1 <f1>)           ; and feature #1 has value 1
  (<f1> ^value 1)
  (<d> ^f0 <f0>)           ; and feature #0 has value 1
  (<f0> ^value 1)
  (<d> ^f2 <f2>)           ; and feature #2 has value 0
  (<f2> ^value 0)
  (<d> ^f4 <f4>)           ; and feature #4 has value 8
  (<f4> ^value 8)
-->
  [ then predict that the object is in class 7 ]
```

Figure 3.4: Example rule learned by the SCA-Fixed system.

`(<g> ^problem-space <p>)`. So the number of rules affected by this WME will increase in direct proportion to the number of rules in the system. A WME `(S3 ^count 6)` will affect all the rules that look for six features in the description of the object. In SCA-Fixed, every rule tests between one and twelve features, so on average this WME would affect one twelfth of the rules,[9] so again the number of rules affected will increase in direct proportion to the number of rules in the system. A WME `(D4 ^f3 I4)` will affect all the rules which pay attention to feature number three; again, this will increase in direct proportion to the number of rules in the system.

The preceding example was taken from SCA-Fixed. Similar (though sometimes less pronounced) effects arise in the other testbed systems. The learned rules in each system contain conditions testing WMEs indicating the general problem-solving context (e.g., the current goal, problem space, state, or operator), so WMEs specifying this context affect most of the rules. The rules also test various specific facts about the particular problem instance being worked on, or the current internal problem-solving situation, like the features and their values in the

[9] For the sake of argument, we ignore the possibility that some numbers of features may be more likely than others.

SCA-Fixed example above.[10] For some of these particular facts, the number of rules testing that fact increases linearly in the total number of rules in the system. Some facts are not tested by very many rules, and some WMEs still affect only a handful of rules, but WMEs with large affect sets occur often enough to drive the average quite high. Consequently, the average number of productions affected by WMEs increases linearly in the total number of rules in the system.

We now turn to the question of why large affect sets were not observed in previous studies of OPS5 systems. The obvious hypothesis — the relatively small number of rules in those systems — is incorrect. This hypothesis would predict that the average number of affected rules in each of these OPS5 systems would be proportional to the total number of rules it has. But this is not the case: the *same* small affect set size (20–30 rules) is observed in systems ranging from a few hundred to a few thousand total rules.

A more likely reason the affect sets remain small in these OPS5 systems has to do with the way people write programs (Gupta, 1987, pages 56–57). People often hierarchically decompose large problems into smaller subproblems, then write a few rules to solve each lowest-level subproblem. In fact, two major books on OPS5 system development describe this approach in detail (Brownston et al., 1985, pages 197–203) and encourage its use in OPS5 systems (Cooper and Wogrin, 1988, Chapter 5). Each lowest-level subproblem is implemented by a small set of rules, called a *rule cluster* or *context*. Certain working memory elements, called *control WMEs* or *context WMEs*, are used to indicate which subproblem is currently being worked on. For example, suppose `check-constraint` is the (user-supplied) name of one lowest-level subproblem. Then a WME `(current-subproblem ^name check-constraint)` would indicate that this subproblem is being worked on. In the rule cluster for this subproblem, each rule would have a condition (usually the first condition) testing for the presence of this control WME in working memory. The other conditions in the rules in that cluster would test certain relevant facts in working memory.

The use of this programming technique tends to keep affect sets small.[11] When it is used, there are two basic types of WMEs in working memory: control WMEs and facts. Each control WME (e.g., `(current-subproblem ^name check-constraint)`) only affects the rules in the corresponding rule cluster; since there are only a small number of rules per cluster, each control WME affects only a small number of rules. Moreover, each fact in working memory tends to be relevant to only one lowest-level subproblem, or perhaps a handful of them, so again we only have a small number of rules affected by it.

Finally, we turn to the question of whether other large learning systems (besides these testbed systems) are likely to exhibit large affect sets, or whether they are more likely to resemble the aforementioned OPS5 systems. Our analysis here is, of course, rather speculative, since we cannot predict with certainty what characteristics future large learning systems will have.

[10] We need not make a precise distinction here between conditions testing "the general problem-solving context" and ones testing "facts about the particular problem instance." For purposes of understanding why the affect sets are so large, this is a useful way of viewing conditions, but the match algorithm makes no such distinction — it has no knowledge of the domain-level semantics of the conditions and WMEs.

[11] This has been noted by (Stolfo et al., 1991), which then advocates using production rule languages other than OPS5 so as to obtain larger affect sets. The use of such languages would make the optimizations for Rete introduced in this thesis, designed especially for systems with large affect sets, even more important.

Nevertheless, we suggest that many are likely to have large affect sets.

It is certainly possible that future machine learning systems could be designed to create and maintain small rule clusters just as human programmers do. Such systems would have to learn new domains in hierarchical fashion, decomposing large problems into smaller ones, deciding which ones were small enough to be solvable with a small number of rules, and building a new rule cluster for each lowest-level subproblem. Additionally, when such systems acquire a large amount of new knowledge about an existing lowest-level subproblem, instead of adding a large number of new rules to the existing cluster, they would have to "split" the existing cluster, revising existing rules as necessary to further decompose the subproblem before adding the new rules. However, this is clearly beyond the current state of the art in machine learning. Rather than placing these constraints on future learning techniques, an alternative approach is simply to improve the match algorithm so that small rule clusters are not necessary for efficiency.

Not only are current machine learning systems not designed to keep affect sets small, but some of them have features that directly lead to large affect sets. First, general-purpose search-oriented problem solvers such as Soar and Prodigy (Minton et al., 1989) often represent their current problem-solving context using a group of general WMEs designating the current goal, current state or search node, and current (selected or candidate) operator(s). Because they are so general, these WMEs tend to affect a large number of rules. For example, one of these WMEs might designate which search node is the current one, but all details of that node are represented in *other* WMEs. Of course, which search node is current tends to be relevant to many rules. If all the information about the current node were "compressed" or "flattened" into one WME, that WME would tend to affect fewer rules; however, this form of representation makes it difficult to represent complex objects and relationships between them.

Second, knowledge compilation mechanisms often tend to increase the sizes of affect sets. Such mechanisms may generate rules that act as "macros," solving many subproblems at once. This tends to increase the number of rules in the system affected by fact WMEs for a given subproblem: those facts now affect both the original rules (in one cluster for that subproblem) and the new macro-rules (solving that subproblem and others).

Although we cannot predict with certainty what characteristics future large learning systems will have, the above analysis suggests that large affect sets will arise in a broad class of systems, not just the particular testbed systems studied here.

3.5 Sharing

Given the increase in the number of productions affected by changes to working memory, how can we avoid a slowdown in the matcher? One partial solution can already be found in the basic Rete algorithm. As discussed in Chapter 2, when two or more productions have the same first few conditions, the same nodes in the beta part of the Rete network are used to match those conditions (see Figure 2.2 on page 10). By sharing nodes among productions in this way, Rete avoids duplicating match effort across those productions.

In our testbed systems, this sharing becomes increasingly important as more and more rules are learned. This is because the learned rules often have many conditions in common. For

instance, consider the first four conditions from the rule shown in Figure 3.4:

```
(<g> ^problem-space <p>) ; if the problem space used for the current goal ...
(<p> ^name predict)      ;   ... is named ''predict''
(<g> ^state <s>)         ; and the current state ...
(<s> ^task predict)      ;   ... says the task is to make a prediction
```

All the learned rules in SCA-Fixed have these conditions in common, so the same Rete nodes will be used to represent them.[12] The more rules the system learns, the more rules will share these nodes, and hence the more match effort will be saved via sharing. Similar effects arise in all the testbed systems.

Figure 3.5 shows, for each system, the factor by which sharing reduces the number of tokens generated in the basic Rete algorithm. The y-axis is obtained by taking the number of tokens that would have been generated if sharing were disabled, and dividing it by the number that actually were generated with sharing. This ratio is plotted as a function of the number of rules in each of the systems.[13] This shows the result of sharing in the beta part of the network only, not sharing of alpha memories or nodes in the alpha network. With just the initial rules (no learned rules) in each system, sharing reduces the number of tokens by a factor of 1.5 (in SCA-Fixed) to 10.2 (in Dispatcher). This factor increases dramatically as the number of rules in each system increases. With 100,000 rules in each system, sharing reduces the number of tokens by a factor of $\sim$450 (in SCA-Random) to $\sim$1000 (in Merle).[14]

Note that Figure 3.5 shows the ratio of token counts, not CPU time measurements. The ratio of CPU times would be approximately a constant factor (not necessarily the same factor in each system), independent of the number of rules. The reason is as follows. The basic Rete algorithm slows down linearly in the number of rules even *with* sharing, as Figure 3.2 shows. If we disable sharing, the match process would proceed independently for each affected rule; since the number of affected rules increases linearly in the total number of rules (as Figure 3.3 shows), the match cost would increase linearly as well. Thus, the basic Rete algorithm slows down linearly in the total number of rules in these testbed systems, with or without sharing. The ratio of CPU times is thus the ratio of two linearly increasing functions, so it asymptotically approaches some constant factor as the number of rules grows very large.

If Figure 3.5 does not reflect the actual speedup factor we gain by using sharing in the basic Rete algorithm, why is it important? It is important because it shows that the cost of *one part of the Rete algorithm* increases linearly if we do not use sharing. The overall match cost can be broken down into several parts. If we want to avoid a linear increase in overall match cost, then we must avoid having the cost of any one part increase linearly. Later in this thesis, we develop

[12] Assuming the conditions get ordered in the same way. See Section 6.3.3.

[13] A previous paper (Doorenbos, 1993) reported that sharing was less effective in Assembler, yielding only a factor of six. This was an artifact of using a different implementation of Rete, an implementation which sometimes did not share memory nodes in places where our current implementation does. The previous implementation would use a merged beta-memory-plus-join node even when the memory had more than one child join node — see Section 2.9.2.

[14] The figure shows the increase to be roughly linear in all the testbed systems except SCA-Random, where the increase is sublinear (but still over two orders of magnitude). The difference is due to a larger number of tokens generated in unshared portions of the Rete network in SCA-Random, as will be discussed later in Section 6.3.2.

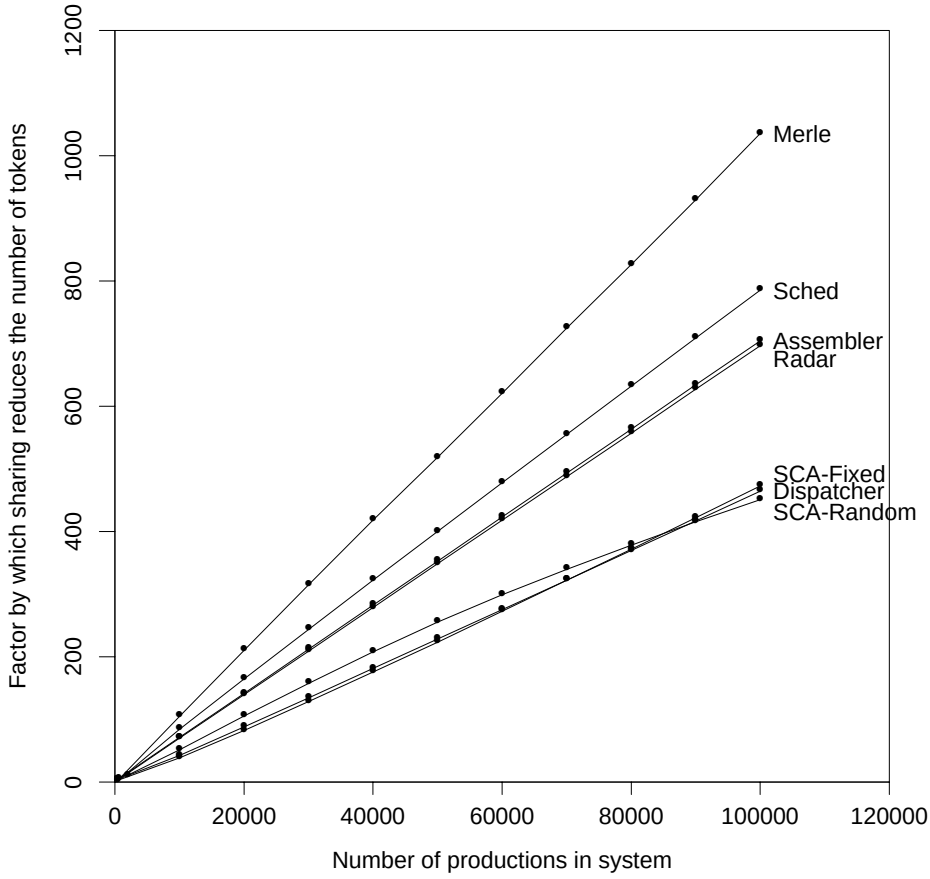


Figure 3.5: Reduction in tokens due to sharing.

techniques for avoiding having the cost of certain *other* parts increase linearly. The data shown in Figure 3.5 imply that without sharing, all our efforts would be for naught, since the cost of generating tokens would increase linearly.

3.6 Discussion

The large average number of affected productions, increasing linearly in the total number of rules in each of our testbed systems, is the fundamental source of increasing match cost in these systems. Since this problem has not been observed in previous studies of production systems, previous research on production match algorithms has not been aimed at solving it. As a result, most previous research would be of no help in avoiding the linear increase in match cost in these systems. Of the optimizations mentioned in Section 2.10, modify-in-place, scaffolding, collection Rete, and Uni-Rete might yield small constant factor improvements, but they are essentially orthogonal to the problem of large affect sets. Alternative ways of implementing the

alpha network cannot solve the problem, because the bad effects of large affect sets arise in the beta network, not the alpha network. Lazy match and other ways of incorporating conflict resolution into the matcher are not applicable in these systems, because Soar does not employ any conflict resolution methods.[15] Two optimizations — factored arc consistency and nonlinear topologies — may have the potential to address this problem, but previous work on them has not studied this possibility. Factored arc consistency is discussed in Section 5.8.4. Although we touch on nonlinear networks briefly in Section 5.7, a thorough study is beyond the scope of this thesis.

In addition to the large number of affected productions, a second difference between the large learning systems we study here and previously studied production systems is the importance of sharing. The importance of sharing in our testbed systems was demonstrated in Section 3.5. However, previous measurements on smaller systems have found sharing of beta nodes to produce only very limited benefit, typically only a factor of 1–2 (Gupta, 1987; Tambe et al., 1988; Miranker, 1990). This is probably due to the small affect sets in those systems.

Because of this limited benefit, previous work in production systems has often ignored sharing. For example, the Treat match algorithm does not incorporate sharing in the beta part of the match. Consequently, on each working memory change, it must loop over all the affected rules. Although Treat incorporates an optimization (*rule-active* flags) which may enable each affected rule to be processed very quickly, the number of iterations of the loop would increase linearly in the total number of rules in each of our testbed systems. So Treat would not scale well for these systems — like Rete, Treat would slow down linearly in the total number of rules. We will later discuss Treat in more detail, including the possibility of incorporating sharing into it, in Section 5.8.5.

The Match Box algorithm (Perlin and Debaud, 1989; Perlin, 1991a), another recent development, also fails to incorporate any form of sharing of match effort across productions. Match Box differs from Rete in that it operates on tuples of *variable bindings* rather than on tuples of *WMEs*. Due to its lack of sharing, Match Box requires work linear in the number of affected productions. It may be possible to incorporate sharing into Match Box, but that is beyond the scope of this thesis.

Work on machine learning systems has also often failed to incorporate sharing into the match process.[16] For example, the match algorithm used by Prodigy (Minton, 1988a) treats each rule independently of all the others. As more and more rules are learned, the match cost increases, leading to a utility problem. Prodigy's approach to this problem is to discard many of the learned rules in order to avoid the match cost. An alternative approach was tried by (Doorenbos and Veloso, 1993): sharing was incorporated into the matcher, with the rules organized as a tree much like the beta part of a Rete network (but without any alpha network or any memory nodes — the match is repeated from scratch every cycle). While this did not eliminate the

[15] Soar's version of conflict resolution takes place at a higher level, in its *decision procedure*, which selects one operator to apply, rather than one production to fire.

[16] Notable exceptions include (Veloso, 1992), which organizes a case library into a tree in which parts of descriptions of problem-solving cases are shared, and (Wogulis and Langley, 1989), which stores learned concept definitions in a hierarchy so the match effort for common subconcepts can be shared between multiple higher-level concepts.

increase in match cost, it did speed up the matcher by up to a factor of two, and in one case it avoided the utility problem — with the new matcher, the increase in match cost was offset by a reduction in the number of search nodes the system explored. (It the other cases, sharing reduced the increase in match cost, but not by enough to avoid the utility problem.)

We will return to discuss the effect of match algorithms on the utility problem further in Chapter 7, but first we need to examine the linear increase in match cost we observed in the basic Rete algorithm in this chapter. The next two chapters discuss the two major sources of this linear increase, and develop techniques to eliminate them.

Chapter 4

Adding Right Unlinking to Rete

In the previous chapter we showed that in each of our testbed systems, the basic Rete algorithm slows down significantly and linearly in the number of rules, in spite of the effectiveness of sharing. How can such a slowdown arise? As we saw in Section 3.4, a fundamental phenomenon in these systems is the very large average number of productions affected by a WME. Recall that if a production is affected by a WME, there is some join node[1] (used to represent one of the conditions of the production) that gets right-activated when that WME is added to working memory. If a WME affects *many* productions, there *might* be only *one* join node that gets right-activated — one node shared by all the affected productions. This often arises with nodes near the top of the network, shared by many productions. In this case, we have only one right activation. (Dataflow may propagate down the network from that node, causing left activations of many other nodes, and the number of such left activations can increase with the number of rules in the system; we will discuss this possible cause of Rete's slowdown in Chapter 5.) On the other hand, if a WME affects many productions, there might be *many* (unshared) join nodes, one for each affected production, that get right-activated. In this case, the number of right activations can increase linearly in the number of rules in the system. This possible cause of Rete's slowdown is the focus of the current chapter.

After we look at a concrete example of this increasing number of right activations, we explain in Section 4.1 why most of these activations are useless work. Section 4.2 explains the basic idea of *right unlinking*, our extension to Rete which avoids most of these activations. Section 4.3 presents the implementation of right unlinking. Finally, Section 4.4 gives empirical results.

We begin with a concrete example. Figure 4.1 shows the Rete network for the rules learned by the SCA-Fixed testbed system. (One example rule is shown in Figure 3.4 on page 73.) The figure is substantially simplified for expository purposes. It shows only the alpha memories and join nodes; all the beta memory nodes are omitted. It also assumes that all the rules test the same number of features (six, named `f1` through `f6`); it assumes that each feature has ten possible values; it "compresses" each feature/value test into a single condition, instead of two conditions as in Figure 3.4; and it assumes that the conditions in different rules are given similar orderings before the rules are added to the Rete network.

[1] Or negative node. In this chapter, what we say about join nodes applies to negative nodes as well.

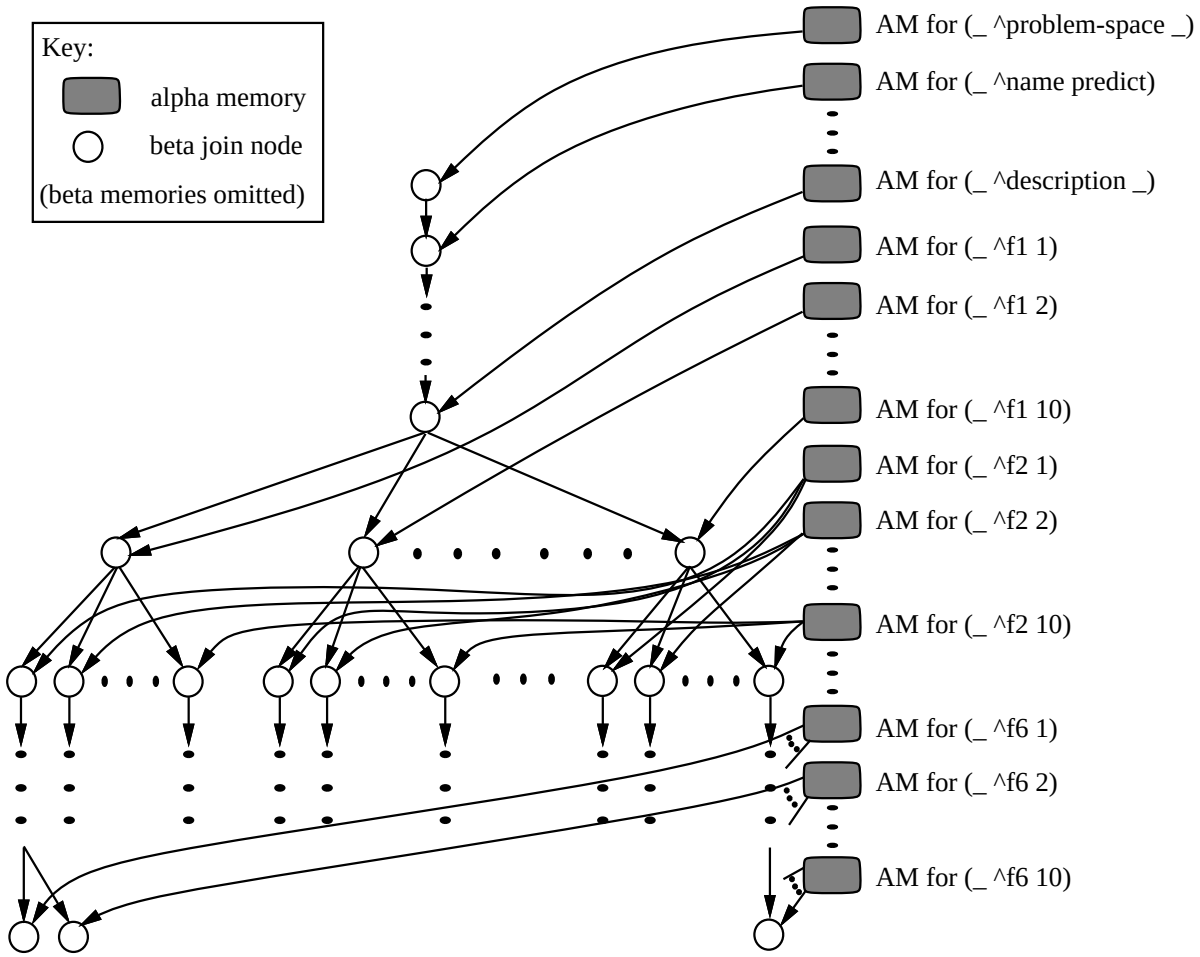


Figure 4.1: Simplified view of the Rete network for SCA-Fixed.

As the figure shows, the rules share the same join nodes for their first few conditions, down to the first feature/value test. The more rules we have, the more share these join nodes, and the more match effort will be saved by the use of this sharing. The portion of the network below these shared join nodes is a simple subtree with a depth of six and a branching factor of ten. (In the actual SCA-Fixed Rete, only some of the 10^6 leaves would be present; the number would increase as the system learned more and more rules.)

Whenever a new match is found for the first few conditions, the dataflow propagates down this subtree, starting at the top. First the ten children testing different values of `f1` are left-activated. Assuming the object in question has exactly one value for `f1`, nine of these left activations will "fail" and one will "succeed" — dataflow will be propagated further from only one of the ten join nodes. Assuming the object in question has exactly one value for each of the other five features, the dataflow will propagate down five more levels of the subtree in the same way. The total number of left activations in this subtree is sixty-one (ten activations at each of six levels, plus one activation of the top node in the subtree). This is quite a moderate number of activations, considering that we have 10^6 rules represented in this Rete network.

However, when we encounter a new value of a feature, we may have a serious problem. Consider what happens when we add a WME `(I27 ^f6 2)`. The WME is first added to the appropriate alpha memory. Then, we right-activate all its successor join nodes — all 10^5 of them, since 1/10 of the 10^6 leaves in the subtree are successors of this alpha memory. In general, adding a WME representing the value of feature i would cause 10^{i-1} join nodes to be right-activated. Moreover, the number of right activations triggered by `(I27 ^f6 2)` increases with the number of rules in the system. Initially, the system has no learned rules, and this whole tree is not present in the Rete network. As more and more rules are learned, the tree gradually fills out. With $n \leq 10^6$ rules, there are n leaves in the tree, and a WME representing the value of `f6` triggers (on average) $n/10$ right activations.

Note that there is no way to avoid the problem by changing the ordering of the conditions here. If we move the conditions testing `f6` up higher in the network, sharing will reduce the number of right activations from WMEs for `f6`; however, this comes at the expense of increasing the number of right activations from some other feature which takes `f6`'s place at the bottom of the network.

To generalize this situation beyond the SCA-Fixed testbed system, consider a system with a large number n of rules, where no rule has conditions which are a subset of those of another rule; i.e, no rule is a strict generalization of another rule, and no two rules are duplicates. Suppose that a relatively small number k of alpha memories are required for all the rules. (In the SCA-Fixed example above, n would be 10^6 and k would be 60–70.) For each rule, there is at least one *unshared* join node — if all the join nodes for a rule were shared with some other rule, we would have a duplicate or generalized rule. This implies that there are at least n unshared join nodes. Each one is a successor of some alpha memory, but there are only k alpha memories. If we add k WMEs to working memory, chosen so that at least one goes into each alpha memory, this will trigger a right activation of each of these join nodes. Thus, there is a small set of k WME additions which together cause an average of n/k right activations each; of course, this number increases linearly in the total number of rules in the system.

4.1 Null Right Activations

Continuing our SCA-Fixed example from above, it is important to note that of the 10^5 right activations triggered by the addition of `(I27 ^f6 2)`, only one will "succeed" (i.e., result in a new match being found). The *join-node-right-activation* procedure below (copied here from page 24) will be called 10^5 times, once for each of 10^5 different join nodes.

```
procedure join-node-right-activation (node: join-node, w: WME)
    for each t in node.parent.items do {"parent" is the beta memory node}
        if perform-join-tests (node.tests, t, w) then
            for each child in node.children do left-activation (child, t, w)
end
```

Each time, the procedure iterates over all the tokens in the beta memory just above that join node (the beta memories are not shown in Figure 4.1). But the beta memory will not have any

tokens in it unless there is at least one match in working memory for all the earlier conditions in the production. For example, consider the join node at the bottom left of Figure 4.1 (the leftmost leaf in the tree). The beta memory just above it store matches for all the earlier conditions — conditions testing that `f1` has value 1, `f2` has value 1, ..., and `f5` has value 1. This beta memory will be empty unless there are WMEs in working memory indicating that the object in question does have all these properties. Other beta memories for other join nodes at the bottom level store matches for other sets of conditions which test for other values of these features. Assuming that the object has just one value for each feature, all but one of these beta memories will be empty.

More generally, in most systems, one expects most join nodes near the bottom of the network to have empty beta memories at any given time. This is because usually there is some join node higher up that has no matches — some earlier condition in the rule fails. To see why, consider a rule with conditions $c_1, c_2, \ldots, c_k$, and suppose the c_i's have independent probabilities p_i of having a match. For the very last join node to have a non-empty beta memory, all the earlier conditions must match; this happens with the relatively low probability $p_1 p_2 \cdots p_{k-1}$. So at any given time, most of these join nodes have empty beta memories.

A right activation in which the join node's beta memory is empty is called a *null right activation*. On a null right activation, since there are no tokens for the *join-node-right-activation* procedure to iterate over, no iterations of the **for** loop are performed, and the whole procedure call is essentially useless work. In the above example, `(I27 ^f6 2)` will trigger $10^5 - 1$ null right activations — all wasted effort — plus one non-null right activation. Moreover, as the system grows, learning more and more rules and gradually filling out the tree, the number of non-null right activations remains at most one, while the number of null right activations increases linearly in the total number of rules in the system. Although each individual null right activation takes only a handful of CPU instructions (procedure call overhead plus checking the termination condition of the **for** loop), the number of null right activations grows very large, so that their cost becomes the dominant factor in the overall match cost.

Our discussion so far has focussed on a particular example, a simplified view of the SCA-Fixed testbed system, since it makes a good illustration of how the problem of increasing null right activations arises. However, this problem arises in all the other testbed systems as well. Figure 4.2 shows, for each testbed system, the average number of null right activations of join nodes per change to working memory, plotted as a function of the number of rules in the system. It clearly shows that in each system, the number of null right activations increases linearly in the number of rules. With 100,000 rules in each system, the number ranges from ~2,000 in Dispatcher to ~17,000 in Merle. (The number varies from one system to another, depending on what proportion of WMEs get added to alpha memories with large fan-outs, and on the magnitude of those fan-outs.)

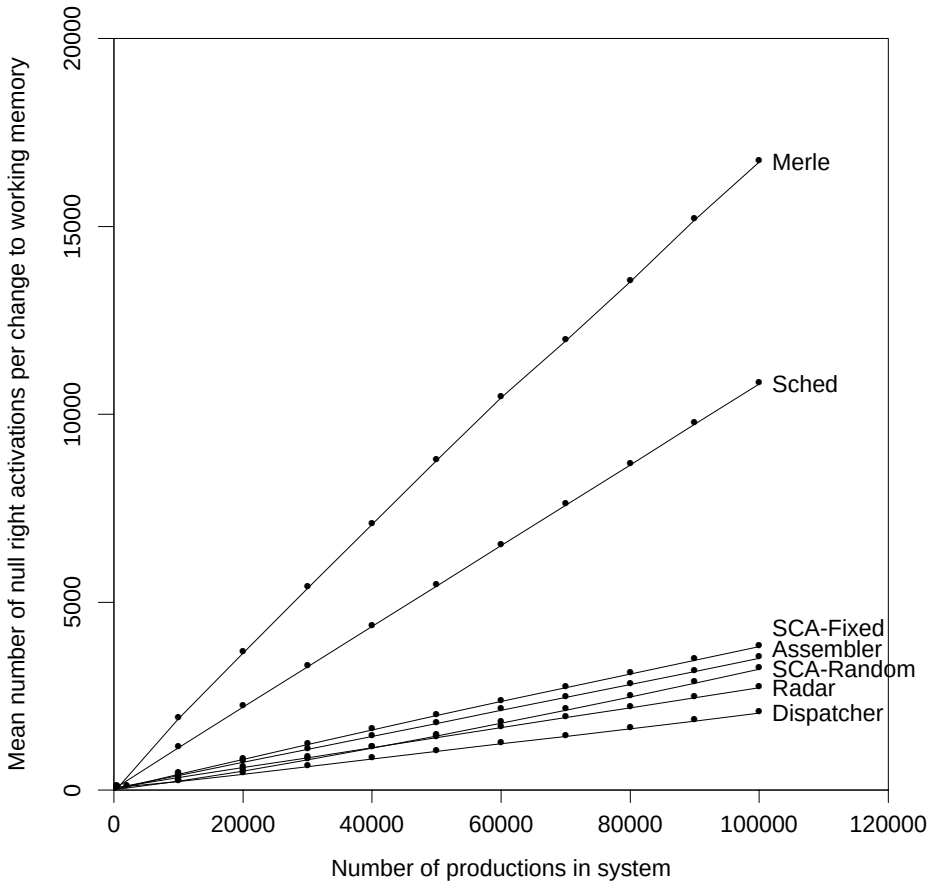


Figure 4.2: Number of null right activations per change to working memory.

4.2 Right Unlinking: Basic Idea

In this section, we introduce an optimization for the Rete algorithm which avoids all null right activations.[2] With this optimization incorporated into the matcher, we will no longer see the linear increase in null right activations shown in Figure 4.2.

By saying we avoid a null right activation, we do not mean just avoiding executing a procedure call — that could be easily accomplished simply by "inlining" the *join-node-right-activation* procedure everywhere it is called. While this would reduce the cost of null right activations somewhat (instead of each one incurring the cost of a procedure call overhead plus the cost of checking the termination condition of the **for** loop, it would only incur the cost of checking the termination condition), the total cost of null right activations would still increase linearly in the number of rules. To avoid having the total cost increase, the cost of each null right activation must be zero. So when we say we avoid a null right activation, we mean we avoid *executing any*

[2]The optimization we present here is not the only possible way to avoid null right activations; we discuss some others later, in Section 5.8.

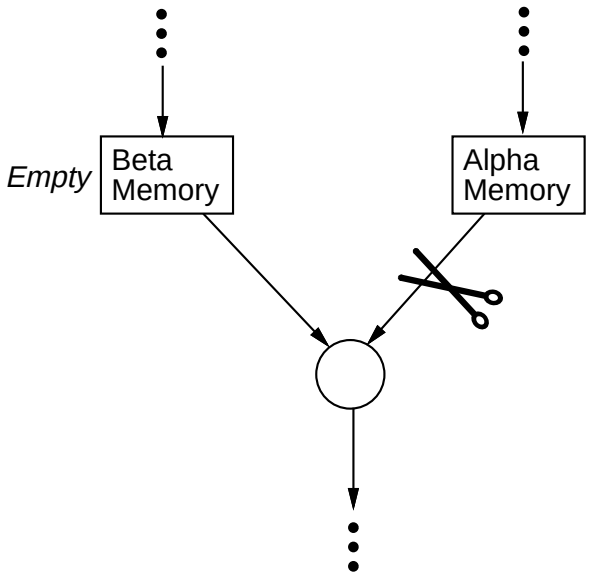


Figure 4.3: Unlinking a join node from its associated alpha memory.

CPU instructions for it — i.e., the alpha memory does not even "look at" the join node or its beta memory.

Our optimization is called *right unlinking*. The basic idea is that if we know in advance that a join node's beta memory is empty, then we will arrange for right activations of that join node to be skipped. We do this by unlinking the join node from its alpha memory; i.e., removing the join node from the alpha memory's *successors* list. This is illustrated in Figure 4.3: whenever the beta memory is empty, we cut the link (the dataflow path) from the alpha memory (the one on the right) to the join node. Note that since the only activations we are skipping are null right activations — which would not yield a match anyway — this optimization does not affect the set of complete production matches that will be found; the semantics of the algorithm (viewed as a "black box") are unchanged.

In a running system, as WMEs are added to and removed from working memory, beta memories will sometimes change from empty to nonempty or vice-versa; we will dynamically splice join nodes out of and back into the *successors* lists on alpha memories when this happens. If a join node has been spliced out of its alpha memory's *successors* list, then when a WME is added to that alpha memory, the join node will not be right-activated; in fact, the match algorithm will perform *no CPU instructions at all* pertaining to that join node, since the *alpha-memory-activation* procedure (page 32) just iterates over the alpha memory's *successors* list, and this join node is not in that list.

Another way of looking at right unlinking is to view it as a way to reduce the (potentially large) fan-out from alpha memories. In Figure 4.1, for example, the lowest few alpha memories, for conditions testing values of `f6`, each have fan-outs of 10^5 in the basic Rete algorithm. If we added right unlinking to the algorithm, this fan-out would be reduced to zero or one — most of the join nodes have empty beta memories, hence would not be on *successors* lists.

4.3 Right Unlinking: Implementation

To implement right unlinking, we modify the Rete interpreter routines so they dynamically splice join nodes out of and back into their alpha memories' *successors* lists. We add code to detect when the number of items in a beta memory changes from one to zero, and remove the memory's child join nodes from the appropriate *successors* lists when this happens. We also add code to insert join nodes back onto *successors* lists when their beta memories are no longer empty.

Although we have only discussed right unlinking of (positive) join nodes so far, it can also be done on negative nodes. Recall from Section 2.7 that negative nodes store tokens (matches for the previous conditions) along with local result memories. If there are no tokens in a given negative node, we can unlink it from its associated alpha memory.

We first modify the *delete-token-and-descendents* procedure. Whenever it deletes a token from a memory node, it now also checks whether the memory has just become empty.[3] If so, it iterates over the memory's child join nodes, unlinking each one from its associated alpha memory. Also, whenever it deletes a token from a negative node, it checks whether there are any other tokens at that node; if not, it unlinks that node from its associated alpha memory. (The lines in the margin indicate the changed parts of the pseudocode below.)

```
procedure delete-token-and-descendents (tok: token) {revised from page 51}
    while tok.children ≠ nil do
        delete-token-and-descendents (the first item on tok.children)
    if tok.node is not an NCC partner node then
        remove tok from the list tok.node.items
    if tok.wme ≠ nil then remove tok from the list tok.wme.tokens
    remove tok from the list tok.parent.children
    if tok.node is a memory node then
        if tok.node.items = nil then
            for each child in tok.node.children do
                remove child from the list child.amem.successors
    if tok.node is a negative node then
        if tok.node.items = nil then
            remove tok.node from the list tok.node.amem.successors
        for each jr in tok.join-results do
            remove jr from the list jr.w.negative-join-results
            deallocate memory for jr
    ...{extra handling for NCC's unchanged from page 51} ...
    deallocate memory for tok
end
```

[3] The pseudocode below and the implementation used in this thesis do this by checking whether the memory's *items* list is now NIL. In some other implementations of Rete, an empty/nonempty indicator may not already be conveniently available. In that case, a *count* field can be added to the data structure for each memory node to store a count of the number of tokens it contains. Other procedures are then modified to update this *count* when tokens are added to or deleted from the memory.

Next, we modify the *join-node-left-activation* procedure. Whenever a beta memory changes from empty to nonempty, each child join node will be left-activated (with the new token that was just added to the beta memory); we must relink each child join node to its associated alpha memory at this point.[4] This pseudocode uses a procedure *relink-to-alpha-memory* to do the relinking; this procedure will be given below.

```
procedure join-node-left-activation (node: join-node, t: token)
{revised from version on page 32}
    if node.parent just became nonempty[5] then relink-to-alpha-memory (node)
    for each item in node.amem.items do
        if perform-join-tests (node.tests, t, item.wme) then
            for each child in node.children do left-activation (child, t, item.wme)
end
```

We also modify the *negative-node-left-activation* procedure. Whenever this procedure is called, if the negative node currently has no tokens in it (i.e., this procedure is about to build and store the first one), we must relink the negative node to its associated alpha memory.

```
procedure negative-node-left-activation (node: negative-node, t: token, w: WME)
{revised from version on page 42}
    if node.items = nil then relink-to-alpha-memory (node)
    {build and store a new token, just like a beta memory would (page 30)}
    new-token ← make-token (node, t, w)
    insert new-token at the head of node.items

    {compute the join results}
    new-token.join-results ← nil
    for each item in node.amem.items do
        if perform-join-tests (node.tests, new-token, item.wme) then
            jr ← allocate-memory()
            jr.owner ← new-token; jr.wme ← w
            insert jr at the head of the list new-token.join-results
            insert jr at the head of the list w.negative-join-results

    {If join results is empty, then inform children}
    if new-token.join-results=nil then
        for each child in node.children do left-activation (child, new-token, nil )
end
```

[4] This relinking could also be done in the *beta-memory-left-activation* procedure. The choice makes little difference as far as efficiency is concerned. We chose to put it in the *join-node-left-activation* procedure to make it symmetrical with some code we will later add in Chapter 5; see Footnote 2 on page 102.

[5] There are several ways of determining this. We can have the *beta-memory-left-activation* procedure pass the *join-node-left-activation* procedure a flag indicating this. Alternatively, we can just check whether the number of items in the beta memory is exactly one. Another approach (taken by the implementation used in this thesis) is to keep a flag on each join node indicating whether it is currently right-unlinked.

A few other minor modifications to the pseudocode are needed. The routines that build the Rete network must be modified so they properly initialize newly-created join and negative nodes; e.g., a newly-created join node should *not* be linked to its associated alpha memory if its beta memory is empty. These modifications can be found in Appendix A but for brevity will not be shown here.

4.3.1 Relinking: Strictly Tree-Structured Networks

It remains to discuss the *relink-to-alpha-memory* procedure. We must be careful here. Recall from Section 2.4.1 that in order to avoid creating duplicate tokens, we must ensure that the join nodes on each alpha memory's *successors* list are appropriately ordered. If J_1 and J_2 are on the list and J_1 is a descendent of J_2, then J_1 must come *earlier* in the list than J_2. This property is (trivially) maintained whenever a node is *removed* from a *successors* list. But when we relink a node to its alpha memory, we must be careful to splice it into the *successors* list in the right place, so as to maintain this property. Consider relinking a node J_{new} to its alpha memory. For the property to be maintained, we need two things:

1. If there is an ancestor of J_{new} (call it J_{old}) already on the *successors* list, then J_{new} must end up on the list *before* J_{old}; and
2. If there is a descendent of J_{new} (call it J_{old}) already on the *successors* list, then J_{new} must end up on the list *after* J_{old}.

It turns out that this is easy if we do not allow conjunctive negations (NCC's), but somewhat tricky otherwise. Recall from Section 2.8 that when conjunctive negations are allowed, the beta part of the network is not quite a tree, due to the subnetwork used for the conditions inside the NCC. We first consider the case without conjunctive negations, i.e., where the beta part of the network forms a tree. In this case, we can simply insert the node at the head of the *successors* list, and the property will be maintained. Clause 1 above trivially holds when J_{new} is inserted at the *head* of the list. For clause 2 to hold, it must be the case that there are no descendents J_{old} of J_{new} on the list when we splice in J_{new}. Suppose J_{old} is a descendent of J_{new}, and we are about to relink J_{new} to its alpha memory. The only time we do this relinking of J_{new} is right after its beta memory changes from empty to nonempty. If J_{new}'s beta memory was empty, then so must have been all the beta memories in the subtree underneath J_{new}. In particular, J_{old}'s beta memory must have been empty. Since the dataflow has not gotten down below J_{new} (the relinking occurs at the start of J_{new}'s activation procedure), J_{old}'s beta memory is still empty — which means J_{old} is currently right-unlinked. Thus, no descendent J_{old} of J_{new} could be on the *successors* list when we splice in J_{new}. Thus, the desired property is maintained.

If we have no conjunctive negations, then, the following simple procedure suffices. This would normally be coded "inline" for efficiency.

```
procedure relink-to-alpha-memory (node: rete-node)
{simple version, does not allow conjunctive negations}
    insert node at the head of the list node.amem.successors
end
```

4.3.2 Relinking: Totally Ordered Ancestors

Unfortunately, the above procedure breaks down in the presence of conjunctive negations. The problem is that in the discussion of clause 2, the fact that J_{new}'s beta memory is empty does not necessarily mean that the beta memories of all its descendents are empty. If J_{new} is a node in the *subnetwork* for an NCC (see Figure 2.7 on page 46), this may not be true. Basically, a lack of matches in the NCC subnetwork implies that the NCC is satisfied; hence dataflow propagates down from the NCC node to other nodes below it. Thus, if J_{new} is a join node in the subnetwork for an NCC, and J_{new} has an empty beta memory, there could be a J_{old} with a nonempty beta memory somewhere below the NCC node.

To support conjunctive negations, then, we need a different approach. Allowing conjunctive negations means the beta part of the network may not be a tree. However, for any individual join node, if we look at all its ancestors, there is a *total ordering* on them — if J_1 and J_2 are both ancestors of some node, then either J_1 is an ancestor of J_2, or J_2 is an ancestor of J_1. This is in contrast to bilinear networks (Tambe et al., 1992) or more general nonlinear networks (Schor et al., 1986; Ishida, 1988; Lee and Schor, 1992; Hanson, 1993), where the ancestors of a node may be only *partially ordered.*

If every node's ancestors are totally ordered, as is the case in the implementation used in this thesis, then the following approach to relinking works. When relinking a node J_{new} to its alpha memory, we splice it into the *successors* list *immediately in front of its nearest ancestor linked to the same alpha memory* (or at the tail of the list, if it has no such ancestor). Here "nearest" refers to the total ordering. Note that the nearest ancestor may be inside the subnetwork for a conjunctive negation higher up in the network.

With this approach, clause 1 above obviously holds for the nearest J_{old} (call it $J_{nearest}$). Any other ancestor J_{old} of J_{new} must be an ancestor of $J_{nearest}$, since the ancestors are totally ordered. Since the *successors* list is initially appropriately ordered before we splice in J_{new}, $J_{nearest}$ must be on the list before J_{old}. So when we splice in J_{new} before $J_{nearest}$, it will also end up coming before J_{old}. Thus, clause 1 holds. As for clause 2, if J_{old} is a *descendent* of J_{new}, then J_{old} is also a descendent of $J_{nearest}$, so it is initially on the list before $J_{nearest}$; this means that when we splice J_{new} into the list *immediately* in front of $J_{nearest}$, we end up with J_{new} on the list after J_{old}. So clause 2 holds as well.

To implement this, we'll add a field to each join node and each negative node, pointing to its nearest ancestor that happens to use the same alpha memory. This is not really required — we could always find that ancestor for a given node by walking up the network from that node — but that would make the relinking procedure much slower. So we modify the *join-node* and *negative-node* data structures:

```
structure join-node: {revised from version on page 24}
    amem: alpha-memory {points to the alpha memory this node is attached to}
    tests: list of test-at-join-node
    nearest-ancestor-with-same-amem: rete-node
end
```

```
structure negative-node: {revised from version on page 41}
    {just like for a beta memory}
    items: list of token
    {just like for a join node}
    amem: alpha-memory {points to the alpha memory this node is attached to}
    tests: list of test-at-join-node
    nearest-ancestor-with-same-amem: rete-node
end
```

We also modify the Rete network construction routines (Section 2.6) to have them initialize these fields when new nodes are created, and to have them immediately right-unlink newly created nodes whose beta memories are empty. For brevity, these modifications are not shown here; see Appendix A.

Finally, we have the *relink-to-alpha-memory* procedure. Again, this would normally be coded "inline" for efficiency. From a given node, it follows the chain of *nearest-ancestor-with-same-amem* pointers up the network, in order to find the first ancestor on the chain (i.e., the nearest one) that is currently linked to the alpha memory. It then splices the given node into the *successors* list just before that ancestor, or at the tail of the list, if there is no such ancestor. Note that splicing it in at the tail requires the *alpha-memory* data structure (page 32) to have a pointer to the *tail* of the *successors* list, in addition to the usual pointer to the head.

```
procedure relink-to-alpha-memory (node: rete-node)
{version allowing conjunctive negations}
    {follow links up from node, find first ancestor that's linked}
    ancestor ← node.nearest-ancestor-with-same-amem
    while ancestor ≠ nil  and ancestor is right-unlinked[6] do
        ancestor ← ancestor.nearest-ancestor-with-same-amem
    {now splice in the node in the right place}
    if ancestor ≠ nil
        then insert node into the list node.amem.successors immediately before ancestor
        else insert node at the tail of the list node.amem.successors
end
```

Since this procedure contains a **while** loop, one might expect that it would be rather time-consuming, and that right unlinking would therefore add a lot of overhead to the Rete algorithm. The worst-case number of iterations of this loop is $O(C)$, where C is the number of conditions in a production (treating a conjunctive negation containing k conjuncts as if it were k separate conditions). However, the worst case only arises if all the conditions in a production use the same alpha memory; this is very unlikely to happen in practice. Furthermore, in any production not containing a conjunctive negation, at most a single evaluation of the **while** condition will be needed: the first *ancestor* cannot be right-unlinked, since its beta memory cannot be

[6] The implementation used in this thesis determines this by keeping a flag on each join node indicating whether it is currently right-unlinked. An alternative is to check whether the node's beta memory is empty.

empty (the reasoning is similar to that in the Section 4.3.1). Thus, unless productions contain many conjunctive negations and have many conditions using the same alpha memory, very few iterations of the **while** loop will be performed.

4.3.3 Relinking: Fully General

There is a fully general approach to relinking nodes to their alpha memories which does not require ancestors to be totally ordered, and which does not have a potentially large number of iterations of a loop in the worst case. The idea is to partition each *successors* list by *level*. Each join node is assigned a *level number* when the network is constructed; level numbers are assigned so that the number assigned to any given node is lower than the number assigned to any of its descendents. (This can be done via a topological sort.) Let C be the maximum number of conditions in any production. When a WME is added to an alpha memory, the successors at level C are right-activated first, then those at level $C-1$, and so on. To relink a node to its alpha memory, we simply insert it at the head of the list for the appropriate level. The disadvantage of this approach is the extra space required for many more list headers: we now need separate list headers for *successors-at-level-1 successors-at-level-2*, ..., *successors-at-level-C*.

4.4 Right Unlinking: Empirical Results

The graphs in Figure 4.4 show the results of adding right unlinking to Rete. The graph for each testbed system shows two lines: one plots the average match time per change to working memory *in the basic Rete algorithm* as a function of the number of rules, the same data as shown in Figure 3.2 on page 70; the other line plots the average match time *with right unlinking added to Rete*.[7] Note that the scales on the vertical axes differ from one testbed to another.

As the figure shows, the addition of right unlinking to the basic Rete algorithm greatly reduces the match cost in six of the seven testbed systems. (It has only a small effect in Assembler.) Moreover, it allows two of the systems, Dispatcher and SCA-Fixed, to avoid the linear slowdown incurred with the unmodified Rete algorithm. In Merle, SCA-Random, and Sched, there is still a small linear increase in match cost; this effect is not pronounced on the graphs in Figure 4.4, but will become clear in the next chapter. The graphs in Figure 4.4 clearly show a significant linear increase still present in Assembler and Radar. This is because increasing null right activations are not the *only* major cause of Rete's linear slowdown in Assembler and Radar. Eliminating the remaining major cause is the topic of the next chapter.

[7] Note that the figure gives CPU time measurements, not token counts or comparison counts (see Footnote 4 on page 67). Since a null activation does not lead to any tokens being created or any variable binding comparisons being performed, examining token or comparison counts here would be misleading — such counts would suggest that right unlinking had no effect on the match algorithm's performance, whereas Figure 4.4 demonstrates otherwise.

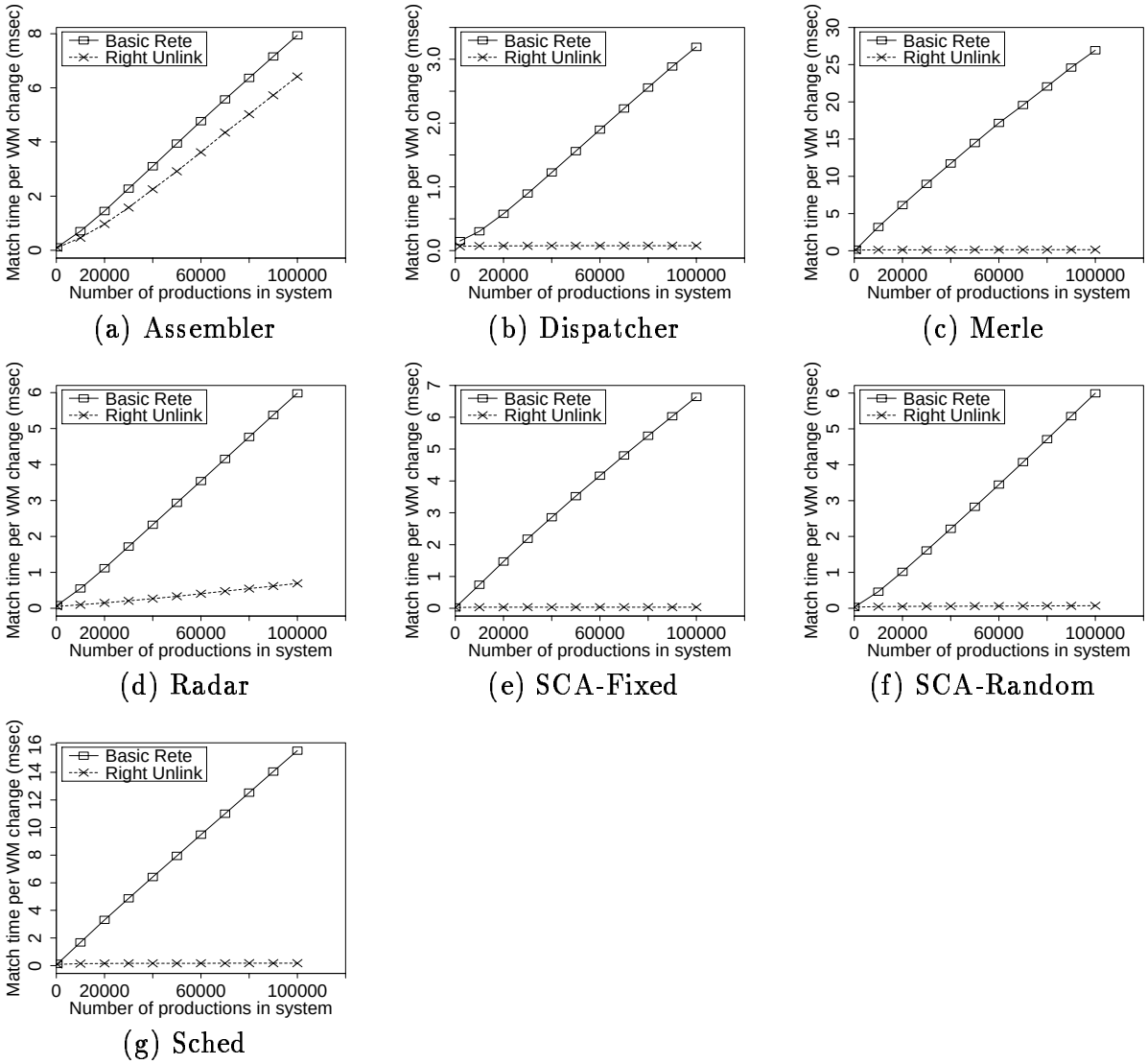


Figure 4.4: Match cost with right unlinking added to the basic Rete algorithm.

Chapter 5

Adding Left Unlinking to Rete

At the beginning of the previous chapter, we mentioned two possible causes of the linear slow-down observed in the basic Rete algorithm in all the testbed systems. We then introduced right unlinking as a way to reduce or eliminate one of these — a linear increase in the number of right activations. The other one — a linear increase in the number of left activations — is the subject of this chapter. Due to the sharing of nodes in the Rete network, a WME that affects a large number of rules may only trigger one right activation — namely, the right activation of a shared join node near the top of the network. However, dataflow may propagate down the network from that node, causing left activations of many other nodes, and the number of such left activations can increase linearly in the number of rules in the system.

After we look at a concrete example of this increasing number of left activations, we explain in Section 5.1 why most of these activations are useless work. Section 5.2 explains the basic idea of *left unlinking*, our extension to Rete which avoids most of these activations. Section 5.3 discusses why left unlinking and right unlinking interfere with each other, and shows how to combine the two so as to minimize this interference. Section 5.4 presents the implementation of this combination of left and right unlinking, which yields the Rete/UL match algorithm. Sections 5.5 and 5.6 present theoretical and empirical results. Section 5.7 shows how to generalize unlinking to systems with non-binary join nodes. Finally, Section 5.8 discusses some alternatives to unlinking.

As in the previous chapter, we begin our discussion with a concrete example — in this case, the Assembler system. As mentioned in Section 3.1.1, this system is a cognitive model of a person assembling printed circuit boards, e.g., inserting resistors into the appropriate slots. Most of the rules it learns are specific to the particular slot on the board being dealt with at the moment. The first few conditions in these rules are always the same, but the next condition is different in each rule. As illustrated in Figure 5.1, this leads to a large fan-out from one beta memory. (As with our SCA-Fixed example in the previous chapter, this figure is simplified for expository purposes.) The first few conditions in all the rules share the same nodes, but at this point, sharing is no longer possible because each rule tests for a *different* slot. As the system deals with more and more slots, it learns more and more rules, and the fan-out increases linearly in the total number of rules.

Now, whenever all of the first few conditions of these rules are true, the dataflow in the Rete

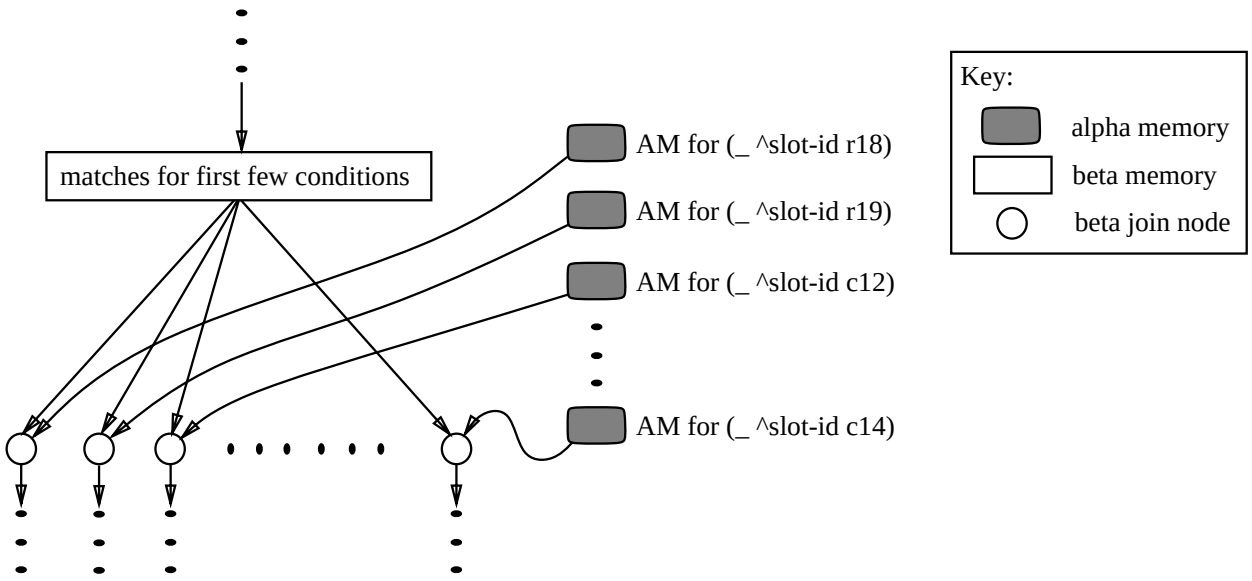


Figure 5.1: Part of the Rete network for Assembler.

network reaches this beta memory and a token is stored there. This token is then propagated to all the memory's child join nodes, left activating each one. Since the number of such nodes is increasing linearly in the number of rules, the work done here is also linearly increasing. This causes the Rete algorithm to slow down linearly in the number of rules.

Note that as was the case with SCA-Fixed, we cannot eliminate this problem by changing the ordering of the conditions here. If we reorder the conditions so this fan-out point occurs lower in the network, it may alleviate the problem somewhat, by reducing the number of tokens that reach the fan-out point, but the slowdown will still be linear, since the work performed each time a token reaches the fan-out point will still be linear. Moving the fan-out point higher in the network would aggravate the problem, except in the special case where the fan-out occurs directly from the top node of the beta network; in this one case, the problem would be avoided because tokens are never added to or removed from the dummy top node. However, there are three drawbacks to this: it wastes a lot of space, since it precludes a lot of potential sharing; it is unclear how to arrange for conditions to be ordered this way automatically; and it cannot be used if rules contain *more than one* problematic condition like this — e.g., if the rules in Assembler were specific to *pairs* of slots, so each had *two* slot-id-testing conditions — since only one of them could be placed directly below the dummy top node.

5.1 Null Left Activations

In the SCA-Fixed example of the previous chapter, we observed that most of the linearly increasing number of right activations were wasted work — only one of them would "succeed," while the others were *null* right activations. A similar phenomenon occurs with the left activations in Assembler. Since the system is only focusing its attention on a few slots at a time, most of the

join nodes have empty alpha memories. Their activations are therefore *null left activations*, and no new matches result from them. Although each individual null left activation takes very little time to execute, the number of null left activations per change to working memory is linearly increasing, and so this can grow to dominate the overall match cost.

This problem arises in other systems in addition to Assembler. The large fan-out from beta memories can arise in any system where the domain has some feature with a large number of possible values, and the learned rules are specific to particular values of that feature. For instance, in a robot domain, if the appropriate action to be taken by the robot depends on the exact current room temperature, it might learn a set of rules where each one checks for a different current temperature. In a system with a simple episodic memory, learned rules implementing that memory might contain different timestamps in their conditions. In cases like these, learned rules will often share nodes in the Rete network for their *early* conditions, up to but not including the conditions testing the feature in question. If this feature can have only one value at a time, then most of the rules will fail to match at this condition, so there will be a large number of null left activations.

Turning to our testbed systems now, Figure 5.2 shows, for each testbed system, the average number of null left activations incurred by join nodes per change to working memory, plotted as a function of the number of rules in the system. In two of the systems, Assembler and Radar, the number of null left activations per change to working memory increases quite significantly and linearly in the number of rules. (It increases by $\sim$98 and $\sim$16 per 10,000 rules in these two systems, respectively). In two others, Sched and Merle, it increases slightly and linearly (by $\sim$2 and $\sim$1 per 10,000 rules). In the other three systems, it does increase somewhat, but only very slightly (by much less than 1 per 10,000 rules). (The points for these systems are very close to the horizontal axis in Figure 5.2.)

5.2 Left Unlinking: Basic Idea

In this section, we introduce an optimization for the Rete algorithm which avoids all null left activations. (Some other possible ways to avoid null left activations are discussed in Section 5.8.) With this optimization incorporated into the matcher, we will no longer see the linearly increasing null left activations shown in Figure 5.2. As in the previous chapter, by saying we avoid a null left activation, we do not mean just that we avoid executing a procedure call — we mean we avoid *executing any CPU instructions* for it — i.e., the beta memory does not even "look at" the join node or its alpha memory.

Our optimization, called *left unlinking*, is symmetric to right unlinking: whereas with right unlinking, a join node is spliced out of its alpha memory's list of successors whenever its beta memory is empty, with left unlinking, a join node is spliced out of its beta memory's list of children whenever its alpha memory is empty. This is illustrated in Figure 5.3: whenever the alpha memory is empty, we cut the link (the dataflow path) from the beta memory (the one on the left) to the join node. Then whenever a token is added to the beta memory, the join node will not be left-activated; in fact, the match algorithm will perform *no CPU instructions*

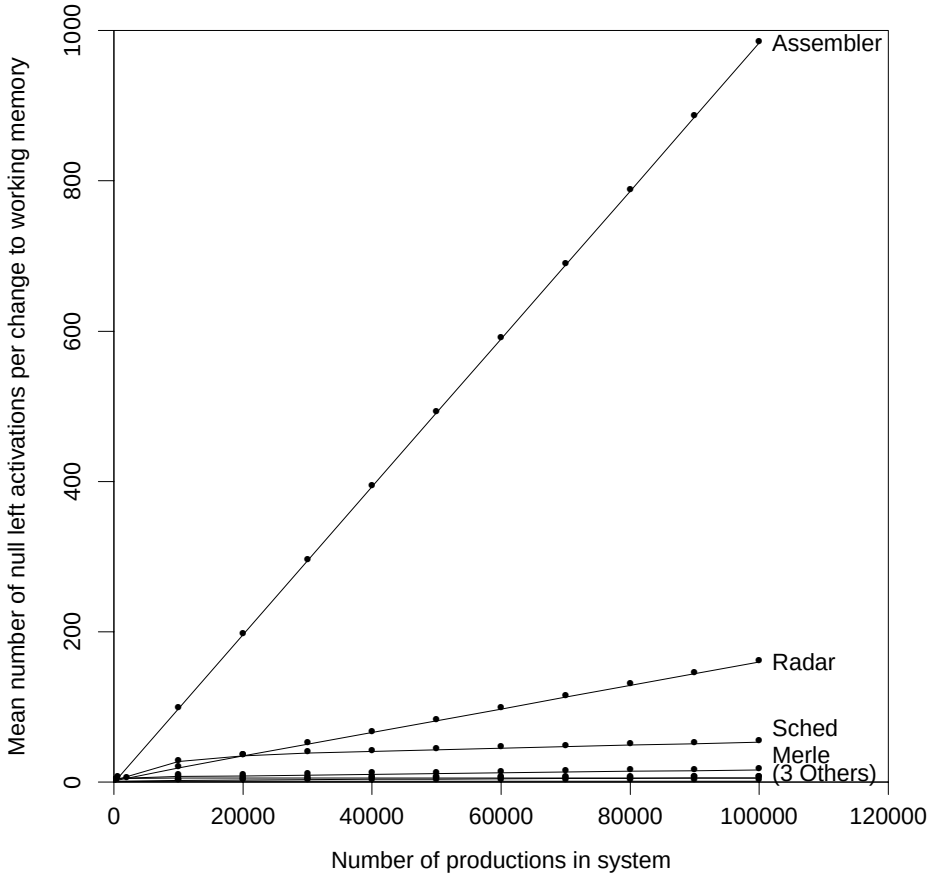


Figure 5.2: Number of null left activations per change to working memory.

at all pertaining to that join node, since the *beta-memory-left-activation* procedure (page 30) just iterates over the beta memory's *children* list, and this join node will not be in that list.

Whenever alpha memories change from nonempty to empty or vice-versa in a running system, we will dynamically splice join nodes out of and back into the *children* lists on beta memories. So in the Assembler system, at any given time, most of the join nodes would be unlinked from the beta memory shown in Figure 5.1, and hence they would not be activated whenever the first few conditions in the rules are true. As was the case with right unlinking, the only activations we are skipping are null activations, which would not yield a match anyway, so this optimization does not affect the set of complete production matches that will be found; the semantics of the algorithm (viewed as a "black box") are unchanged.

Just as right unlinking can be viewed as a way to reduce the potentially large fan-out from alpha memories, left unlinking can be viewed as a way to reduce the fan-out from beta memories. It is expected to be an important optimization in Assembler and other systems where some beta memories have large fan-outs. In addition, we will see in Section 5.6 that left unlinking can also be beneficial even in systems where the fan-out isn't especially large and null left activations

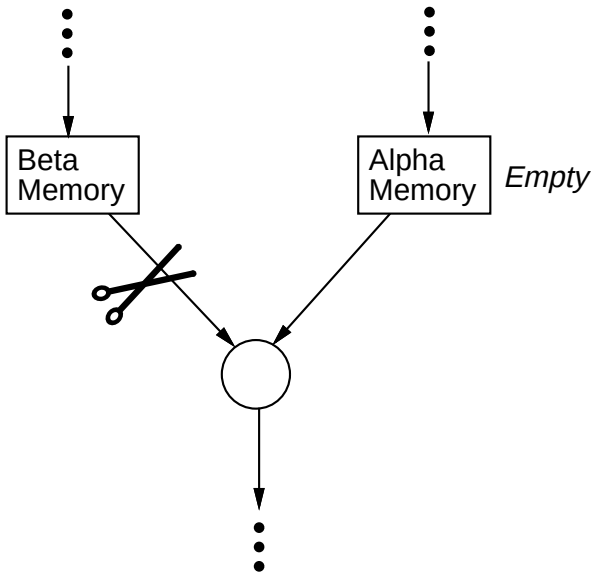


Figure 5.3: Unlinking a join node from its associated beta memory.

don't dominate the overall match cost.

In the previous chapter, we did right unlinking not only on (positive) join nodes, but also on negative nodes. Unfortunately, left unlinking cannot be used on negative nodes. With an ordinary join node, if the alpha memory is empty, then the (positive) condition is false, hence we do not want the dataflow to propagate further down the network. With a negative node, though, if the alpha memory is empty, then the (negative) condition is true, hence we *do* want the dataflow to propagate further down the network. Left-unlinking a negative node would prevent this, thus destroying the correctness of the match algorithm. Hence, negative nodes cannot be left-unlinked. Fortunately, negated conditions are typically much less common than positive conditions (Gupta, 1987).

We will give pseudocode for the implementation of left unlinking in Section 5.4 below. However, we must first discuss how to combine left unlinking with right unlinking. We clearly need to use both in systems such as Assembler and Radar, where *both* null right activations and null left activations are increasing linearly in the number of rules, as shown in Figures 4.2 and 5.2, respectively. Combining left and right unlinking is not entirely straightforward, as we discuss in the next section.

5.3 Interference Between Left and Right Unlinking

Since right unlinking avoids all null right activations, and left unlinking avoids all null left activations, we would like to combine both in the same system and avoid all null activations entirely. Unfortunately, this is not possible, because the two optimizations can interfere with each other. The problem arises when a join node's alpha and beta memories are *both* empty, as illustrated in Figure 5.4. Left unlinking dictates that the node be unlinked from its beta

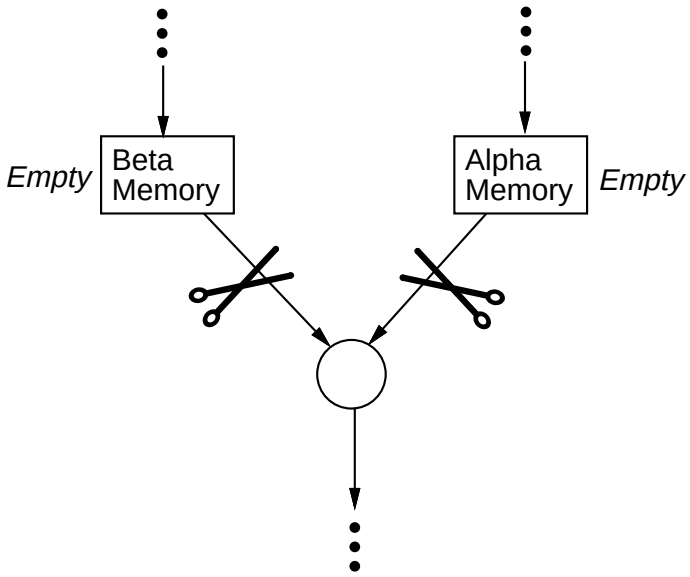


Figure 5.4: Unlinking a join node from both its alpha memory and its beta memory at the same time, destroying the correctness of the match algorithm.

memory. Right unlinking dictates that the node be unlinked from its alpha memory. If we do both, then the join node will be completely cut off from the rest of the network — *all* the dataflow links to it will have been removed — so it will never be activated again, even when it should be.[1] The correctness of the match algorithm would be destroyed. *To ensure correctness, when a join node's memories are both empty, we can use either left unlinking or right unlinking, but not both.* But which one? If we use left but not right unlinking in this situation, then we can still suffer null right activations. If we use right but not left unlinking, then we can still suffer null left activations. Thus, no scheme for combining left and right unlinking can avoid *all* null activations.

If both memories are empty, which one should the join node be unlinked from? A number of possible heuristics come to mind. We might left unlink nodes whose beta memories have sufficiently large fan-out, as in Figure 5.1. Or we might do a trial run of the system in which we record how many null left and right activations each node incurs; then on later runs, we would unlink from the side that incurred more null activations in the trial run.

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

Definition: *In the* **first-empty-dominates** *scheme for combining left and right unlinking, a join node J with alpha memory A and beta memory B is unlinked as follows. (1) If A is empty*

[1] Assuming it does not somehow get relinked at some later time. Of course, we could have a piece of code somewhere that periodically checked all the nodes currently unlinked from a certain memory, and decided to relink some of them. But this would require executing a few CPU instructions for each of the unlinked nodes, which is precisely what we *don't* want to do, because the match cost would again increase linearly.

but B is nonempty, it is linked to A and unlinked from B. (2) If B is empty but A is nonempty, it is linked to B and unlinked from A. (3) If A and B are both empty, it is (i.e., remains) linked to whichever memory became empty earlier, and unlinked from the other memory.

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

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

It turns out that first-empty-dominates is the optimal scheme for combining left and right unlinking: except for the possible one-activation initialization cost, it minimizes the number of null activations. Thus, this simple scheme yields the minimal interference between left and right unlinking. We formalize this result in Section 5.5 below, but first we give pseudocode for this scheme in the next section.

5.4 Left Unlinking and Rete/UL: Implementation

We now add left unlinking to Rete, combining it with right unlinking using the first-empty-dominates scheme. We call the resulting match algorithm *Rete/UL*. We must make several modifications to the Rete interpreter routines. There are four main things to consider:

- What extra code do we need when an alpha memory changes from nonempty to empty?
- What extra code do we need when a beta memory changes from nonempty to empty?
- What extra code do we need when an alpha memory changes from empty to nonempty?
- What extra code do we need when a beta memory changes from empty to nonempty?

For the first question, whenever an alpha memory changes from nonempty to empty, we need to left-unlink each join node on its *successors* list. We add code to the *remove-wme* procedure to do this. Note that some join nodes using this alpha memory may not currently be on the

successors list — some may be right-unlinked because their beta memories are already empty. We leave these join nodes alone, letting the empty beta memory "dominate" the now-empty alpha memory.

```
procedure remove-wme (w: WME) {revised from version on page 44}
    for each item in w.alpha-mem-items do
        remove item from the list item.amem.items
        if item.amem.items = nil then {alpha memory just became empty}
            for each node in item.amem.successors do
                if node is a join node then {don't left-unlink negative nodes}
                    remove node from the list node.parent.children
        deallocate memory for item
    while w.tokens ≠ nil do
        delete-token-and-descendents (the first item on w.tokens)
    for each jr in w.negative-join-results do
        remove jr from the list jr.owner.join-results
        if jr.owner.join-results=nil then
            for each child in jr.owner.node.children do
                left-activation (child, jr.owner, nil )
        deallocate memory for jr
end
```

For the second question, whenever a beta memory changes from nonempty to empty, we need to right-unlink each join node on its *children* list. This modification was already made in the previous chapter for right unlinking (see the revised *delete-token-and-descendents* procedure on page 87); nothing further needs to be done here. Note, however, that now that we are combining right unlinking with left unlinking, a beta memory's *children* list may not contain all the join nodes using that beta memory — some of those join nodes may currently be left-unlinked because their alpha memories are already empty. We leave these join nodes alone, letting the empty alpha memory "dominate" the now-empty beta memory. (The code needs no modification, since it simply loops over the memory's *children* list.)

Turning to the third question, whenever an alpha memory changes from empty to nonempty, we need to relink each join node on its *successors* list to its beta memory. Also, note that during the time period these join nodes were unlinked from their beta memories, some of those beta memories may have become empty — in which case, some of these join nodes now have empty beta memories but a nonempty (shared) alpha memory. Consequently, in addition to relinking these join nodes to their beta memories, we unlink the ones with empty beta memories from the alpha memory. To do this, we add code to the *join-node-right-activation* procedure, and also create a straightforward *relink-to-beta-memory* procedure.[2]

[2] An alternative is to add the code to the *alpha-memory-activation* procedure, which calls the *join-node-right-activation* procedure. The drawback to placing the code in the *join-node-right-activation* procedure is that it makes the "alpha memory just became nonempty" condition often get tested multiple times (once per join node activation) rather than just once (for the alpha memory activation). The drawback to placing the code in the

```
procedure join-node-right-activation (node: join-node, w: WME)
{revised from version on page 24}
    if node.amem just became nonempty[3] then
        relink-to-beta-memory (node)
        if node.parent.items = nil then
            remove node from the list node.amem.successors
    for each t in node.parent.items do {"parent" is the beta memory node}
        if perform-join-tests (node.tests, t, w) then
            for each child in node.children do left-activation (child, t, w)
end

procedure relink-to-beta-memory (node: join-node)
    insert node at the head of the list node.parent.children
end
```

Finally, for the fourth question, whenever a beta memory changes from empty to nonempty, we need to relink each join node on its *children* list to its alpha memory. This modification was already made in the previous chapter for right unlinking (see the revised *join-node-left-activation* procedure on page 88). However, one further change must be made now. During the time period these join nodes were unlinked from their alpha memories, some of those alpha memories may have become empty — in which case, some of these join nodes now have empty alpha memories but a nonempty (shared) beta memory. Consequently, in addition to relinking these join nodes to their alpha memories, we unlink the ones with empty alpha memories from the beta memory. To do this, we add some more code to the *join-node-left-activation* procedure. This is symmetrical to the code we added to the *join-node-right-activation* procedure above.

```
procedure join-node-left-activation (node: join-node, t: token)
{revised from version on page 88}
    if node.parent just became nonempty then
        relink-to-alpha-memory (node)
        if node.amem.items = nil then
            remove node from the list node.parent.children
    for each item in node.amem.items do
        if perform-join-tests (node.tests, t, item.wme) then
            for each child in node.children do left-activation (child, t, item.wme)
end
```

alpha-memory-activation procedure is that an extra conditional statement must be executed there, once for each successor node, because the relinking code is specific to the type of successor node — join nodes get relinked to their beta memories, while negative nodes don't need to be relinked since we never left-unlink them to begin with. With the relinking code in the *join-node-right-activation* procedure, this extra conditional is not needed, since it essentially gets "folded into" the jumptable or `switch` statement that dispatches the appropriate right-activation procedure.

[3] There are several ways of determining this. We can have the *alpha-memory-activation* procedure pass the *join-node-right-activation* procedure a flag indicating this. Alternatively, we can just check whether the number of items in the alpha memory is exactly one. Another approach (taken by the implementation used in this thesis) is to keep a flag on each join node indicating whether it is currently left-unlinked.

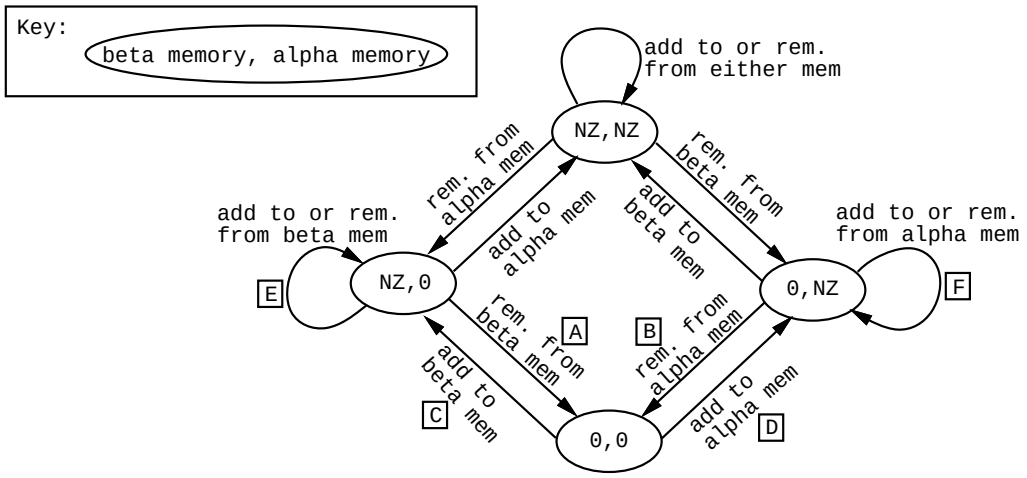


Figure 5.5: Possible states of a join node and its alpha and beta memories.

We also modify the Rete network construction routines (Section 2.6) to have them immediately unlink newly-created nodes if at least one of their memories is empty. For brevity, these modifications are not shown here; see Appendix A.

5.5 Interference: Theoretical Results

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

Theorem 5.1 (Optimality of First-Empty-Dominates) *Any scheme for combining left and right unlinking must incur at least as many null activations of each join node as first-empty-dominates incurs, ignoring the possible one-activation initialization cost.*

Proof: For any given join node, Figure 5.5 shows the four states its alpha and beta memories can be in: the number of items in each memory can be 0 or nonzero (NZ). The figure also shows all the possible state transitions that can occur on changes to the alpha and beta memories. All the transitions into and out of (NZ,NZ) are *non-null* activations. Unlinking never avoids non-null activations, so the join node will incur one activation on each of these transitions no matter what unlinking scheme we use.

The remaining transitions (labeled A–F) are null activations if no unlinking is done; but the join node will not be activated on these if it is unlinked from the appropriate memory. Under first-empty-dominates, the join node is always unlinked from its beta memory when in state (NZ,0). This means it will not incur a null activation on transition A or E. Similarly, it is always unlinked from its alpha memory when in state (0,NZ), so it will not incur a null activation on transition B or F. This leaves just C and D to consider. In state (0,0), the

join node is unlinked from its beta memory if its alpha memory became empty *before* its beta memory did — i.e., if it arrived at (0,0) via transition A — and unlinked from its alpha memory otherwise — i.e., if it arrived via B. (This ignores the case where the node *starts* at (0,0).) This means a null activation is incurred by first-empty-dominates only when D follows A or when C follows B.

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

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

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

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

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

Without any unlinking, or with left or right unlinking but not both, ν is $O(n)$ in the worst case. This is proved in the following theorem:

Theorem 5.2 (Worst Case Null Activations Without Unlinking) *Over any finite sequence of changes to working memory, $\nu \leq 1 + n$, where n is the number of join nodes in the network.*

Proof: In the dataflow operation of the Rete algorithm, join node activations can only be triggered by activations of alpha memories or beta memories. Moreover, the number of join node activations (null or non-null) triggered by a single alpha or beta memory activation is at most the number of successors of that alpha memory or children of that beta memory; this is

in turn at most n. Thus, if a sequence of changes to working memory triggers a alpha memory activations and b beta memory activations, then it can trigger at most $n(a+b)$ null join node activations. It follows that $\nu \leq 1 + \frac{n(a+b)}{a+b} = 1 + n$. □

This worst-case result is not merely *theoretical*: ν often turns out to be linear in n *in practice*, too. This is illustrated by our SCA-Fixed and Assembler examples (see Figure 4.1 on page 82 and Figure 5.1 on page 96, respectively) and clearly demonstrated by the empirical results on right unlinking (Section 4.4) and left unlinking (Section 5.6 below). However, the first-empty-dominates combination of left and right unlinking reduces the theoretical worst case to $O(\sqrt{n})$, as we prove below.

Before we get to the worst-case analysis, two assumptions must be made. First, we will ignore the possible initial null activation of each join node which starts out with its alpha and beta memories both empty — of course, this initialization cost is negligible in the long run. Second, we will assume that no two join nodes use both the same alpha memory *and* the same beta memory. This is the normal situation in practice.[4]

It turns out that with the first-empty-dominates combination of left and right unlinking, $\nu \leq 1 + \frac{1}{2}\sqrt{n}$. Before we present a proof of this, it is instructive to see how the worst case can arise. Consider how we might design a system so as to maximize ν, by causing a large number of null activations with only a few other activations. First, we want all join node activations to be null, since non-null join node activations just decrease ν. The denominator in the above definition of ν then comprises just alpha and beta memory activations. So we want each alpha or beta memory activation to trigger as many null join node activations as possible. The key to this, of course, is having large fan-outs from alpha and beta memories, since the number of null activations triggered by a single memory activation is at most the fan-out from that memory. Suppose we have a single alpha memory with a large fan-out — an alpha memory with b successor join nodes, each of which uses a different beta memory (as required by our assumption above). Recall that with first-empty-dominates, null activations occur when join nodes take transition A and then D (see Figure 5.5), or transition B and then C. We can trigger b null activations, one of each join node, with a single activation of the alpha memory (adding a WME to it), by causing each join node to take transition D. However, this requires each join node to have just taken transition A; causing these transitions requires removing a token from each of the b beta memories. Thus, we trigger b null activations with a total of $b+1$ alpha and beta memory activations, and $\nu = 1 + \frac{b}{b+1} < 2$. The problem is that although the fan-out from the alpha memory is large, the fan-out from each beta memory is only one, so it requires a large number of beta memory activations to cause all the A transitions. If instead we have a single beta memory with a large fan-out, but each of its child join nodes uses a different alpha memory, a similar analysis holds.

This suggests that to get bad case behavior, we need a large number of join nodes for which both (1) the beta memories they use have large fan-outs, and (2) the alpha memories they use

[4]It is possible for two or more join nodes to use the same alpha and the same beta memories — if there are productions whose conditions test exactly the same constants in the same places, but have different inter-condition variable binding consistency checks — but this is not very common in practice. Even in theory, the number of join nodes using the same pair of memories can be bounded *independent* of the number of productions in the system. If this bound is c, the worst-case bound in Theorem 5.4 becomes $1 + \frac{c}{2}\sqrt{N}$.

have large fan-outs. Importantly, this situation does not arise in our testbed systems. In the SCA-Fixed example (see Figure 4.1 on page 82), the join nodes at the bottom of the network use alpha memories with large fan-outs (10^5), but the beta memories they use have only relatively small fan-outs (10). In the Assembler example (see Figure 5.1 on page 96), the join nodes use a beta memory with a large fan-out, but the alpha memories they use have only relatively small fan-outs.

What happens if conditions (1) and (2) both hold? Suppose we have a system with a alpha memories and b beta memories, and suppose that all possible join nodes are present — i.e, for every alpha memory, for every beta memory, there is a join node using that alpha memory and that beta memory, for a total of $n = ab$ join nodes. Each alpha memory has fan-out b and each beta memory has fan-out a. We can trigger ab null activations, one of each join node, by causing each one to take transition A and then D. This requires removing a token from each beta memory and then adding a WME to each alpha memory, for a total of $a + b$ alpha and beta memory activations. We then have $\nu = 1 + \frac{ab}{a+b}$. We can also trigger ab null activations by causing each join node to take transition B and then C; a similar analysis yields the same value of ν in this case. This expression attains its maximum value when $a = b$; when this happens, we have $n = a^2$ and $\nu = 1 + \frac{a^2}{2a} = 1 + \frac{a}{2} = 1 + \frac{1}{2}\sqrt{n}$.

How might this situation arise in a real system? Consider once again the situation in our SCA-Fixed example of Figure 4.1. The lowermost join nodes use alpha memories with large fan-outs (10^5), but the beta memories they use have only relatively small fan-outs (10). Generalizing this example, suppose we have f features, each with v possible values, and a total of v^f rules. (In our SCA-Fixed example, $f = 6$ and $v = 10$.) The lowermost join nodes then use alpha memories with fan-outs of v^{f-1} and beta memories with fan-outs of v. To make these fan-outs equal, we take $f = 2$. So to get this bad behavior in a real system, we would need a large number of rules that test different values of exactly two features in working memory, both of which have a large number of possible values.

The analysis just presented made several assumptions not likely to hold in practice: that there are no non-null activations of join nodes; that alpha and beta memory activations always occur in just the right sequence so as to cause B-C and A-D transition sequences; that all join nodes incur the same number of null activations; that there is a "complete set" of join nodes, one for each alpha memory-beta memory pair. We now generalize our analysis. That $1 + \frac{1}{2}\sqrt{n}$ is actually the worst possible value of ν in the general case is proved in Theorem 5.4 below (which actually proves a slightly stronger statement). To prove the theorem, we will need the following lemma.

Lemma 5.3 *Let (x_{ij}) be an $A \times B$ matrix with all $x_{ij} \geq 0$. Let S be the sum of all entries, R be the sum of the row maxima, C be the sum of the column maxima, and N be the number of nonzero entries. Then $S^2 \leq NRC$.*

Proof: Let r_i denote the maximum of row i, and c_j denote the maximum of column j. The Cauchy-Schwarz inequality from linear algebra tells us that for vectors $\mathbf{u}$ and $\mathbf{v}$ of real numbers, $(\mathbf{u}\cdot\mathbf{v})^2 \leq (\mathbf{u}\cdot\mathbf{u})(\mathbf{v}\cdot\mathbf{v})$. Take $\mathbf{u}$ to be a vector of N 1's, and $\mathbf{v}$ to be a vector containing the N

nonzero entries in the matrix. Then $\mathbf{u}\cdot\mathbf{v} = S$, $\mathbf{u}\cdot\mathbf{u} = N$, and the Cauchy-Schwarz inequality gives us the first step below:

$$S^2 \leq N \sum_{i=1}^{A}\sum_{j=1}^{B} x_{ij}^2 \leq N \sum_{i=1}^{A}\sum_{j=1}^{B} r_i c_j = N \left(\sum_{i=1}^{A} r_i\right)\left(\sum_{j=1}^{B} c_j\right) = NRC.$$

□

Theorem 5.4 (Worst Case Null Activations in First-Empty-Dominates) *Consider the first-empty-dominates combination of left and right unlinking, ignoring the one initial null activation of each join node that starts with its alpha and beta memories both empty. Assume that for each pair of alpha and beta memories, there is at most one join node using that pair. Over any finite sequence of changes to working memory, $\nu \leq 1 + \frac{1}{2}\sqrt{N}$, where N is the number of join nodes incurring at least one null activation over that sequence.*

Proof: Let the alpha memories in the network be numbered from 1 to A, where A is the total number of alpha memories, and let a_i be the number of activations of (i.e., items added to or removed from) alpha memory i. Likewise, let the beta memories be numbered from 1 to B, and let b_j be the number of activations of beta memory j. If there is a join node using alpha memory i and beta memory j, then let x_{ij} be the number of null activations it incurs over the given sequence of changes to working memory. If there is no join node using alpha memory i and beta memory j, then let x_{ij} be zero. Finally, let S denote the total number of null activations of all join nodes: $S \stackrel{\text{def}}{=} \sum_{i=1}^{A}\sum_{j=1}^{B} x_{ij}$. (Note that this S is the same as in the statement of Lemma 5.3. We will later apply the lemma to this matrix (x_{ij}).)

Consider x_{ij}, the number of null activations incurred by the join node (call it J) testing alpha memory i and beta memory j. Under first-empty-dominates, and ignoring the one initial null activation if i and j are both initially empty, each null activation of J requires one change to alpha memory i and one change to beta memory j (see Figure 5.5 — each null activation requires a sequence of two transitions: one change to the alpha and one to the beta memory). Thus, for all i and j, $a_i \geq x_{ij}$ and $b_j \geq x_{ij}$.

It follows that for all i, $a_i \geq \max_{j=1}^{B} x_{ij}$; and for all j, $b_j \geq \max_{i=1}^{A} x_{ij}$. Thus, the total number of activations of all alpha memories is at least $R \stackrel{\text{def}}{=} \sum_{i=1}^{A} \max_{j=1}^{B} x_{ij}$, and the total number of activations of all beta memories is at least $C \stackrel{\text{def}}{=} \sum_{j=1}^{B} \max_{i=1}^{A} x_{ij}$.

Now consider ν. By the definition of ν, and since R and C are lower bounds on the number of memory activations, we have $\nu \leq 1 + \frac{S}{R+C}$. Then

$$\nu \leq 1 + \sqrt{\frac{S^2}{(R+C)^2}} = 1 + \sqrt{\frac{S^2}{(R-C)^2 + 4RC}} \leq 1 + \sqrt{\frac{S^2}{4RC}} = 1 + \frac{1}{2}\sqrt{\frac{S^2}{RC}}$$

Applying Lemma 5.3, we get $\nu \leq 1 + \frac{1}{2}\sqrt{N}$. □

	Join node activations per change to working memory:				
		Null, when using this type of unlinking:			
System	Non-null	None	Left only	Right only	Both
Assembler	13.2	4,487.8	3,504.5	983.3	0.11
Dispatcher	19.8	2,050.2	2,044.7	5.5	0.17
Merle	17.2	16,724.6	16,708.2	16.3	0.24
Radar	9.6	2,885.4	2,725.4	160.0	0.10
SCA-Fixed	7.4	3,820.3	3,819.0	1.3	0.21
SCA-Random	12.4	3,219.8	3,214.1	5.7	2.69
Sched	19.7	10,862.3	10,809.3	53.0	0.17
Average	14.2	6,292.9	6,117.9	175.0	0.53

Table 5.1: Average number of join node activations per change to working memory with 100,000 rules on each testbed, using different versions of the matcher.

5.6 Left Unlinking and Rete/UL: Empirical Results

In this section we present empirical results from Rete/UL on our testbed systems. There are two major questions here. First, how bad is the interference between left and right unlinking? Second, how effective is the combination of left and right unlinking in avoiding a linear increase in the overall match cost?

For the first question, as mentioned above, in order to get a large amount of interference between left and right unlinking, we need a large number of join nodes for which both (1) the beta memories they use have large fan-outs, and (2) the alpha memories they use have large fan-outs; this situation does not arise in our testbed systems. Table 5.1 shows, for each system, the average number of null and non-null join node activations per working memory change, when there are 100,000 rules in the system.[5] For null activations, four different numbers are given, corresponding to four different match algorithms: the basic Rete algorithm without any unlinking, Rete with left but not right unlinking, Rete with right but not left unlinking, and Rete/UL. The table shows that without any unlinking, or with left unlinking only, the matcher is essentially swamped by null activations in all the systems. With right unlinking but no left unlinking, there are still a large number of null (left) activations in both Assembler and Radar, a fair number in Sched, and a few in Merle. Finally, with left and right unlinking combined in Rete/UL, the number of null activations is very small in all the systems. Thus, the interference between left and right unlinking turns out to be insignificant in practice, at least for this diverse set of systems.

We now turn to the second question — how effective is Rete/UL in avoiding a linear increase in the overall match cost? The graphs in Figure 5.6 show the results. The graph for each testbed system shows four lines, each plotting the average match time per change to working memory as a function of the number of rules. The four lines correspond to four different match algorithms: the basic Rete algorithm (this is the same data as in Figure 3.2 on page 70), Rete plus right

[5] See (Doorenbos, 1994) for a similar table giving the activation counts averaged over the course of the whole run of each system, from just its initial rules out to beyond 100,000 rules.

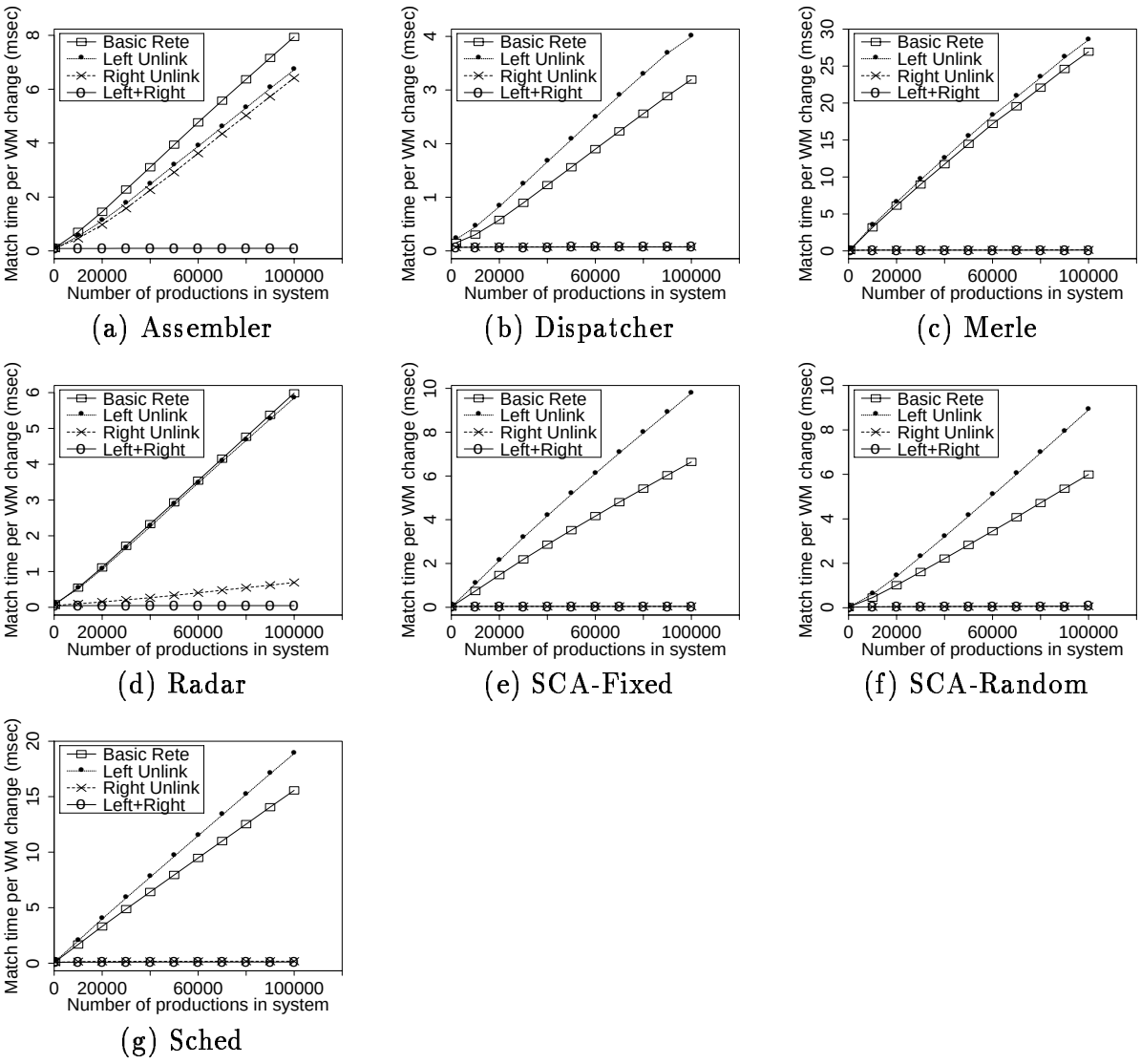


Figure 5.6: Match cost with left and right unlinking added to the basic Rete algorithm.

unlinking but not left unlinking (this is the same data as in Figure 4.4 on page 93), Rete plus left unlinking but not right unlinking, and Rete/UL. Note that the scales on the vertical axes differ from one testbed to another.

The figure shows that with just the basic Rete algorithm, or with left but not right unlinking, all the systems suffer a major linear slowdown as the system learns more and more rules. Incorporating left unlinking into the matcher without using right unlinking actually slows down the matcher in all the testbeds except Assembler and Radar. This is because left unlinking adds a small overhead to various node activation procedures. This overhead slows down the matcher by a small factor, and if this small factor is not outweighed by a large factor reduction in the number of activations, a slowdown can result. In most of the testbed systems, when left unlinking is used *alone*, null right activations are so numerous that avoiding null left activations

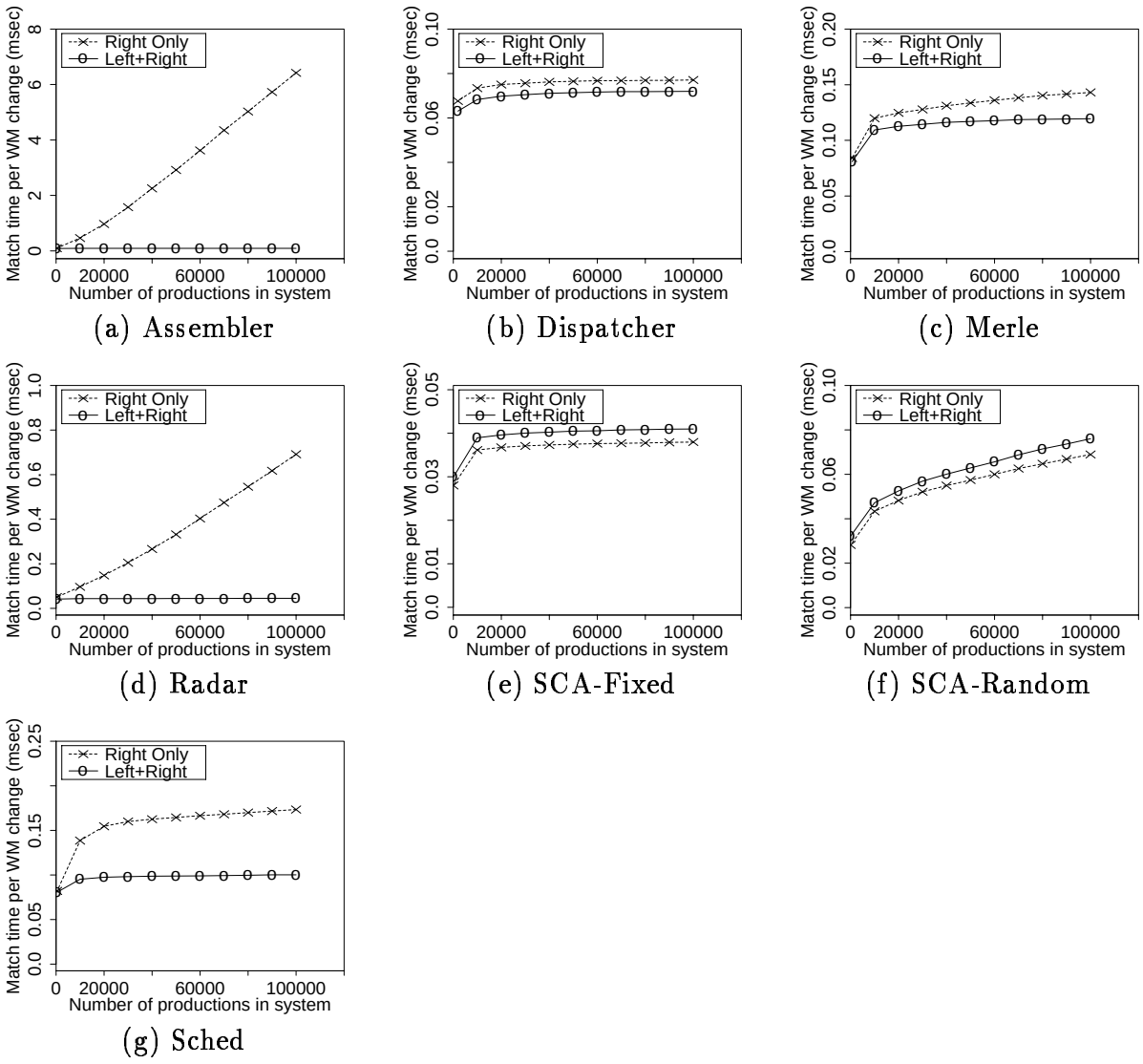


Figure 5.7: Match cost with right unlinking only, and with right plus left unlinking.

only reduces the total number of activations by a very small factor. In the Assembler and Radar systems, though, null left activations constitute a sufficient fraction of the activations in the basic Rete algorithm that avoiding them using left unlinking does pay off.

For most of the systems in Figure 5.6, the lines for Rete plus right unlinking and Rete plus both right and left unlinking are quite close together. Figure 5.7 shows just these two lines for each system, with the scale on the vertical axis changed to make them clear. As the figure shows, the addition of left unlinking (in combination with right unlinking) enables both Assembler and Radar to avoid a significant linear slowdown as the number of rules increases, and Sched and Merle to avoid a small linear slowdown. In the other three systems, the addition of left unlinking has only a minor effect: it reduces the match cost slightly (7%) in Dispatcher and increases it slightly (8% and 10%, respectively) in SCA-Fixed and SCA-Random, because of the overhead

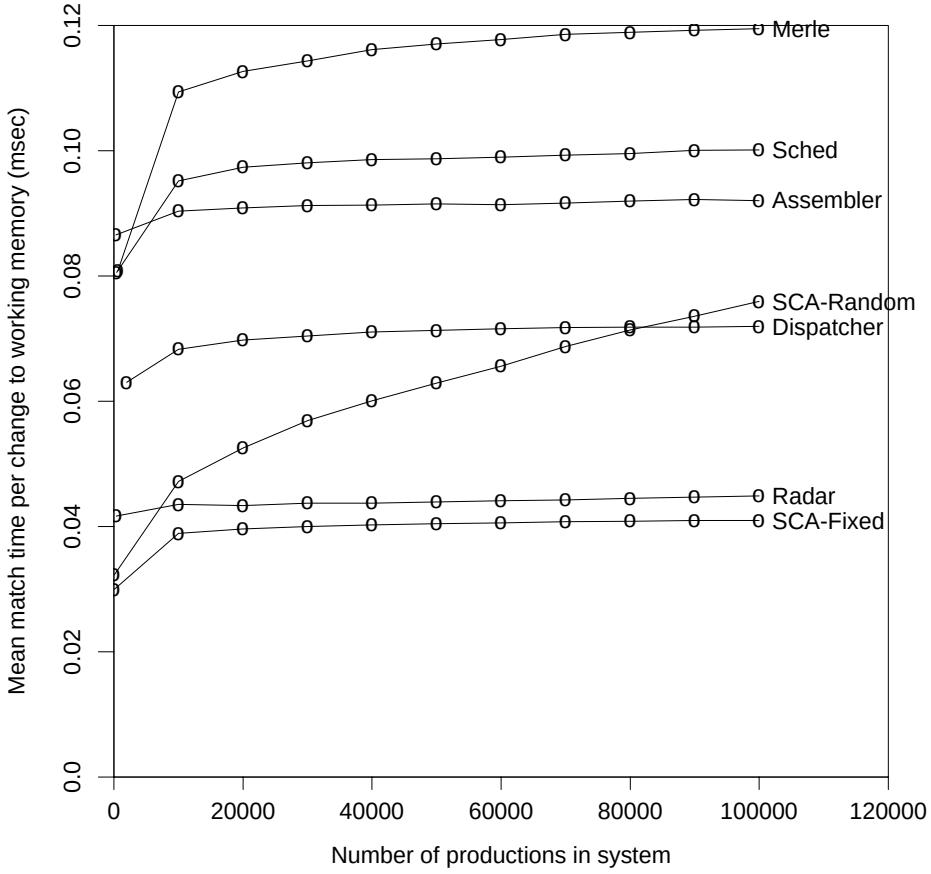


Figure 5.8: Match cost with Rete/UL.

involved. As Table 5.1 shows, the two SCA systems are the systems where adding left unlinking to right unlinking reduces the number of null activations by the smallest amount.

Figure 5.8 shows the match cost in each of the testbed systems with Rete/UL. The linear increase in match cost we saw in the unmodified basic Rete algorithm has now been eliminated in six of the seven systems — in all but SCA-Random, the increase is now either sublinear or zero. (We will examine SCA-Random in Chapter 6.) For most of the systems, the match cost does increase somewhat after the first group of learned rules is added to the initial rules; this is probably because there is little sharing of Rete nodes between the learned rules and the initial rules. As more and more learned rules are added to the network, they share more nodes with previously learned rules, and the match cost does not increase much further.[6]

[6]Although the lines in the figure appear to indicate that the remaining increase in match cost is sublinear or zero, they do not conclusively demonstrate this, since there might be a small linear effect "hidden" by a larger sublinear effect; with enough more rules, the small linear effect would eventually dominate. For the Assembler, Radar, SCA-Fixed, and Sched systems, we conjecture that the "levels-and-structures" theoretical model introduced in Chapter 6 can be applied to obtain a sublinear bound on the match cost. The Dispatcher

System	Speedup Factor
Assembler	86.3
Dispatcher	44.4
Merle	225.4
Radar	133.2
SCA-Fixed	162.0
SCA-Random	78.9
Sched	155.5
Average	126.5

Table 5.2: Speedup factors from using Rete/UL rather than basic Rete, with 100,000 rules in each testbed system.

Table 5.2 shows the speedup factors obtained from using Rete/UL rather than the basic Rete algorithm when there are 100,000 rules in each of the testbed systems. The speedup factors range from 44.4 in Dispatcher to 225.4 in Merle, with an average of 126.5. Thus, Rete/UL reduces the match cost by approximately two orders of magnitude in this diverse set of systems.

Finally, note that Rete/UL's performance scales well on a broader class of systems than Rete and Treat. We first consider Treat. Since Treat iterates over the set of productions affected by each WME, Treat's performance scales well if and only if affect sets remain small. Of course, if affect sets remain small, then the performance of both Rete and Rete/UL will scale well. We now consider Rete. In any system where Rete's performance scales well, Rete/UL's will also, because even in the worst case, the addition of unlinking to Rete merely adds a small constant factor overhead. Thus, in any system where Rete or Treat scales well, Rete/UL will scale well also. Moreover, the empirical results from our testbed systems demonstrate that Rete/UL also scales well on a large class of systems where Rete and Treat do not scale well.

5.7 Generalizing Unlinking

So far we have dealt with networks containing just *binary* joins — every join node has exactly two "input" memories. Unlinking can be generalized to k-ary joins: if any one of a join node's k input memories is empty, the node can be unlinked from each of the $k-1$ others (Barrett, 1993). This can avoid many null activations. A null activation of a k-ary join node is an activation of it from one of its memories during which any one (or more) of its other memories is (are) empty. Such activations yield no new matches or tokens, so they can be skipped without changing the semantics of the match algorithm.

and Merle systems are more complicated, making them difficult to analyze with this model. However, we can still obtain a bound on the severity of the increase in match cost in these systems. Under the pessimistic assumption that the increase is completely linear from 90,000 to 100,000 rules and thereafter, preliminary experiments indicate that it would require over eight million additional rules to double the match cost in any of these testbed systems, except SCA-Random, where about 350,000 would be required. (To obtain a tighter bound, these experiments measured the *number of CPU instructions* executed in the matcher, rather than the *CPU time*, which is influenced by changes in the cache hit rate.)

Since we do not use k-ary joins in the implementation used in this thesis, we are unable to report any detailed empirical results. Anecdotally, however, (Barrett, 1993) reports obtaining a speedup factor of about five by using this generalized form of unlinking on a system with just seven rules. This indicates that although unlinking was designed especially for systems with thousands of rules, it can be very useful even in systems with only a small number of rules. It also provides the one empirical data point we have from a non-Soar-based system — a data point which is very positive.

In Section 5.3, we saw that with binary joins, we must leave each join node linked to one of its memories when both of them are empty. With k-ary joins, a similar restriction applies: each join node must always be linked to at least one of its k memories when all of them are empty. In the first-empty-dominates combination of left and right unlinking, the join node is (i.e., remains) linked to whichever memory became empty first. First-empty-dominates can be extended to k-ary joins: if two or more of a join node's k memories are empty, the join node is (i.e., remains) linked to whichever memory became empty first.

However, with k-ary joins ($k > 2$), first-empty-dominates is no longer optimal for all systems. Consider a join node J with memories $M_1, \ldots, M_k$, and suppose J is currently linked to an empty memory M_1 and unlinked from the other $k-1$ memories. Now suppose M_1 becomes nonempty. With binary joins ($k = 2$), when this happens, we relink J to its other memory (M_2), and also unlink it from M_1 if M_2 is empty. When $k > 2$, this procedure becomes: relink J to each of $M_2, \ldots, M_k$, and then, if there is some empty memory M_i ($i > 1$), then unlink J from each of $M_1, \ldots, M_{i-1}, M_{i+1}, \ldots, M_k$. The problem is that there may be *more than one* such empty memory M_i. In this case, we have to *choose* one of them; J will then be linked to the chosen M_i and unlinked from all its other memories. (Note that with binary joins, there is never a choice: J only has one other memory, M_2.)

Of course, remaining linked to one empty memory when there are others implies risking incurring a null activation: if the linked-to memory becomes nonempty while at least one of the unlinked-from ones remains empty, we incur a null activation. So to minimize residual null activations, when we choose one such empty M_i here, we would like to choose the one that will remain empty the longest. In some cases, the choice will end up not making any difference — a null activation will later be incurred no matter what choice we make — but in cases where the choice matters, this is the best choice. Unfortunately, we cannot predict which memory will remain empty longest — questions of this sort are undecidable. A reasonable heuristic would be to predict that whichever memory has been empty for the longest time is likely to remain empty the longest. This amounts to first-empty-dominates: J ends up linked to whichever one of the empty memories became empty before all the others. Of course, while this first-empty-dominates strategy may work well in some systems, it is just heuristic, not necessarily optimal in any particular system.

There is an interesting correspondence between strategies for unlinking with k-ary joins and strategies for paging in virtual memory systems. Suppose we have a k-ary join where items always happen to get removed from memories immediately after being added to them; i.e., we add an item to some memory, then remove that item, then add some other item to some memory, then remove it, and so on, so that there is at most one nonempty memory at any time. Then all activations of the join node are null activations, and we would like an unlinking strategy

k input memories for a join node	$\longleftrightarrow$	k pages of virtual memory
can unlink from $k-1$ of them	$\longleftrightarrow$	can have $k-1$ in physical memory
must be linked to 1 of them	$\longleftrightarrow$	must have 1 swapped out
add, then remove item in memory i	$\longleftrightarrow$	access page i
no join activation on add/remove in the $k-1$	$\longleftrightarrow$	no page fault on access to the $k-1$
join activation on add/remove in the 1	$\longleftrightarrow$	page fault on access to the 1
choice of memory to remain linked to	$\longleftrightarrow$	choice of page to swap out
first-empty-dominates	$\longleftrightarrow$	least recently used

Table 5.3: Correspondence between unlinking with k-ary joins and paging in virtual memory systems.

(a way of choosing which of the $k-1$ empty memories the join node will be linked to) that minimizes the number of null activations. A null activation occurs each time an item is added to the currently-linked-to memory, at which point our strategy must choose some other memory to link to. The optimal strategy would always choose the memory that will remain empty the longest, but in general this is impossible to predict. First-empty-dominates is a heuristic designed to approximate this strategy.

As shown in Table 5.3, this situation is analogous to a virtual memory system with k pages of virtual memory but only $k-1$ pages of physical memory. At any time, $k-1$ pages are resident in physical memory, and one page is swapped out to disk. We want a page replacement strategy which will minimize the number of page faults. A page fault occurs each time the currently-swapped-out page is accessed, at which point our strategy must choose some other page to swap out. The optimal strategy would always choose the page that will not be accessed for the longest time, but in general this is impossible to predict. In this correspondence, the counterpart of first-empty-dominates — remaining linked to the memory that became empty earlier than all the others — is swapping out the page whose last access time is earlier than those of all the other pages; i.e., "least recently used" paging.

Of course, there is no paging strategy that is optimal for all systems. With any given strategy, the page selected to be swapped out might turn out to be the very next page that gets accessed. One can contrive example programs for which least recently used paging performs terribly. Similarly, there is no strategy for doing unlinking with k-ary joins that is optimal for all systems; first-empty-dominates seems to be a reasonable approach, but it could fail miserably on some systems.

5.8 Alternative Approaches

We introduced unlinking as a way to reduce or eliminate null activations. This section discusses some alternative ways to reduce or eliminate null activations. The key to the success of any approach is, of course, that it should avoid performing many null activations when the fan-out from an alpha or beta memory is large.

5.8.1 Join-Node-Table-Based Propagation

One alternative approach is to replace the whole mechanism by which dataflow is propagated from a memory node to its successors or children. In Rete, with or without unlinking, we simply iterate over a list of nodes to which dataflow should be propagated. For example, to propagate dataflow from a given alpha memory, we iterate over the *successors* list. The problem with this is that the list may contain nodes whose beta memories are empty, resulting in null activations.

From a given alpha memory, we want to propagate the dataflow to exactly the set of join nodes which both (1) are successors of the given alpha memory, and (2) have nonempty beta memories. Similarly, from a given beta memory, we want to propagate the dataflow to exactly the set of join nodes which both (1) are children of the given beta memory, and (2) have nonempty alpha memories.[7] In Rete, we iterate over the nodes satisfying (1), but do some useless work because some of them turn out not to satisfy (2). An alternative is to use criterion (2) first, at the risk of doing useless work because of failure to satisfy (1).

We call this *join-node-table-based propagation.* In this approach, we do away with the *children* and *successors* lists on alpha and beta memories, and replace them with a single global table. The table provides the following mapping:

$$\text{join-node-table}(A,B) = \{J \mid J \text{ is a join node using alpha mem. } A \text{ and beta mem. } B\}$$

This could be implemented using a hash table or some other efficient data structure. To conserve space, the table would not contain entries for pairs (A,B) for which there is no join node using both A and B. We also maintain two extra global lists, one containing all currently non-empty alpha memories, the other containing all currently non-empty beta memories:

```
nonempty-alpha-memories: list of alpha-memory
nonempty-beta-memories: list of beta-memory
```

We add code in various procedures to splice alpha and beta memories into and out of these lists whenever their empty/nonempty status changes.

Now, to propagate updates from a given alpha memory A, we use criterion (2) first, by iterating over all the nonempty beta memories B. For each such B, we propagate the dataflow to any join nodes J which use beta memory B and are successors of A. The join node table is used to find these J's.

```
{To propagate dataflow from alpha memory A}
for each B on the list nonempty-beta-memories do
    nodes ← lookup in join-node-table (A,B)
    if nodes ≠ "no-entry-in-table" then
        for each J in nodes do right-activation (J)
```

Similarly, to propagate updates from a given beta memory B, we iterate over all the nonempty alpha memories A; for each one, we propagate the dataflow to any join nodes J which use alpha memory A and are successors of B.

[7] For simplicity, we ignore the case of negated conditions here.

System	Mean Number of Wasted Lookups Per: Alpha Memory Activation	Beta Memory Activation	Mean Null Act'ns Per WME Change With Rete/UL
Assembler	152	132	0.11
Dispatcher	773	187	0.17
Merle	420	175	0.24
Radar	200	87	0.10
SCA-Fixed	45	36	0.21
SCA-Random	58	31	2.69
Sched	202	84	0.17

Table 5.4: Average number of "wasted" join node table lookups, with 100,000 rules in each testbed system, together with the average number of null activations in Rete/UL.

```
{To propagate dataflow from beta memory B}
for each A on the list nonempty-alpha-memories do
    nodes ← lookup in join-node-table (A,B)
    if nodes ≠ "no-entry-in-table" then
        for each J in nodes do left-activation (J)
```

The drawback to this algorithm is that useless work may be done because of a failure to satisfy criterion (1) after criterion (2) has already been satisfied. This is manifested by join-node-table lookups returning "no-entry-in-table" — this happens, for example, when we are trying to propagate dataflow from an alpha memory A, and there is a beta memory B which is nonempty, but none of A's successor join nodes is a child of B. These lookups are basically wasted work.

How would the performance of join-node-table-based propagation compare to that of unlinking in practice? To get an estimate of this, we measured the number of lookups that would return "no-entry-in-table" in our testbed systems, with 100,000 rules in each one. The results are shown in Table 5.4. For comparison, the table also shows the average number of residual null activations we incur using unlinking, just as in the last column of Table 5.1. The table shows that in practice, for these systems, join-node-table-based propagation would result in quite a large amount of useless work being performed, compared to the very small number of residual null activations in Rete/UL. We therefore conclude that unlinking works much better in practice, at least for these systems.

Join-node-table-based propagation does have one advantage over unlinking, though: a better worst-case performance when the number of rules is large. As we saw in Section 5.5, interference between left and right unlinking can cost us a factor on the order of $\sqrt{n}$, where n, the number of join nodes, increases linearly in the number of rules in the system. In Chapter 6, we will see that under certain restrictions, the number of nonempty alpha and beta memories can be bounded independent of the number of rules. It follows from this that the number of "wasted" lookups with join-node-table-based propagation — in contrast to the number of residual null activations with unlinking — can be bounded independent of the number of rules.

It would be nice if we could use join-node-table-based propagation for one kind of memories (alpha or beta) and unlinking on the other, so as to avoid null left activations using one technique and null right activations using another technique, without the problem of interference. Unfortunately, this is not possible. Suppose, for instance, that we use join-node-table-based propagation to go from beta memories to their child join nodes, but use the usual *successors* lists on alpha memories, with right unlinking (but not left unlinking — so whenever a beta memory is empty, *all* its child join nodes are right-unlinked, even those whose alpha memories became empty first). As usual in right unlinking, whenever a beta memory changes from empty to nonempty, we need to relink each child join node to its alpha memory. But how do we locate all these join nodes? Using join-node-table-based propagation to go from that beta memory to its child join nodes, we reach only those whose alpha memories are currently nonempty. There might be others whose alpha memories are empty, and join-node-table-based propagation will not reach them. Those nodes would incorrectly remain right-unlinked. Of course, we could keep an extra list of such nodes on the beta memory, and use this whenever the beta memory changes from empty to nonempty. However, this is essentially just a way of resorting to the standard list-based method of propagation, without left unlinking. So it would yield a linearly increasing match cost in systems with linearly increasing fan-outs from beta memories.

5.8.2 Replacing Large Fan-Outs with Binary Trees

Another way to reduce null activations when the fan-out from a memory node is large is to replace this one large fan-out with a binary tree (Stolfo, 1993). For example, suppose that in the standard Rete network, there is a beta memory with a fan-out of $n = 2^k$, its child join nodes using alpha memories $A_1, \ldots, A_{2^k}$. We replace this with a binary tree, as illustrated in Figure 5.9. Instead of propagating dataflow directly to all 2^k join nodes, we first propagate it to two nodes — the two first-level nodes of a k-level-deep binary tree, with the 2^k join nodes as its leaves. One of these first-level nodes checks whether any of $A_1, \ldots, A_{2^{k-1}}$ are nonempty; if so, it propagates the dataflow on to its two children at the second level of the binary tree. The other first-level node checks whether any of $A_{2^{k-1}+1}, \ldots, A_{2^k}$ are nonempty, and if so, propagates the dataflow on to its two children. Propagation continues down the tree in this fashion. Importantly, if only one of the alpha memories is nonempty, then only $2k$ activations are incurred, rather than 2^k. How do we perform the tests in the interior nodes of the tree? For each of these nodes, we need to maintain a counter indicating the number of nonempty alpha memories in the appropriate set. Each alpha memory contributes to $k - 1$ of these counters, so whenever an alpha memory changes from empty to nonempty, we increment $k - 1$ counters; when it becomes empty again, we decrement them.

This example shows how to replace large fan-outs from beta memories with binary trees. A symmetric construction can replace large fan-outs from alpha memories with binary trees.

How would the performance of this approach compare to that of unlinking in practice? It would certainly be slower than unlinking on all our testbed systems. This is because the binary tree approach always yields a logarithmic cost — the number of "extra" node activations we incur is proportional to the depth of the tree, hence the logarithm of the fan-out — in the worst case, and even in the best case. As the last column in Table 5.1 shows, the interference between

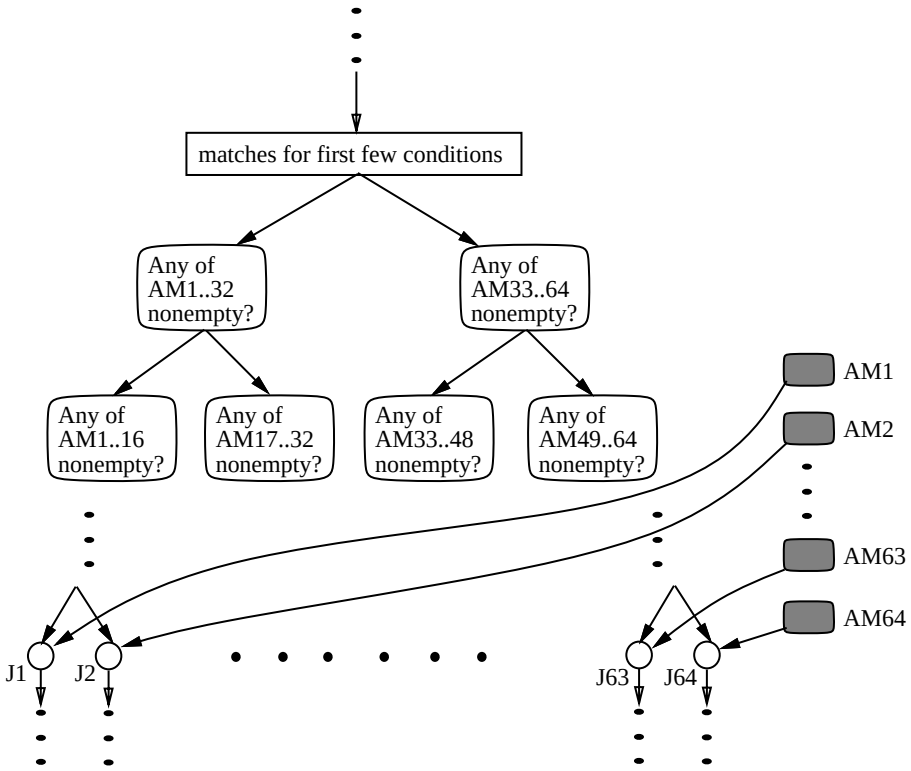


Figure 5.9: Binary tree taking the place of a large fan-out from a beta memory.

left and right unlinking is so small in practice that unlinking would almost certainly yield fewer "extra" activations (i.e., null activations) in practice. Of course, the worst-case performance of the binary tree approach is better than that of unlinking, since this logarithmic cost is better than the square-root worst-case cost in unlinking.

5.8.3 The Tree Match Algorithm

The Tree match algorithm has recently been proposed as an alternative to Rete in certain systems (Bouaud, 1993). Although it was not intended to address null activations, Tree turns out to avoid null right activations, as we will discuss shortly. It was developed as a way to reduce the cost of performing individual join operations. In this respect, it is similar to the use of hashed or indexed memories, as discussed in Section 2.3; however, its approach is different.

We briefly summarize Tree here; see (Bouaud, 1993) for a full presentation. In Rete, when a WME is added to working memory, we feed it into a dataflow network, eventually right-activating many join nodes. On each right activation, we attempt to "extend" each existing token in a beta memory, i.e., add the new WME to the existing token in order to form a new token. Tree's aim is to replace this dataflow with a simple index. The basic idea is to maintain a global index of all tokens in all beta memories, so that when given a new WME, we can quickly locate all tokens that the WME can extend. In other words, we index all tokens according to their answers to the question, "What would a WME have to look like in order to extend this

token?" Similarly, to process left activations, Tree maintains a global index of all WMEs in working memory, so that when a new token is passed to a join node, it can quickly locate all WMEs that can extend the token.

The key to Tree's performance is the effectiveness of these global indices. First, they must have *high selectivity*: given a new WME or token, the index lookup should return just the appropriate tokens or WMEs, with as few "extras" as possible. Second, they must yield *fast access*: storing a new WME or token should be fast, and lookups should be fast.

In Tree, each global index is actually three separate index structures: one indexes according to the contents of the identifier field, one according to the attribute field, and one according to the value field. Tree is designed especially for systems where WMEs have the simple three-tuple form we have been using in this thesis. These three indices definitely offer fast access — given a WME, we can quickly store it in each of three indices, and perform three quick lookups to retrieve a set of tokens. The empirical results in (Bouaud, 1993) indicate that the selectivity of these indices is roughly comparable to that obtained using hashed memories in standard Rete.

The main drawback to Tree is that it is unclear how to modify these indices to handle WMEs of a more general form, while still affording both fast access and high selectivity. Consider r-tuple WMEs. A straightforward extension of Tree would store each WME or token under r different indices, each one indexing according to the value of one field. The larger r is, the lower the selectivity of these indices will be, because each one essentially ignores $r-1$ of the fields of the WME. In contrast, hashed memories in standard Rete can index on an arbitrary number of fields, simply by having the hash function take into account the value of more than one field. Of course, we could add additional index structures to Tree to obtain higher selectivity; for example, we could obtain perfect selectivity by having a separate index for each of the 2^r possible subsets of the r fields. But increasing the selectivity this way slows down access times, since now a WME addition requires a large number of index structures to be updated or searched.

As mentioned earlier, Tree avoids all null right activations. This is because if there are no tokens in a given beta memory in Rete, then there will be no corresponding entries in the global token index in Tree. So on a WME addition in Tree, the result of the index lookup will not contain any tokens from that beta memory, and Tree will not waste any CPU cycles doing processing for any of its child join nodes.

Tree does not address the problem of null left activations, however. Like the basic Rete algorithm, it iterates over all the join nodes that are children of a given beta memory. For each one, it asks, "What would a WME have to look like in order to extend this token at this join node?" and stores an appropriate entry in the global token index.

Unfortunately, incorporating left unlinking into Tree yields exactly the same interference problem as we encountered with right unlinking. We could modify Tree to iterate over only *some* of the child join nodes, storing entries in the global token index for only these join nodes. However, the global token index would then be incomplete. So we would have to modify the *add-wme* processing to perform extra checks, in order to preserve the correctness of the match algorithm in spite of the incomplete index. These extra checks essentially amount to extra right activations, many of which may be null.

5.8.4 Generalized Dataflow Reduction

We have now seen several optimizations which are essentially aimed at reducing the dataflow in the Rete network: right unlinking, left unlinking, join-node-table-based propagation, and Tree. In this section, we introduce a framework in which these and other optimizations can be viewed.

In general, we can skip any element of dataflow — i.e., propagating a WME or token to a node — if we can guarantee that it will be fruitless, i.e., that it will not result in any new matches. It is helpful to view this skipping of dataflow as being the result of placing *blockages* on certain dataflow links in the network. For example, left unlinking is essentially a way of placing a blockage on the link between a beta memory and one of its child join nodes.

A blockage can safely be placed on a given link only if we can guarantee that any dataflow it prevents would be fruitless. To be able to make such a guarantee, we must have information about the current status of other parts of the network. For example, with left unlinking, in order to guarantee that the blockage between a beta memory and a join node is safe, we need to know that the join node's alpha memory is currently empty. When the beta memory is activated, the blockage acts as a way of taking into consideration information — the fact that the alpha memory is empty — which would normally be considered only later — during the join node activation. In general, a blockage is essentially a way of moving part of a test into a generator: it takes information that would normally enter into consideration only later in the dataflow, and arranges for it to be considered earlier in the dataflow.

In Rete, there are several possibilities for moving information "upstream" in the dataflow:

- Blockages can be placed on links in the beta network based on information normally considered lower down in the beta network — specifically, information about the contents of alpha memories used by join nodes lower down in the network.

- Blockages can be placed on links between alpha memories and join nodes, based on information normally considered at those join nodes — specifically, information about the contents of the beta memories used by those join nodes — or based on information normally considered lower down in the beta network — specifically, information about the contents of alpha memories used by join nodes lower down in the network.

- In versions of Rete using a dataflow implementation of the alpha network, blockages can be placed on links between constant-test nodes, based on information normally considered later in the alpha network or in the beta network.

This last possibility would yield little benefit in most systems, because the alpha net typically accounts for only a small fraction of the overall match cost. Consequently, we confine our discussion to the first two.

Schemes for using blockages can be classified along three dimensions. First, blockages can be placed in links in the beta part of the network, or on links between the alpha part and the beta part, as we just discussed. Second, blockages can be either coarse-grained, blocking all dataflow along the links, or fine-grained, blocking only some of the dataflow. Third, blockages can represent the moving of information upstream either a single level, so it gets considered

	Coarse-Grained	Fine-Grained
Alpha-to-Beta, Single-Level	Right Unlinking	Tree
Alpha-to-Beta, Multiple-Level	Treat	Rete/AC
Beta-to-Beta, Single-Level	Left Unlinking	none
Beta-to-Beta, Multiple-Level	Deactivating Beta Nodes	none

Table 5.5: Classifying existing schemes for dataflow reduction within the space of possibilities.

only one node (one dataflow link) earlier than it normally would, or multiple levels, so it gets considered several nodes earlier.

We now examine various existing schemes for reducing dataflow in Rete or related match algorithms. Table 5.5 shows the space of possible schemes, using the three dimensions just discussed, and locates existing schemes within the space. Of course, the usefulness of any scheme in practice depends on (1) how much dataflow it saves, and (2) how much overhead cost it incurs in the process. We want to find schemes that save a great deal of dataflow with only a very small overhead.

The first row of the table lists schemes which place blockages on links from alpha memories to join nodes, based on moving information a single level upstream. Information normally used at a join node — specifically, information about the tokens in its beta memory — is used to place a blockage in between the join node's alpha memory and the join node. Right unlinking does this in coarse-grained fashion. Tree is a fine-grained way of doing this: although there are no alpha memories per se in Tree, its global token index is basically a way of reducing the flow of WMEs to join nodes, based on information contained in their beta memories. Empirically, right unlinking reduces the dataflow tremendously in our testbed systems, with only a small overhead cost. Tree would probably have the same effect; although we have not tried it in these systems, the algorithm is quite simple, so its overhead is probably small.

The second row of the table lists schemes which place blockages on links from alpha memories to join nodes, based on moving information multiple levels upstream. The Treat match algorithm associates a *rule-active* flag with each rule, indicating whether every alpha memory used by that rule is nonempty. Treat does not perform any joins for a rule if it is not active, i.e., if any of its alpha memories are empty. This can be viewed as a coarse-grained way of reducing the flow of WMEs to join nodes, based on information multiple levels away. This blockage is somewhat different from those in right unlinking and Tree: instead of being implicit (in the absence of a join node in a *successors* list or the absence of any tokens in the global index), the Treat algorithm *explicitly checks* the rule-active property for each rule in a WME's affect set. So while this blockage would reduce the dataflow tremendously in our testbed systems, Treat would still slow down linearly in the number of rules. A fine-grained version of this blockage scheme has been implemented in Rete/AC, which is formed by adding factored arc consistency to Rete (Perlin, 1992). In Rete/AC, an arc consistency algorithm is used to prune the set of WMEs in each alpha memory that need to be considered during the beta part of the match. Unfortunately, the empirical results in (Perlin, 1992) indicate that the overhead of running the arc consistency algorithm usually outweighs the savings in dataflow.

The third row of the table lists schemes which place blockages on links from beta memories

to join nodes, based on moving information a single level upstream. Information normally used at a join node — about the WMEs in its alpha memory — is used to place a blockage between the beta memory and the join node. Left unlinking does this in coarse-grained fashion, and as discussed in Section 5.6, its benefits in our testbed systems usually, but not always, outweigh its overhead costs. To our knowledge, no fine-grained blockage scheme of this type has been implemented. Such a scheme might be better than left unlinking in systems where beta memories have large fan-outs but left unlinking is too coarse-grained to avoid many fruitless left activations. For example, it is possible to defeat left unlinking by adding an "extra" useless WME to all the alpha memories — with all the alpha memories nonempty, we cannot left unlink any join nodes, and yet left activations of these join nodes would often be fruitless. A fine-grained scheme would be useful in such situations.

Finally, the last row of the table lists schemes which place blockages on links from beta memories to join nodes, based on moving information multiple levels upstream. Information normally used lower down in the beta network — about the contents of alpha memories used by join nodes lower down in the network — is used to place a blockage in between a beta memory and a join node. To our knowledge, no such scheme has been implemented in any match algorithm. However, (Perlin, 1990b; Perlin, 1988) proposes a scheme for "deactivating" beta nodes based on coarse-grained information moved multiple levels upstream from *production nodes* rather than alpha memories. The idea is that one can "disable" a set of productions by blocking off any sections of the Rete network that are used by those productions only. This would be useful in systems where at any given time, a large number of productions can be ignored *a priori*.

In general, there is a limit to how much benefit schemes of the last type can yield, at least for our testbed systems and any other systems where right unlinking or other alpha-to-beta, single-level schemes are important. The reason involves a general interference phenomenon between dataflow reduction schemes. In order to achieve large dataflow reduction via a beta-to-beta, multiple-level blockage, we must take information in alpha memories and move it many levels up the beta part of the network — the more levels up we move it, the larger the dataflow reduction. When a blockage is placed multiple levels up from its information source, the beta memories in between — below the blockage but above the join nodes whose alpha memories are the information source — are not kept up-to-date, i.e., they may not contain tokens they would if the blockage were absent. As a result, single-level, alpha-to-beta dataflow reduction schemes cannot be used at join nodes in these intermediate levels — the appropriate information cannot be moved over to the alpha memories, because the appropriate information is simply unavailable. Thus, interference between alpha-to-beta, single-level schemes and beta-to-beta, multiple-level schemes limits the effectiveness of any combination of the two. A similar argument applies to the combination of alpha-to-beta, single-level schemes with alpha-to-beta, multiple-level schemes.

5.8.5 Modifying Treat

As we mentioned before, the Treat match algorithm (Miranker, 1990) iterates over all the rules affected by a WME, and hence would not scale well in our testbed systems. It is interesting to examine how Treat might be modified to improve its performance in these systems. We give

some ideas in this section; however, implementing and evaluating them remains an interesting direction for future work. We begin with a brief review of Treat.

In Treat, we use an alpha network just as in Rete. However, instead of using a beta network, with beta memories and join nodes, we just keep on each alpha memory a list of all the productions that use that alpha memory. When a WME is added to working memory, we first use the alpha network as in Rete; this yields a list of all the productions in the WME's affect set. Now, for each affected production having, say, C conditions, we find any new complete matches by forming the C-way join between this new WME and the contents of all the other alpha memories the production uses; i.e., for a production using alpha memories $AM_1, \ldots, AM_C$, if a WME w is added to AM_i, we find new matches by evaluating $AM_1 \bowtie \ldots \bowtie AM_{i-1} \bowtie \{w\} \bowtie AM_{i+1} \bowtie \ldots \bowtie AM_C$. As a shortcut, we first check whether any AM_j is empty, in which case the join result is null — this is the *rule-active* property discussed above. To handle WME removals in Treat, we use list-based removal (see Section 2.5). Note that since we do not store tokens in beta memories, we need to delete only complete matches (i.e., conflict set entries) involving the WME being removed, so removals are extremely fast in Treat.

If Treat is to perform well in our testbed systems, we must modify it so it does not always iterate over the set of affected productions. One way to do this is to incorporate unlinking into the algorithm, but still leave out sharing. Another way is to incorporate sharing. We consider each of these in turn.

Treat performs the beta part of the match separately for each production. The match for a production having C conditions is essentially done using a single C-ary join node. The trouble is that in our systems, the fan-outs from alpha memories to these C-ary join nodes would grow very large. Recall our discussion of generalizing unlinking to k-ary joins in Section 5.7. These fan-outs could be significantly reduced by incorporating unlinking — splicing productions out of and back into the lists on alpha memories.

Another way we might modify Treat to avoid iterating over all the affected productions is to incorporate sharing into the beta part of the match. Unfortunately, this is not straightforward. We want to modify Treat so it uses a network of sometimes-shared beta memories and join nodes, just as in Rete, except that we do not *save* the contents of beta memories — after we finish processing each WME, we throw away any tokens that were generated during its handling. (If we didn't throw them away, this algorithm would just be *Rete*.) So as a first cut, we might try an algorithm basically the same as Rete except that the contents of beta memories are calculated "on demand" — whenever we need to know the tokens in a given beta memory, we perform any necessary joins for earlier conditions, and "cache" the results in the given beta memory and others higher up. At the conclusion of the processing of each WME addition, we delete any cached join results.

There are two problems with this scheme. First, it will incur a large number of null left and null right activations, just as the basic Rete algorithm does. Second, it does not make use of the *rule-active* property, an important optimization in Treat. These problems are related. In order to avoid null left activations, we simply add left unlinking, but *right* unlinking cannot be added (yet) because it requires continuous knowledge of the empty/nonempty status of beta memories — and we keep throwing away this information. We can easily save an *approximation* of this information, though, without much extra space. Instead of maintaining the true

empty/nonempty status of each beta memory, we maintain an *earlier-conditions-active* flag on each beta memory; this flag is true if and only if all the alpha memories used for earlier conditions are nonempty. Note the similarity to *rule-active* here — the *rule-active* property of a production is equivalent to the *earlier-conditions-active* flag its p-node would have (if we used such flags on p-nodes). With this flag, we can safely right-unlink a join node if its beta memory has *earlier-conditions-active* false.

This approach achieves only part of the dataflow reduction that the standard Treat algorithm achieves using *rule-active*. To get the full reduction, we would also need *later-conditions-active* information. This would be an instance of an alpha-to-beta, multiple-level dataflow reduction scheme, as discussed in the previous section. Unfortunately, this would lead to interference with right unlinking, as we discussed — the more we reduced dataflow using *later-conditions-active* flags on nodes higher in the network, the less we would be able to reduce dataflow using *earlier-conditions-active* flags on nodes lower in the network. So it is unclear whether the full dataflow savings of *rule-active* can be achieved by a version of Treat that does not iterate over all the affected productions.

The techniques we have discussed here are all ways of reducing or avoiding increasing null activations, a problem which has been the subject of this chapter and the previous one. We addressed this problem by incorporating left and right unlinking into Rete, and presented empirical results from the resulting Rete/UL algorithm. The next chapter complements the largely empirical character of this thesis by giving a theoretical analysis of match cost, including an analysis of the circumstances under which Rete/UL guarantees efficient matching. The chapter also discusses the remaining slowdown in SCA-Random.

Chapter 6

Theoretical Analysis

In this chapter, we give a theoretical analysis of several important questions about match cost. The theoretical nature of this chapter stands in contrast to the predominantly empirical character of the rest of this thesis. It is important to keep in mind while reading this chapter that theoretical analyses, especially those concerning worst-case behavior, do not necessarily reflect what commonly happens in practice. For example, we have already seen in Chapter 5 that in the theoretical worst-case, interference between left and right unlinking can be so bad that residual null activations slow down the matcher by a factor of $O(\sqrt{n})$, where n is the number of join nodes in the network, but the empirical results on our testbed systems show the interference to be insignificant. Although many of the results in this chapter may seem discouraging, we have already shown that match cost can be kept quite small on most of our testbed systems.

Our analysis asks under what circumstances efficient matching can be guaranteed. By "efficient," we mean the match cost should be (1) polynomial in W, the number of WMEs in working memory; (2) polynomial in C, the number of conditions per production;[1] and (3) sublinear in P, the number of productions.

The rest of this chapter is organized as follows. We first consider requirements (1) and (2) in Section 6.1. These requirements probably cannot be met in general, but they can be met if we restrict the representations used in working memory and productions. We then consider requirement (3) in Section 6.2. By placing some restrictions on what types of conditions we allow, such a bound can in fact be achieved; unfortunately, the bound is so high that it would be unacceptable for most systems in practice. From these general discussions of matching using any algorithms, we turn to one particular match algorithm — Rete/UL — and ask in Section 6.3 under what circumstances we can guarantee efficient match with it. We also explain in that section why SCA-Random still has a significantly increasing match cost, while SCA-Fixed does not. Finally, we discuss in Section 6.4 whether efficient match can be guaranteed through simple representational restrictions alone.

[1] For simplicity, we assume here that all rules have the same number of conditions. We also assume that all rules are different — no two rules have exactly the same conditions.

6.1 Keeping Match Cost Polynomial in W and C

We first ask whether match cost can be kept polynomial in W, the number of elements in working memory, and C, the number of conditions per production. In general, the answer is "no." The amount of output a match algorithm must generate (i.e., the list of all complete matches for all productions) can be exponential in C. Consider a system with W WMEs in working memory, and just one rule, having C conditions, each of which happens to match all W WMEs. The number of different matches for this rule is W^C. In this case, of course, the running time of the match algorithm must be exponential in C.[2] We might consider easing the matcher's job by requiring it merely to list the productions having *at least one* complete match, without listing all the different matches. Unfortunately, this is of little help — even in this case, match cost cannot be kept polynomial in W and C unless P=NP. Without any restrictions placed on the WMEs or conditions allowed, determining whether a given rule has any complete matches is NP-hard. This is easily proven using a reduction from one of various graph problems, particularly graph colorability (Smith and Genesereth, 1986) or subgraph isomorphism (Minton, 1988a, Section 2.5). For subgraph isomorphism, the basic idea is that given two graphs, we can encode one using a set of WMEs and the other using a set of conditions for a single rule; that rule then has a match if and only if the graph it represents is isomorphic to a subgraph of the other.

If we place certain restrictions on the WMEs or conditions allowed, though, matching a single rule can be done in time polynomial in W and C. A variety of such restrictions are possible (Tambe and Rosenbloom, 1994). For example, with the *unique attributes* restriction on working memory (Tambe et al., 1990), a single rule can be matched in time linear in W and C. With such restrictions, matching a single rule is no longer NP-hard — the aforementioned NP-hard graph problems can no longer be transformed into questions of whether a single rule has a match. Instead, they can be transformed into questions of whether any of a large set of rules has a match; the number of rules required is exponential in the size of the given graphs. For example, instead of transforming a graph colorability problem into a set of WMEs plus one production which tests for the existence of any legal coloring, we can transform it into a set of WMEs plus many productions, each of which tests for the legality of one particular coloring. Since the number of possible (legal or illegal) colorings is exponential in the size of the graph, so is the number of productions we need.[3]

It is tempting to conclude from this that even with these restrictions on WMEs or conditions, the match cost of a large number of rules must necessarily be large in the worst case, since the matcher is essentially being made to answer the same NP-hard graph problem as with one rule and no restrictions. This is incorrect. The transformations used in proofs of NP-hardness — e.g., "given two graphs, here's how we transform them into a set of WMEs and

[2] With this simple example production, match algorithms such as Collection Rete (Acharya and Tambe, 1992) may avoid this by denoting the W^C matches by the cross-product of C sets, each containing W WMEs. However, this technique can easily be defeated. By adding appropriate inequality tests to the production, we can have it pick out $W!/(W-C)!$ *permutations* of the WMEs, and these cannot be denoted as a simple cross product.

[3] This does not mean systems using these restrictions in practice necessarily require exponentially many rules. If they are solving problems which are not NP-hard, they may need only polynomially many rules. Moreover, a system solving an NP-hard problem could match a small number of rules to solve a few commonly-occurring problem instances quickly, but resort to non-match methods to solve other instances.

a set of conditions" — must run in polynomial time. With these restrictions on WMEs or conditions, we have transformations that run in exponential time, since their output is a list of an exponential number of rules. Thus, the aforementioned reduction of graph colorability to the problem of matching a large number of rules with unique attributes is not a polynomial-time reduction. Another way of looking at this is as follows. Consider any NP-hard problem. Suppose we are asked to solve problem instances of size N (their descriptions are N bits long). We can precompute the answers to all 2^N possible problem instances and store the answers in a big table. (This takes a very long time, but once it's done, that doesn't matter any more.) Now, given a problem instance of size N, we can "solve" it simply by doing a table lookup. The "solving" step can be done very efficiently, because an exponential amount of work was done in the preprocessing phase. In the same way, it *may* be possible to match an exponential number of restricted-representation rules very efficiently, because the matcher is not really "solving" an NP-hard problem, but merely "looking up" a previously-computed solution to one instance of it.

In summary, if WMEs and conditions are unrestricted, then matching a single rule is NP-hard, hence no algorithm can guarantee efficient matching (unless P=NP). With certain restrictions on WMEs or conditions, there are algorithms that match each individual rule in time polynomial in W and C. Unfortunately, the total match cost of all rules can still be linear in P with these algorithms, as we discuss in the next section.

6.2 Keeping Match Cost Sublinear in P

We now turn to the question of whether match cost can be guaranteed to be sublinear in P, the number of rules. In general, the answer is again "no." This is because the number of rules that successfully match can be linear in the total number of rules; this implies that the amount of output a match algorithm must generate (i.e., the list of all complete matches) can be linear in P. For example, consider a system with P rules, each of the form "If today's winning lottery number is *not* $\langle d_1 d_2 \ldots d_k \rangle$, then ...," where each rule has different particular values for the digits d_i. The number of successfully matching rules here will be at least $P - 1$.

Clearly, the key to this contrived example is the presence of the word "not." Remove the *not*, and the number of matching rules will be at most one. Another way to construct contrived examples like this is to use tests other than equality tests — e.g., "less-than," or "greater-than." If we replace the "not" in the above example with "less than," then $P/2$ of the rules will match on average.

Suppose we restrict the conditions that can appear in rules so as to preclude examples like these. We will require that all conditions be positive — no negative conditions (or conjunctive negations) are allowed. Furthermore, all tests appearing in conditions must be tests for equality — either equality with a constant or a variable. In this case, it turns out that match cost can not only be bounded *sublinearly* in the number of rules, but in fact can be bounded *independent* of the number of rules. We formalize this result as follows:

Theorem 6.1 (Matching with Simple Positive Conditions) *Suppose all conditions in all rules are positive conditions containing equality tests only. Then all matches of all rules can be found with an algorithm whose worst-case running time is independent of the number of rules.*

Proof sketch: The key observation here is that whatever the W WMEs in working memory are, there are only finitely many ways to write a set of C conditions that match them. (That is, except for renaming variables; we can canonicalize the variable names in all the rules ahead of time to take care of this.) The number is very large — exponential in C — but it is finite. So to find the set of matching rules, we can just iterate over all the possible syntaxes for a matching set of conditions, and for each one, check to see whether we have a rule with that syntax. The number of syntaxes depends on W and C, but is independent of P, so this algorithm runs in time independent of P.

Why is the number of matching syntaxes finite? If a rule has C conditions, then a match for it is some C-tuple of WMEs $\bar{w} = \langle w_1, \ldots, w_C \rangle$. If there are W WMEs in working memory, then there are W^C possible C-tuples. How many ways can we write C conditions that match a given C-tuple $\bar{w}$? By the theorem's hypothesis, each field in each condition tests for either equality with a constant or equality with a variable. So all syntaxes matching $\bar{w}$ can be obtained by starting with $\bar{w}$ and replacing some of the constants with variables. With three fields in each WME, there are a total of $3C$ fields; each can either remain a constant or be replaced with a variable. Since we canonicalize variable names, and we will end up with at most $3C$ constants being replaced with variables, there are at most $3C$ variable names to choose from. Thus, there are at most $3C + 1$ things that can happen to each field. So there are at most $(3C + 1)^{3C}$ syntaxes that match $\bar{w}$. With W^C possible $\bar{w}$'s, we have an upper bound on the number of ways to write C conditions that match working memory: $W^C(3C + 1)^{3C}$. A more careful analysis could tighten this bound a bit, but that is unimportant; it would still be exponential in C (since the W^C part cannot be tightened) and independent of P. □

Of course, it would be foolish to use this exhaustive generate-and-test algorithm in practice. If we use Rete/UL, we obtain a bound on match cost which is exponential in C and sublinear in but not independent of P. The reason for this bound is as follows. Without negative conditions or non-equality tests, the number of alpha and beta memories that are nonempty at any time is bounded independent of P (but exponential in C) — this follows from an analysis similar to the one above, using the fact that memories store k-tuples of WMEs ($0 \leq k < C$). From this it follows that the number of *non-null* activations per change to working memory is bounded independent of P. With the first-empty-dominates combination of left and right unlinking, the additional cost of null activations adds a factor of at most $O(\sqrt{n}) = O(\sqrt{PC})$ (see Theorem 5.4 on page 108). Thus, the upper bound on the total match cost is exponential in C but only sublinear in P.

This bound can be improved further by using the join-node-table-based propagation technique introduced in Section 5.8.1. Since this method avoids all null activations, we avoid the additional factor of $O(\sqrt{n})$ incurred above. So the upper bound on the total match cost is then exponential in C but independent of P.

Unfortunately, these theoretical results are not very useful in practice. The worst-case bounds are exponential in the number of conditions per production: $O(W^C)$ or worse. For the values

Match cost sublinear in P?	Match cost poly in $W\&C$?	Theoretical results
no	no	any algorithm will suffice
no	yes	various restrictions on representations
yes	no	no negative cond's & no non-equality tests
yes	yes	unclear — see Sections 6.3 & 6.4

Table 6.1: Summary of theoretical results on efficient matching.

that typically arise in practice — W in the tens or hundreds, and C anywhere from about five to fifty — the bounds are astronomical. If worst-case behavior were to arise in practice, these match algorithms would take centuries to run, even on a supercomputer. Moreover, what might happen in practice is that as the number of rules increases, the match cost would increase nearly linearly for some initial period, then gradually approach the bound as an asymptote. If the initial period lasts for the first trillion rules, the asymptote would be irrelevant for all practical purposes.

Table 6.1 summarizes the theoretical results discussed so far in this chapter. Representational restrictions such as unique attributes yield a match cost polynomial in W and C but linear in P. Disallowing negated conditions and non-equality tests yields a match cost sublinear in (in fact, independent of) P, but exponential in C. Unfortunately, combining this with unique attributes still leaves the total match cost exponential in C (though the cost per rule is only linear in C) — at least with current match algorithms. Whether some as-yet-undeveloped match algorithm can yield a bound polynomial in W and C and sublinear in P without overly restricting the WMEs and conditions allowed is an open question. We will say more about this later, but we have no definitive answer.

6.3 Analysis of Rete/UL

We now turn to the question of under what circumstances our current best match algorithm (in practice), Rete/UL, can be guaranteed to perform efficiently. We will assume rules do not contain negative conditions or non-equality tests. As discussed in the previous section, we cannot guarantee efficient match if we allow (unlimited) use of such constructs.

The basic idea of our analysis is to limit the number of tokens that will be present in the network at any given time. We will present conditions under which the number of tokens can be kept small and bounded independent of the number of productions. It follows from this that the number of nonempty beta memories is similarly bounded. Moreover, the number of nonempty alpha memories is also bounded — it is at most $8W$, since each WME can go into at most eight different alpha memories, as discussed in Section 2.2.3. From these bounds, it follows that the number of non-null activations per change to working memory is similarly bounded, since a non-null activation requires a nonempty alpha memory and a nonempty beta memory. Finally, since null activations can add at most a square-root factor to the match cost, the total match cost will still be sublinear in the number of rules.

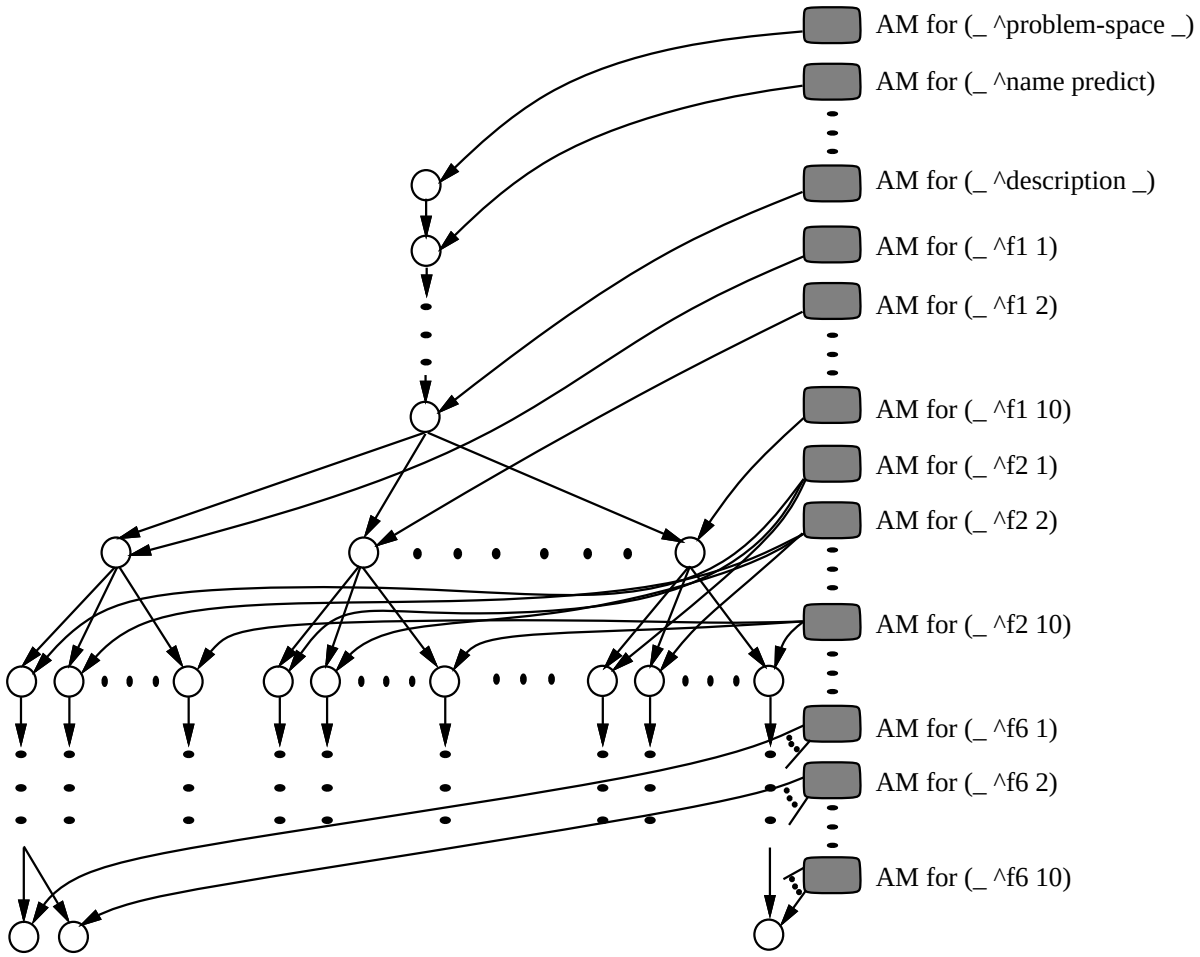


Figure 6.1: Simplified view of the Rete network for SCA-Fixed.

How can we keep the number of tokens small as the number of productions grows very large? An illustrative example is provided by the SCA-Fixed system we discussed at length in Chapter 4. Figure 6.1 shows a simplified view of the Rete network for SCA-Fixed. The beta part of the Rete network in this system can be divided conceptually into several levels; as dataflow propagates down the network, a number of join nodes are activated at each level, but only one of them "succeeds," finding any new matches and creating any new tokens. The result is that at any given time, tokens are present in beta memories *only along one path* from the top of the beta network to a leaf, so the number of tokens stays small, no matter how many productions there are.

6.3.1 The Levels-and-Structures Model

We now develop our model by abstracting the details away from this simplified SCA-Fixed example. In SCA-Fixed, tokens are present only along one path from the top node to a leaf. We will allow multiple paths, as long as the number of paths is kept small and independent

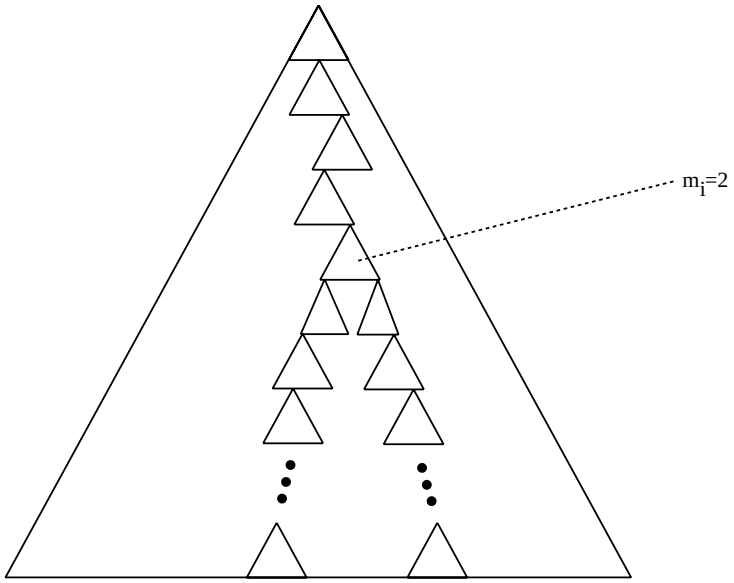


Figure 6.2: The levels and structures model. At any given time, only a few small subnetworks out of the whole beta network contain tokens.

of the number of productions. We will also generalize the notion of a "path" — instead of a sequence of individual beta memories and join nodes, a path will be a sequence of *subnetworks*, i.e., portions of the whole beta network. This is illustrated in Figure 6.2. The large triangle represents the whole beta network; small triangles represent subnetworks. (The "$m_i = 2$" will become clear shortly.) The figure shows two top-to-leaf paths. If tokens are only present along a small number of paths at any given time, then they will only be present in a small number of subnetworks at any given time.

Suppose we bound the number of token-containing subnetworks; call this bound E. In Figure 6.2, E would be the number of small triangles — these are the subnetworks containing tokens. Suppose we also bound the number of tokens in any single subnetwork; call this bound T_{single}. In the figure, T_{single} would be the maximum number of tokens in any single small triangle. Let T_{tot} denote the maximum total number of tokens in the whole network (the big triangle) at any given time. Our goal is to limit the total number of tokens T_{tot}. From this construction, $T_{\text{tot}} \leq ET_{\text{single}}$. Since clearly $E \geq 1$ and $T_{\text{single}} \geq 1$, in order to keep T_{tot} small, we must keep both E and T_{single} small. We will return to this shortly, after briefly discussing how this model would be applied.

In general, we need to partition the beta network into some number of levels; call the number of levels ℓ. In Figure 6.2, each level corresponds to a row of subnetworks. Each level can test for the presence of arbitrary "structures" in working memory — arbitrary groups of WMEs — rather than just simple feature/value pairs represented by a single WME as in SCA-Fixed. For example, in a "blocks world" system, one "structure" might be a red block on top of a green block; this would be represented by a group of three WMEs, two for `color` and one for `on-top`. Note that this means a "level" of the network may be several join nodes "deep." A structure corresponds to one path of nodes from the top of a subnetwork (small triangle) to the bottom

of that subnetwork.

When we apply this model to a given system, it is up to us, the modelers, to pick the "levels" and "structures." For every rule in the system, with conditions $c_1, \ldots, c_k$ *in that order from the top of the network to the bottom*, we view this sequence of k conditions as the concatenation of ℓ subsequences. We do this by picking "cut points" $i_1 < i_2 < \ldots < i_{\ell-1}$, so that the subsequences are $c_1, \ldots, c_{i_1}$; $c_{i_1+1}, \ldots, c_{i_2}$; ...; and $c_{i_{\ell-1}+1}, \ldots, c_k$. Each of these subsequences becomes a structure. Note that none of these subsequences is allowed to be null; i.e., the cut points must all be different. Also note that our cut points need not be the same on all rules — we can pick them differently for different rules. Once we have chosen these levels and structures, if we know enough about the distribution of WMEs in working memory, we can figure out what E and T_{single} will be. The important question is whether there is some way to choose these cut points on all the rules in the system so that both E and T_{single} turn out to be small — if there is such a way, then Rete/UL is guaranteed to perform efficiently.

We now analyze E in more detail; T_{single} will be dealt with below. Whenever dataflow propagates from one level to the next lower one — say, from level $i-1$ to level i — the subnetwork at the lower level tests for the presence of various structures in working memory. To keep E small, we want it to be the case that only a very small number of these various structures-being-tested-for are ever present in working memory at the same time; call this number m_i. We assume $m_i > 0$, otherwise there would never be any complete production matches. In our SCA-Fixed example, all the m_i's equal one. In Figure 6.2, one m_i is two and all the rest are one. In general, we will want most of $m_1, \ldots, m_\ell$ to be one, but we will not require them all to be. We will, however, want them all to be fairly small numbers independent of the number of productions in the system.

Given this model, how many subnetworks could contain tokens at any given time? We consider one level at a time. Let E_i denote the number of token-containing subnetworks at level i. Since there is only a single subnetwork at level one (this is the subnetwork rooted at the top node), $E_1 = 1$. What about level two? Since at most m_1 of the structures tested at level one are present at one time, at most m_1 of the subnetworks at level two will contain any tokens. (A subnetwork can contain tokens only if there is a match for all the preceding conditions.) So $E_2 \leq m_1$. Similarly, in each of these m_1 token-containing level-two subnetworks, at most m_2 structures-tested-for are present, and so at most $m_1 m_2$ subnetworks at level three will contain tokens: $E_3 \leq E_2 m_2 \leq m_1 m_2$. In general, $E_i \leq E_{i-1} m_{i-1} \leq m_1 m_2 \cdots m_{i-1}$. Hence,

$$E = \sum_{i=1}^{\ell} E_i \leq 1 + \sum_{i=2}^{\ell} \prod_{j=1}^{i-1} m_j \leq \ell \prod_{j=1}^{\ell-1} m_j$$

Note that as long as most of the m_j's are one, and the others are fairly small, E will be fairly small. On the other hand, if most of the m_j's are larger than one, then E will be quite large.

We now analyze T_{single} further. We present two simple ways to bound T_{single} here; there may be others. One way is simply to make sure all the subnetworks are very small. Recall that each structure is represented by some number of WMEs, and rules test for its presence using corresponding conditions. Let s be the size of the largest structure that rules test for, i.e., the largest number of WMEs or conditions any structure requires. Each subnetwork is then a

network with depth at most s, as opposed to the depth-C whole network. Now we can obtain a bound for T_{single} by a counting argument similar to that in Section 6.2: just as the number of tokens in the whole depth-C network is bounded exponential in C, the number of tokens in a depth-s subnetwork is bounded exponential in s. Thus, T_{single} can be bounded exponential in s but independent of P, the number of productions. Thus, as long as each structure is represented by only a very small number of WMEs — one or two is okay, and perhaps three or four — T_{single} will not be too large.

Another way to bound T_{single} involves placing restrictions on representations. T_{single} can be bounded quadratic in W, independent of s and P if two conditions hold. First, working memory must form a tree (Tambe and Rosenbloom, 1994): if we treat working memory as a directed, labeled graph, viewing each WME `(id ^attr value)` as an arc `id` $\xrightarrow{\texttt{attr}}$ `value`, then this graph must be a tree. Second, all structures must be linear: if we treat structures as directed, labeled graphs in a similar way, these graphs must be linear.[4] That is, each structure tests for a sequence of WMEs lying along a single branch of the tree formed by working memory. With these restrictions on working memory and structures, T_{single} can be at most quadratic in W. This is because each token represents an (acyclic) path from one node of the tree to another, and the number of such paths in any tree is quadratic in the size of the tree.

In summary, we can keep the match cost small by keeping the total number of tokens, T_{tot}, small. To keep T_{tot} small, we want to keep both E and T_{single} small, since $T_{\text{tot}} \leq ET_{\text{single}}$. E can be kept small if the network can be viewed in terms of levels and structures such that the resulting m_i's are small, usually one. T_{single} can be kept small if either (1) structures are represented by only a very small number of WMEs, or (2) working memory forms a tree and structures are linear.

6.3.2 Applying the Model to SCA-Fixed and SCA-Random

How do we apply this model to a given system? As mentioned above, it is up to us, the modelers, to pick the "levels" and "structures," by picking the "cut points" in each rule in the system. Once we have chosen these levels and structures, if we know enough about the distribution of WMEs in working memory, we can figure out what E and T_{single} will be. Different choices of cut points can lead to different values of E and T_{single}. The important question is whether there is a way to choose these cut points on all the rules in the system so that both E and T_{single} turn out to be small.

In our SCA-Fixed example above, we can obviously choose cut points so that all the m_i's are one, hence $E = 1$. Moreover, each structure is represented by a single WME, so $s = 1$ and T_{single} is quite small. (Actually, our description of SCA-Fixed was simplified; in the real system, we would have $s = 2$, but T_{single} is still small.)

What about SCA-Random? As mentioned in Section 3.1.5, the difference between SCA-Fixed and SCA-Random is that SCA-Fixed is always consistent about which features it focuses

[4] Strictly speaking, we need two other minor restrictions here. First, the identifier field of each condition except the first one in the structure must contain a pre-bound variable — one that occurs in some previous condition. Second, the attribute field of each condition must contain a constant, not a variable.

its attention on, whereas SCA-Random focuses on a different randomly chosen subset on each training example. Consequently, in SCA-Fixed, if one rule tests the value of feature `f5`, then so do *all* the rules. In contrast, in SCA-Random, some rules will test a value of `f5`, while other rules completely ignore `f5`.

The problem this leads to in SCA-Random is that instead of having $m_i = 1$, we have $m_i = 12$. In SCA-Fixed, the dataflow at each level fans out to up to twelve nodes testing up to twelve different possible values of the *same* feature; since this feature only takes on one value at a time, the working memory distribution is such that $m_i = 1$. In SCA-Random, though, the dataflow at each level fans out to up to 144 nodes testing up to twelve different values of up to twelve *different* features. Since each of the twelve features can take on one value, there may be up to twelve "successes" at these nodes; hence, $m_i = 12$. As a result, E is very large. This is why SCA-Random slows down significantly even when left and right unlinking are added to Rete — the number of tokens (and consequently, the number of non-null activations) increases significantly, rather than remaining bounded, as more and more rules are learned.

Note that there is no way to avoid this problem by changing the order of the conditions in SCA-Random. To get $m_1 = 1$, we need there to be *one feature* that everything at level one tests a value of. But whatever feature we might try, there are some rules that completely ignore that feature. For those rules, there is no way to order their conditions to arrange for that feature to be tested at level one — no matter what ordering we try, some other feature will be tested at level one, and so m_1 will be at least two. A similar argument applies to all the other m_i's as well.

The different effects SCA-Fixed and SCA-Random have on the match cost illustrate an important, but unfortunate, fact: we cannot predict the match cost by considering *just the representation* a system uses, at least for reasonably expressive representations. SCA-Fixed and SCA-Random use exactly the same working memory representation, and learn rules having the same kinds of conditions. Yet these two systems yield very different match algorithm performance.

Applying the levels-and-structures model to a given system requires knowledge not only about the representation it uses — what WMEs will be present and what structures will be used — but also about the rules it learns and the order in which their conditions will be arranged. For a system designer who wants to predict whether match cost will become problematic as a system learns a large number of rules, this unfortunately means it may require a good deal of effort to use this model. In some cases, a designer may not be able to anticipate enough about the rules a system will learn, and thus may be unable to use this model at all. This obstacle may some day be overcome by the development of other analytical models requiring less knowledge on the designer's part; this is an important direction for future work.

We stated above that we cannot predict the match cost by considering just the representation a system uses, *for reasonably expressive representations.* There are, however, at least two cases in which we can use the levels-and-structures model without much knowledge. The trick is to pick $\ell = 1$; i.e., to treat the whole network as one level. In this case, $E = 1$, so we need only ensure that T_{single} is kept small. As above, we have two ways to do this. First, we can make sure structures are represented by only a very small number of WMEs — in other words, make sure s is very small. Of course, with just one level in the network, s is the same as C, the

number of conditions per production. So one way to ensure that the match cost stays small is to only have rules with a very small number of conditions. The second way we can keep T_{single} small is to make working memory form a tree and make all our rules linear. Unfortunately, neither of these restrictions leaves us with a very convenient representation. In most existing Soar systems, one simply cannot get by with just a handful of conditions per production, and the expressiveness of linear rules is so limited as to make encoding many domains extremely difficult.[5] The development of new programming styles or techniques may eventually alleviate these difficulties, but for the systems being written today, these restrictions are unacceptably severe.

6.3.3 Condition Ordering

Applying the levels-and-structures model requires knowledge about the order in which the conditions of each rule are arranged. We cannot do without this knowledge, since condition orderings can significantly affect the match cost. In some systems, certain orderings afford a good choice of cut points, while other orderings preclude any good choice of cut points — in other words, with "good" condition orderings, the match cost will remain small, but with "bad" orderings, it will grow linearly in the number of rules.

The developer can control condition ordering if all the rules are hand-coded, as in OPS5 production systems. However, if rules are created by a learning mechanism, their conditions must be ordered automatically by some algorithm. Unfortunately, such algorithms are currently not well-understood, and a full analysis of them is well beyond the scope of this thesis. We give just a brief discussion here.

We have both good news and bad news about condition ordering. First, the bad news. Finding the optimal ordering of the conditions in a single rule is NP-hard (Smith and Genesereth, 1986). Of course, considering only a single rule ignores sharing. Given two rules, finding the condition orderings which maximize sharing is also NP-hard.[6] So with 100,000 rules, optimizing anything is out of the question. Moreover, most existing algorithms for finding good (but not necessarily optimal) condition orderings do not take sharing into account at all — they order the conditions for one rule independently of any other rules in the system. One notable exception is (Ishida, 1988), which gives an algorithm for ordering all the conditions in a set of rules;

[5] An earlier version of the SCA system, not used in this thesis, actually did use a representation that was mostly linear. While not a *completely* linear representation, it would eliminate the linear increase in match cost now faced by SCA-Random. However, SCA's author abandoned this representation and opted for the increased ease-of-use offered by that of the current version of SCA.

[6] Consider a variation on the subgraph isomorphism construction mentioned in Section 6.1. Given two graphs, instead of encoding one as WMEs and one as a rule, we encode both graphs as rules. One graph is isomorphic to a subgraph of the other if and only if the conditions of one rule are a generalization (except for renaming variables) of those of the other rule. Thus, subgraph isomorphism can be reduced to the problem of determining whether one rule is a generalization of another rule. This in turn can be reduced to the problem of maximizing sharing, since one rule is a generalization of another if and only if *all* the Rete nodes for it can be shared with the other rule. Incidentally, this construction also shows that the related problem of determining whether two rules are isomorphic (i.e., identical except for reordering conditions and renaming variables) is equivalent to the problem of graph isomorphism, the complexity class of which is an open question.

unfortunately, the algorithm runs in time quadratic in the number of join nodes, so it is too computationally expensive to use in systems with a very large number of rules.

Now, the good news. The condition ordering algorithm used in these testbed systems — a very simple heuristic algorithm — turns out to do a "good enough" job in all of them. The algorithm, a simplified version of that described by (Scales, 1986), orders the conditions for one rule independently of any other rules. It essentially tries to minimize the cost of matching that rule alone, as if that rule were the only one present. While the exact details of the algorithm are probably unimportant,[7] there appear to be two important properties which lead it to do "well enough" in our testbeds. First, it produces an ordering of the conditions of any individual rule good enough to avoid any one rule being exorbitantly expensive to match. This was the motivation for the design of this particular ordering algorithm. Second, it is fairly consistent: given two similar productions, it gives their conditions similar orderings. For example, the match cost of any *individual rule* in SCA-Fixed does not depend on the ordering of the various feature/value tests, but with *many rules*, it is important that they be ordered consistently across rules. This was not considered in the design of the algorithm — it simply turned out to have this property. Soar's condition ordering algorithm has recently been enhanced to make it even more consistent, but we have not yet studied its impact in these testbeds. Note that these two properties may sometimes conflict: while increased consistency leads to more sharing, it can sometimes make particular rules more expensive to match.[8] Further study is needed to better understand condition ordering issues and algorithms.

6.4 Minimally Expressive Representations

In Section 6.3.2, we mentioned two cases in which we can guarantee that match cost will remain small, without requiring advance knowledge about the rules a system will learn and the order in which their conditions will be arranged. Unfortunately, in both cases, the representation is so unexpressive that it makes encoding many domains very difficult. In this section, we consider whether these cases can be broadened to make their representations less restrictive.

The first case we discussed simply required the number of conditions in each production, C, to be kept very small. Since matching a single production is NP-hard, it appears unlikely that

[7] For the curious reader — the procedure is basically a greedy algorithm. It begins by picking a condition to put first (i.e., highest in the network), then picks a condition to put second, and so on. Each time, to pick the current condition to output, it uses the following rules, in descending order of importance. (1) Prefer conditions whose identifier field contains the current goal (a Soar idiom) or a bound variable (i.e., a variable used in some condition already selected by this algorithm). (2) Prefer conditions containing the fewest unbound variables. (3) Break any ties remaining after rules (1) and (2) by using a one-step look-ahead — pick the condition which, when used as the current condition, allows the next condition to be selected using as high-priority a rule as possible. (4) Break any remaining ties deterministically (i.e., pick the first condition on the list of tied ones).

[8] At one point we tried using a more complex ordering algorithm which essentially let consistency have free rein: given a new rule, it picked the ordering of its conditions which would allow the maximum number of them to share existing join nodes. Any remaining unshared conditions were ordered using the old (usual) algorithm. The revised algorithm helped in SCA-Random: although the match cost still increased linearly, the magnitude of the increase was reduced by about a factor of three. However, the algorithm actually slowed down the matcher in two other systems.

this restriction can be relaxed very much — unless P=NP, even moderate values of C can lead to very high match cost.

The second case we discussed required working memory to be a tree and rules to be linear. Experience encoding many domains using Soar suggests that for most, the tree restriction on working memory is at least tolerable, if sometimes fairly clumsy; it is the linear restriction on rules that is often unacceptably restrictive. Consider a domain having several different features that need to be represented in working memory. With a tree-form working memory, we can either represent them using several different branches of the tree, or represent them all on one branch. In the former case, a linear rule can only test one feature, not several. This severely limits the expressiveness of rules. In the latter case, there is no way to write a rule that tests a feature "lower down" on the branch while ignoring the features "higher up" on the branch. This often forces rules to be unnecessarily specific.

In a domain with several features, we want to be able to write rules that test any combination of them we desire. We will call a representation which allows this *minimally expressive*. Are there any minimally expressive representations for which efficient matching can be guaranteed? We conjecture that there are not.

Consider what is probably the simplest minimally expressive representation possible. Working memory consists of a set of atomic symbols, without any structure to it at all. (Each symbol represents some domain feature.) Rules simply test for the presence of particular combinations of symbols in working memory. More formally, the matching problem here becomes:

Problem Definition: Matching with atomic WMEs: In the preprocessing phase, we are given P different sets $A_1, \ldots, A_P$, where each A_i is a set of symbols taken from some finite alphabet Σ ($|\Sigma| = m$). (Each A_i is like a production, testing for the presence of certain symbols in working memory.) We may preprocess these sets as we desire. (This corresponds to compiling productions into a Rete network or some other data structure.) However, we are allowed to use at most $O(D \log D)$ storage, where $D = \sum_{i=1}^{P} |A_i|$ is the size of the input data. (This precludes preprocessing into a structure requiring exorbitant space — in practice, even space quadratic in the number of productions is too much.) In the query phase, we are given a set $W \subset \Sigma$ (this specifies the symbols in working memory), and we must output the set $\{i | A_i \subset W\}$ of matching productions.

Although this is clearly an extremely simple representation, the matching problem for it is no easier than that for at least one much more complex representation. If we relax the stringent *linear* requirement on rules in our discussion above, allowing rules to be *trees*, but add the requirement that working memory use unique-attributes, we obtain a minimally expressive representation (tree-structured rules, tree-structured unique-attribute working memory) for which the matching problem is reducible to that for atomic WMEs. The details of this reduction can be found in Appendix D.

An efficient algorithm for matching with atomic WMEs would be one that takes time polynomial in W, polynomial in the size of the largest A_i, and sublinear in P. Unfortunately, there is no known algorithm for this problem guaranteed to be efficient, and we conjecture that none exists.[9]

[9]It was recently pointed out by (Naor, 1994) that this problem is equivalent to the problem of *partial match re-*

If this conjecture is true, it implies that to guarantee efficient match, we must either have advance knowledge about the rules a system will learn and the order in which their conditions will be arranged, or else we must somehow restrict the representation the system uses so that it is not minimally expressive. Unfortunately, encoding useful rules for most domains using representations so restricted is beyond the current capability of our encoding techniques.

If this all seems rather discouraging, keep in mind that this is all based on *worst-case* analyses of match cost. The empirical results of this thesis stand in marked contrast. None of our testbed systems was designed with learning a large number of rules in mind, let alone with this theoretical analysis of match cost in mind. Yet Rete/UL turns out to perform very well on six of the seven systems.

trieval in database systems (Rivest, 1976) (Knuth, 1973b, pages 558–565). An initial study of existing algorithms for partial match retrieval has not revealed any algorithms which guarantee efficient match here.

Chapter 7

Match and the Utility Problem

In the preceding chapters, we have demonstrated the effects of different match algorithms on the cost of matching rules. In this chapter, we examine the effects of different match algorithms on the overall performance of a system — the total time it takes to solve problems. As discussed in Chapter 1, in many machine learning systems — "speedup learning" systems — the purpose of learning rules is to enable the system to solve problems faster. Often, the rules are used to reduce the number of basic problem-solving steps the system requires. Each step involves using the match algorithm to determine which rules are applicable. Thus, the learned rules affect overall system performance in two ways: they affect the number of steps taken, and they affect the cost per step. These two effects are referred to respectively as the *cognitive* and *computational* effects (Tambe et al., 1990), the *search-space* and *architectural* effects (Francis and Ram, 1993), or the *indirect* and *direct* search time costs (Markovitch and Scott, 1988). If learned rules increase the cost per step by some factor, and this factor is not outweighed by a larger factor reduction in the number of steps, then the system will have a utility problem — the "learning" will make the system slower rather than faster. From the preceding chapters, it should be clear that the choice of match algorithm will affect the degree to which learned rules increase the cost per step. If this effect is sufficiently large, the match algorithm will affect whether the system has a utility problem or not. In this chapter, we demonstrate this empirically.

Of course, machine learning systems can be aimed at performance improvements other than speedup. In many, for example, the primary performance metric is classification accuracy, not run time. So for learning systems not aimed at speedup, match cost is not as crucial. This does not mean it is completely irrelevant, though. For example, a system which classifies examples perfectly is useless if it takes intolerably long to produce its output. Thus, it may be acceptable for learned rules to increase the running time of such a system mildly, but it is unacceptable for them to increase it drastically. The choice of match algorithm used in such a system may be relevant, if it drastically affects the overall running time. In this chapter, we demonstrate empirically that it can.

Note that it may seem to have been suggested in (Etzioni, 1993) that even a small increase in match cost will cause a utility problem unless the learned rules *drastically* reduce search — unless they curtail search enough to transform an exponential-time problem-solver into a polynomial-time problem-solver. This is a rather lofty goal for speedup learning systems, considering that

many AI systems attempt to solve problems which are NP-hard. In practice, it can be extremely useful to transform an exponential-time problem-solver into a faster but still exponential-time problem-solver (or, for that matter, to transform a polynomial-time problem-solver into a faster polynomial-time problem-solver). Etzioni writes:

> When η_τ [the number of search nodes expanded when the system is guided by a set τ of rules] is exponential in s [the number of literals in the description of the current state], the total overhead of matching τ is exponential in s. (Proposition 3.4, page 107)

The total overhead is exponential because the number of times the matcher is invoked is exponential — it is invoked once per node, and the number of nodes is exponential — and the cost per invocation is at least some small number of CPU instructions. This may appear to indicate that the rules are a bad thing, because they cause the system to incur an exponential overhead. But this overhead is summed over all the nodes the system expanded. So when η_τ is exponential, the *per-node* overhead may be just a constant factor. Etzioni points this out but does not elaborate. Although elsewhere in the article (page 100) Etzioni explicitly states that the rules reduce problem-solving time if and only if $(\kappa + \mu_\tau)\eta_\tau < \kappa\eta$, where κ is the cost per node without rules, μ_τ is the extra cost per node of matching rules, and η is the number of search nodes expanded in unguided search, the casual reader of Proposition 3.4 may be left with the impression that using the rules is necessarily a bad idea. Of course, if the per-node overhead is a small constant factor, and the rules reduce the number of nodes expanded by a larger constant factor, then using the rules will speed up the system. Moreover, although it will not reduce the system's asymptotic complexity to polynomial, it may reduce it to a smaller exponential. For example, if the original problem-solver explores a search space with branching factor b and depth d, then a single rule that always prunes off one possible branch at every node in the search would reduce the asymptotic complexity from b^d to $(b-1)^d$. This would be a tremendous improvement in practice, even though the total overhead of matching that rule would be exponential: $\Omega((b-1)^d)$.

A small factor increase in match cost will not cause a utility problem as long as the learned rules reduce the number of problem-solving steps by at least that same factor. In this chapter, we demonstrate that the choice of match algorithm has a tremendous impact on the factor by which match cost increases, and hence on whether a system has a utility problem or not. Our basic methodology is described in Section 7.1. The results of our experiments are presented in Section 7.2.

7.1 Methodology

The methodology for our experiments is a typical one for the evaluation of speedup learning systems. For each of our testbed systems, we created two sets of problems: a training set and a test set. We first measured the total time the system takes to solve all problems in the test set. Then, we had the system solve all the problems in the training set. After it had learned from these training problems, we again measured the time the system takes to solve all the

test problems. We then compared the two measured times for the test set, one from before learning on the training set and one from after, to see how effective the speedup learning was and whether the system suffered a utility problem. In order to examine the effect of the choice of match algorithm on the utility problem, we performed this whole process twice for each system — once with the system using the basic Rete algorithm, and once with the system using Rete/UL.

For each system, the training set consisted of the problems that were given to the system to have it learn 100,000 or more rules, as described in Section 3.1. The same problem generator was used to create a small set of test problems for each system. (See Appendix E for details.) Since the generators for the Radar, SCA-Fixed, and SCA-Random systems selected problems from a distribution *with replacement*, there may be some overlap between the test set and the training set, though we have not examined this. In the Dispatcher, Merle, and Sched systems, the test and training sets were disjoint. In the Assembler system, the test set was chosen to be a subset of the training set. This is appropriate in Assembler because Assembler's learning is specifically designed for speedup when encountering the same problem repeatedly (i.e., assembling many identical printed circuit boards); it is not intended to allow the system to assemble an unfamiliar board faster after gaining experience on some other board.

In contrast to the typical methodology for evaluating speedup learning systems, in our experiments the system was allowed to learn as it solved each test problem. This was required because at least one (and possibly more) of the testbed systems will not function properly with learning completely disabled. However, anything the system learned on one test problem was removed before it started working on the next test problem, so all the test problems were effectively run independently of each other.

7.2 Empirical Results

We now summarize the results of these experiments. For the interested reader, more details are presented in Appendix E. Table 7.1 shows, for each system, the run times obtained with the basic Rete algorithm and with Rete/UL, both before and after learning, i.e., with and without the 100,000 or more rules learned on the training set. Columns four and seven show the speedup factors from learning, using each algorithm. These speedup factors are the ratios of the time before learning to the time after learning. A "speedup factor" less than one indicates an overall slowdown due to learning — i.e., a utility problem. The last column shows the factor by which the overall system performance is improved by the use of Rete/UL instead of the basic Rete algorithm, when there are 100,000 or more rules, i.e., the ratio of column three to column six. This factor is approximately an order of magnitude in all the systems.

As the fourth column of the table shows, when the basic Rete algorithm is used, all the testbed systems suffer a utility problem, and in five of the seven cases, the problem is quite severe — they slow down by over an order of magnitude as a result of the "learning" (the speedup factors are less than 0.10). However, the seventh column shows that when Rete/UL is used, the utility problem is avoided in four of the seven systems (Assembler, Dispatcher, Merle, and Sched) — the result of the learning is now a speedup rather than a slowdown. Note that

System	Basic Rete			Rete/UL			Overall Speedup From Unlinking
	Time Before Learning	Time After Learning	Speedup Factor	Time Before Learning	Time After Learning	Speedup Factor	
Assembler	805.0	1,641.0	0.49	756.3	174.6	4.33	9.4
Dispatcher	2,330.2	3,253.0	0.72	1,949.6	574.2	3.40	5.7
Merle	1,255.5	13,278.3	0.09	958.5	624.2	1.54	21.3
Radar	40.5	1,614.1	0.03	32.4	152.9	0.21	10.6
SCA-Fixed	54.6	2,677.8	0.02	55.4	66.5	0.83	40.3
SCA-Random	56.6	2,254.3	0.03	55.0	113.0	0.49	19.9
Sched	342.7	5,530.6	0.06	276.0	213.7	1.29	25.9

Table 7.1: Summary of results of the utility problem experiments. Times are in CPU seconds. Columns four and seven show the speedup factors obtained due to learning. The last column shows the speedup obtained from using Rete/UL instead of Rete, with 100,000 rules.

these are the four systems where the purpose of the learning is in fact to speed up the system. In the other three systems (Radar, SCA-Fixed, and SCA-Random), the purpose of the learning is not to speed up the system, but to improve its classification accuracy; thus, a small slowdown is acceptable, but a drastic one is not. With unlinking, these three systems are still somewhat slower after learning, but unlinking reduces the slowdown factor by approximately an order of magnitude.

These results conclusively demonstrate that by using an improved match algorithm, we can reduce or eliminate the utility problem in all of these testbed systems. Note that our "baseline" case here is not a "straw man," but a state-of-the-art match algorithm (Rete). If, instead, we had started with a simple, straightforward match algorithm like that used in many machine learning systems, and compared the results using it with those using Rete/UL, the effect would have been even more striking than that shown in Table 7.1.

At present, most papers in the machine learning literature describing systems where high match cost is the primary (or only) cause of the utility problem do not state what algorithm the system uses for matching. Our results conclusively demonstrate that the choice of match algorithm is an important consideration in such systems. Thus, future papers on such systems should specify the match algorithms they use. Moreover, if the algorithms are not state-of-the-art, the papers should address the issue of whether incorporating an improved matcher would affect the empirical results or conclusions in the paper.

Chapter 8

Summary and Future Work

This thesis has addressed the machine learning utility problem by developing an improved production match algorithm, which enables scaling up the number of rules in production systems, and which significantly reduces or eliminates the utility problem in a broad class of machine learning systems. In this chapter, we review the major results of the thesis, and point out some interesting directions for future work.

8.1 Summary of Results and Contributions

We began by examining a diverse set of seven large learning systems and observing Rete's behavior on them. This empirical study, the first one ever performed with such systems, revealed several phenomena not found in previous studies of smaller systems. The most fundamental of these phenomena is the linearly increasing number of productions affected by WMEs. This causes existing match algorithms to slow down linearly in the number of productions in the system. Rete's sharing is one way to counter this effect, and its importance does increase with the number of productions, but Rete's sharing alone is insufficient to avoid a linear slowdown.

The problem is that sharing can only be done near the top of the network, while the sources of linear slowdown occur both at the top and the bottom of the network. We observed three such sources in our testbed systems:

- An increasing number of null right activations occurred in all seven systems, and its effect was severe in all seven. Null right activations increased in the bottom (unshared) parts of the network. They would have increased at the top of the network too, had it not been for the use of sharing.
- An increasing number of null left activations occurred in four of the seven systems, and its effect was severe in two. Null left activations increased in the bottom (unshared) parts of the network. They would have also increased at the top of the network in all seven systems, had it not been for the use of sharing.
- An increasing number of non-null activations occurred in just one of the systems, SCA-Random, where its effect was significant but not severe. Here non-null activations increased

in the bottom (unshared) parts of the network. Had it not been for the use of sharing, non-null activations would have also increased at the top of the network in all seven systems.

To avoid the two sources that we identified as responsible for the most severe slowdown — null right activations and null left activations — we introduced two optimizations for Rete, right unlinking and left unlinking. Although these two optimizations can interfere with each other, we presented a particular way of combining them which we proved minimizes the interference. This reduces the theoretical worst-case impact of null activations from a linear factor to a square-root factor; it reduced the practical impact of null activations in the testbed systems from a large linear factor to an insignificant constant factor.

Table 8.1 summarizes the various types of activations, locations in the network where they can increase, and the techniques match algorithms can use to avoid an increase. In addition to the number of activations, it is also possible for the *time per activation* to increase, causing a slowdown. This was not a major problem in any of our testbed systems, and there are already existing techniques, such as indexed memories, for addressing it.

Type of Activation	Location of Occurrence	
	Top of Beta Network	Bottom of Beta Network
Null Left	sharing or left unlinking	left unlinking
Null Right	sharing or right unlinking	right unlinking
Non-Null	sharing	?

Table 8.1: Types and locations of increasing activations, and ways to avoid them.

By incorporating our combination of right and left unlinking into the basic Rete algorithm, we obtained a new match algorithm, Rete/UL, which is a general extension to Rete. Rete/UL eliminated the linear slowdown in six of the seven testbed systems, and significantly reduced it in the remaining one. With 100,000 rules in each system, Rete/UL was approximately two orders of magnitude faster than Rete. Moreover, Rete/UL's performance scales well on a significantly broader class of systems than that of the best previously-existing match algorithms (Rete and Treat). Thus, this thesis has advanced the state-of-the-art in production match.

We also presented a theoretical analysis of the circumstances under which efficient matching can be guaranteed, and introduced a theoretical model which AI system designers can use to analyze the increase (or lack thereof) in match cost in large learning systems.

Finally, we empirically demonstrated that the use of Rete/UL significantly reduces or eliminates the utility problem in all our testbed systems. Since these testbeds represent a diverse set of domains, problem-solving techniques, authors, and research interests, they are representative of a broad class of systems. We conclude that the use of Rete/UL can significantly reduce or eliminate the utility problem in a broad class of large learning systems.

8.2 Directions for Future Work

We mentioned in Chapter 1 that our general approach to reducing match cost so as to avoid the utility problem complements two other approaches — reducing the match cost of individual

rules, and reducing the number of rules. We also mentioned that there are other possible sources of the utility problem in addition to match cost. Combining the techniques developed in this thesis with other techniques for reducing or eliminating the utility problem is an important direction for future work.

A glance at Table 8.1 immediately suggests another important direction: developing techniques to reduce or avoid an increase in the number of non-null activations in unshared parts of the network. The results from our testbed systems suggest that such an increase does not occur *pervasively* in large learning systems, but does occur *sometimes*. Thus, these techniques will probably not tremendously broaden the class of systems where match algorithm performance scales well, but may broaden it somewhat, perhaps enough for it to include certain important systems. One possibility here is to use generalized dataflow reduction techniques, as we discussed in Section 5.8.4 for Rete and in Section 5.8.5 for a modified version of Treat. Another possibility involves the use of nonlinear Rete networks, perhaps including k-ary joins; the advantages and disadvantages of such networks in large systems are not yet well-understood.

In this thesis, we have used WMEs taking the form of three-tuples. As we discussed in Section 2.11, the Rete algorithm can be used with WMEs having many other syntactic forms. The same is true of unlinking: the left and right unlinking optimizations do not depend in any way upon WMEs having any particular form. As Section 2.11 mentioned, we can transform one representation for WMEs into another; this can sometimes have the effect of shifting work from one part of the network to another. For example, if SCA-Random were reimplemented using OPS5-style WMEs, with the values of all twelve features represented on a single WME, the feature/value tests would be shifted out of the beta network and into the alpha network; instead of having a linearly increasing match cost due to increasing non-null join node activations, we would have one due to increasing constant-test node activations. This would probably change the matcher's performance by some constant factor. An interesting direction for future work, then, is to study the relative performance of the matcher when the same systems are encoded with different forms of WMEs (although the speedups this yields are unlikely to be as large as those already obtained in this thesis).

Another interesting possibility is to investigate extending Rete to perform *partial matching*, as we suggested in Section 2.11. If successful, such an extension would allow Rete and unlinking to be used not only in systems that learn rules, but also in systems that learn cases (previously-encountered problems and their solutions), where they could support efficient retrieval of relevant cases from a large case library.

Another important area needing much work is the issue of how to obtain good orderings of the conditions in each rule. As we mentioned in Section 6.3.3, this issue is not well understood, especially in the context of systems where sharing is very important. A reliable algorithm for finding good condition orderings while taking sharing into account would be another major advance in the field of production match.

In Chapter 6, we conjectured that efficient matching cannot be guaranteed for any minimally expressive representation. Proving or disproving this conjecture remains for future work. If the conjecture is true, then it will be important to develop techniques to help designers of AI systems predict whether match cost will become problematic. The levels-and-structures model is one such technique, but it requires a good deal of knowledge on the designer's part. Other techniques

might be developed to make the designer's job easier. On the other hand, if the conjecture is false — if there are minimally expressive representations which afford guaranteed efficient matching — then investigating the appropriate match algorithms for these representations will obviously be important.

Finally, this thesis has addressed scalability along the dimension of the number of rules. As we mentioned in Chapter 1, other researchers in production systems have recently developed techniques for improving the scalability of match algorithms along the dimension of the number of WMEs. An interesting possibility for future work is to try to combine the results of the two efforts, with the aim of obtaining an algorithm capable of quickly matching millions of rules against millions of working memory elements.

Appendix A

Final Pseudocode

This appendix gives pseudocode for Rete/UL. Most of this pseudocode is simply copied from appropriate places in the body of the thesis, as indicated below on each code fragment. We also include here a few details that were omitted in the body of the thesis; lines in the margin indicate the parts of the pseudocode that are changed.

We begin with the basic data structures for WMEs and tokens:

```
structure WME {copied from page 41}
    fields: array [1..3] of symbol
    alpha-mem-items: list of item-in-alpha-memory {the ones with wme=this WME}
    tokens: list of token {the ones with wme=this WME}
    negative-join-results: list of negative-join-result
end

structure token {copied from page 48}
    parent: token {points to the higher token, for items 1...i-1}
    wme: WME {gives item i}
    node: rete-node {points to the node this token is in}
    children: list of token {the ones with parent=this token}
    join-results: list of negative-join-result {used only on tokens in negative nodes}
    ncc-results: list of token {similar to join-results but for NCC nodes}
    owner: token
        {on tokens in NCC partners: token in whose local memory this result resides}
end
```

We now give the code for the alpha part of the network. The pseudocode here implements the alpha network using the exhaustive-hash-table-lookup technique described in Section 2.2.3. We need to make one minor change to this code when we use right unlinking. Each alpha memory must have a *reference-count* on it, indicating the number of join or negative nodes using it. We need this count so that when we remove productions from the network, we can tell when an alpha memory is no longer needed and should be deallocated. In Chapter 2 we did this by checking whether the *successors* field of the alpha memory was nil, but when we use right unlinking, it is possible for *successors* to be nil even when the alpha memory is still in use — if every node using it happens to be right-unlinked.

```
structure alpha-memory {revised from version on page 32}
    items: list of item-in-alpha-memory
    successors: list of rete-node
    reference-count: integer
end

structure item-in-alpha-memory {copied from page 32}
    wme: WME {the WME that's in the memory}
    amem: alpha-memory {points to the alpha memory node}
end

procedure alpha-memory-activation (node: alpha-memory, w: WME)
{copied from page 32}
    new-item ← allocate-memory()
    new-item.wme ← w; new-item.amem ← node;
    insert new-item at the head of node.items
    insert new-item at the head of w.alpha-mem-items
    for each child in node.successors do right-activation (child, w)
end

procedure add-wme (w: WME) {exhaustive hash table version}
{copied from page 17}
    let v1, v2, and v3 be the symbols in the three fields of w
    alpha-mem ← lookup-in-hash-table (v1,v2,v3)
    if alpha-mem ≠ "not-found" then alpha-memory-activation (alpha-mem, w)
    alpha-mem ← lookup-in-hash-table (v1,v2,*)
    if alpha-mem ≠ "not-found" then alpha-memory-activation (alpha-mem, w)
    alpha-mem ← lookup-in-hash-table (v1,*,v3)
    if alpha-mem ≠ "not-found" then alpha-memory-activation (alpha-mem, w)
    ⋮
    alpha-mem ← lookup-in-hash-table (*,*,*)
    if alpha-mem ≠ "not-found" then alpha-memory-activation (alpha-mem, w)
end
```

All nodes in the beta part of the network have the following fields in common:

```
structure rete-node: {copied from page 22}
    type: "beta-memory", "join-node", "p-node", etc. {or various other node types}
    children: list of rete-node
    parent: rete-node
    ...(variant part — other data depending on node type) ...
end
```

We now give the implementation of beta memory nodes. With left unlinking, we need to add an extra field containing a list of *all* the node's children, *including unlinked ones* — recall that some of the node's children will be spliced out of the *children* list. Having access to *all* the children is important so that when we are adding new productions to the network, we can find nodes to share. Without having a second list, we would be unable to find and share any currently-left-unlinked nodes.

```
structure beta-memory: {revised from version on page 22}
    items: list of token
    all-children: list of rete-node
end

procedure beta-memory-left-activation (node: beta-memory, t: token, w: WME)
{copied from page 30}
    new-token ← make-token (node, t, w)
    insert new-token at the head of node.items
    for each child in node.children do left-activation (child, new-token)
end

function make-token (node: rete-node, parent: token, w: wme)
returning token {copied from page 42}
    tok ← allocate-memory()
    tok.parent ← parent
    tok.wme ← w
    tok.node ← node {for tree-based removal}
    tok.children = nil {for tree-based removal}
    insert tok at the head of parent.children {for tree-based removal}
    if w ≠ nil then {we need this check for negative conditions}
        insert tok at the head of w.tokens {for tree-based removal}
    return tok
end
```

We now give the implementation of join nodes. The *relink-to-alpha-memory* procedure allows the use of conjunctive negations, as described in Section 4.3.2.

```
structure join-node: {copied from page 90}
    amem: alpha-memory {points to the alpha memory this node is attached to}
    tests: list of test-at-join-node
    nearest-ancestor-with-same-amem: rete-node
end

structure test-at-join-node: {copied from page 24}
    field-of-arg1: "identifier", "attribute", or "value"
    condition-number-of-arg2: integer
    field-of-arg2: "identifier", "attribute", or "value"
end

function perform-join-tests (tests: list of test-at-join-node, t: token, w: WME)
returning boolean {copied from page 25}
    for each this-test in tests do
        arg1 ← w.[this-test.field-of-arg1]
        {With list-form tokens, the following statement is really a loop}
        wme2 ← the [this-test.condition-number-of-arg2]'th element in t
        arg2 ← wme2.[this-test.field-of-arg2]
        if arg1 ≠ arg2 then return false
    return true
end

procedure join-node-left-activation (node: join-node, t: token)
{copied from page 103}
    if node.parent just became nonempty then
        relink-to-alpha-memory (node)
        if node.amem.items = nil then
            remove node from the list node.parent.children
    for each item in node.amem.items do
        if perform-join-tests (node.tests, t, item.wme) then
            for each child in node.children do left-activation (child, t, item.wme)
end

procedure join-node-right-activation (node: join-node, w: WME)
{copied from page 103}
    if node.amem just became nonempty then
        relink-to-beta-memory (node)
        if node.parent.items = nil then
            remove node from the list node.amem.successors
    for each t in node.parent.items do {"parent" is the beta memory node}
        if perform-join-tests (node.tests, t, w) then
            for each child in node.children do left-activation (child, t, w)
end
```

```
procedure relink-to-alpha-memory (node: rete-node)
{version allowing conjunctive negations} {copied from page 91}
    {follow links up from node, find first ancestor that's linked}
    ancestor ← node.nearest-ancestor-with-same-amem
    while ancestor ≠ nil  and ancestor is right-unlinked do
        ancestor ← ancestor.nearest-ancestor-with-same-amem
    {now splice in the node in the right place}
    if ancestor ≠ nil
        then insert node into the list node.amem.successors immediately before ancestor
        else insert node at the tail of the list node.amem.successors
end

procedure relink-to-beta-memory (node: join-node) {copied from page 103}
    insert node at the head of the list node.parent.children
end
```

We now give the implementation of negative nodes.

```
structure negative-join-result {copied from page 41}
    owner: token {the token in whose local memory this result resides}
    wme: WME {the WME that matches owner }
end
structure negative-node: {copied from page 91}
    {just like for a beta memory}
    items: list of token
    {just like for a join node}
    amem: alpha-memory {points to the alpha memory this node is attached to}
    tests: list of test-at-join-node
    nearest-ancestor-with-same-amem: rete-node
end
procedure negative-node-left-activation (node: negative-node, t: token, w: WME)
{copied from page 88}
    if node.items = nil then relink-to-alpha-memory (node)
    {build and store a new token, just like a beta memory would}
    new-token ← make-token (node, t, w)
    insert new-token at the head of node.items

    {compute the join results}
    new-token.join-results ← nil
    for each item in node.amem.items do
        if perform-join-tests (node.tests, new-token, item.wme) then
            jr ← allocate-memory()
            jr.owner ← new-token; jr.wme ← w
            insert jr at the head of the list new-token.join-results
            insert jr at the head of the list w.negative-join-results

    {If join results is empty, then inform children}
    if new-token.join-results=nil then
        for each child in node.children do left-activation (child, new-token, nil )
end
procedure negative-node-right-activation (node: negative-node, w: WME)
{copied from page 43}
    for each t in node.items do
        if perform-join-tests (node.tests, t, w) then
            if t.join-results=nil then delete-descendents-of-token (t)
            jr ← allocate-memory()
            jr.owner ← t; jr.wme ← w
            insert jr at the head of the list t.join-results
            insert jr at the head of the list w.negative-join-results
end
```

We now give the implementation of negated conjunctive conditions (NCC's). As discussed in Section 2.8, we assume that the *children* list on the join node for the condition preceding the NCC is *ordered* so that the NCC subnetwork gets activated *before* the NCC node.

```
structure ncc-node {copied from page 47}
    items: list of token
    partner: rete-node {points to the corresponding NCC partner node}
end

structure ncc-partner-node {copied from page 48}
    ncc-node: rete-node {points to the corresponding NCC node}
    number-of-conjuncts: integer {number of conjuncts in the NCC}
    new-result-buffer: list of token
        {results for the match the NCC node hasn't heard about}
end

procedure ncc-node-left-activation (node: ncc-node, t: token, w: WME)
{copied from page 49}
    new-token ← make-token (node, t, w) {build and store a new token}
    insert new-token at the head of node.items
    new-token.ncc-results ← nil {get initial ncc results}
    for each result in node.partner.new-result-buffer do
        remove result from node.partner.new-result-buffer
        insert result at the head of new-token.ncc-results
        result.owner ← new-token
    if new-token.ncc-results=nil then {No ncc results, so inform children}
        for each child in node.children do left-activation (child, new-token, nil )
end
```

```
procedure ncc-partner-node-left-activation (partner: rete-node, t:token, w:WME)
{copied from page 50 — see additional comments there}
    ncc-node ← partner.ncc-node
    new-result ← make-token (partner, t, w) {build a result token ⟨t, w⟩}
    {Find the appropriate owner token (into whose local memory we should put this
    result)}
    owners-t ← t; owners-w ← w
    for i = 1 to partner.number-of-conjuncts do
        owners-w ← owners-t.wme; owners-t ← owners-t.parent
    {Look for this owner in the NCC node's memory. If we find it, add new-result to its
    local memory, and propagate (deletions) to the NCC node's children.}
    if there is already a token owner in ncc-node.items with parent=owners-t
            and wme=owners-w then
        add new-result to owner.ncc-results; new-result.owner ← owner
        delete-descendents-of-token (owner)
    else
        {We didn't find an appropriate owner token already in the NCC node's memory,
        so we just stuff the result in our temporary buffer.}
        insert new-result at the head of partner.new-result-buffer
end
```

We now give the pseudocode for handling removals of WMEs.

```
procedure remove-wme (w: WME) {copied from page 102}
    for each item in w.alpha-mem-items do
        remove item from the list item.amem.items
        if item.amem.items = nil then {alpha memory just became empty}
            for each node in item.amem.successors do
                if node is a join node then {don't left-unlink negative nodes}
                    remove node from the list node.parent.children
        deallocate memory for item
    while w.tokens ≠ nil do
        delete-token-and-descendents (the first item on w.tokens)
    for each jr in w.negative-join-results do
        remove jr from the list jr.owner.join-results
        if jr.owner.join-results=nil then
            for each child in jr.owner.node.children do
                left-activation (child, jr.owner, nil )
        deallocate memory for jr
end
```

```
procedure delete-token-and-descendents (tok: token) {copied from page 87}
    while tok.children ≠ nil do
        delete-token-and-descendents (the first item on tok.children)
    if tok.node is not an NCC partner node then
        remove tok from the list tok.node.items
    if tok.wme ≠ nil then remove tok from the list tok.wme.tokens
    remove tok from the list tok.parent.children
    if tok.node is a memory node then
        if tok.node.items = nil then
            for each child in tok.node.children do
                remove child from the list child.amem.successors
    if tok.node is a negative node then
        if tok.node.items = nil then
            remove tok.node from the list tok.node.amem.successors
        for each jr in tok.join-results do
            remove jr from the list jr.w.negative-join-results
            deallocate memory for jr
    if tok.node is an NCC node then
        for each result-tok in tok.ncc-results do
            remove result-tok from the list result-tok.wme.tokens
            remove result-tok from the list result-tok.parent.children
            deallocate memory for result-tok
    if tok.node is an NCC partner node then
        remove tok from the list tok.owner.ncc-results
        if tok.owner.ncc-results = nil then
            for each child in tok.node.ncc-node.children do
                left-activation (child, tok.owner, nil )
    deallocate memory for tok
end

procedure delete-descendents-of-token (t: token) {copied from page 43}
    while t.children ≠ nil do
        delete-token-and-descendents (the first item on t.children)
end
```

Finally, we have the pseudocode for adding and removing productions. The code here includes several things omitted in the body of the thesis: creating nodes for negative conditions, creating nodes for NCC's, filling in the *nearest-ancestor-with-same-amem* fields on newly-created join and negative nodes, and unlinking newly-created nodes that have an empty memory. It also makes some minor changes we need in order to be able to remove productions when unlinking is used.

We modify the *build-or-share-alpha-memory* function so it initializes the *reference-count* on newly-created alpha memory nodes.

```
function build-or-share-alpha-memory (c: condition) {exhaustive table lookup version}
returning alpha-memory {revised from version on page 36}
    {figure out what the memory should look like}
    id-test ← nil ; attr-test ← nil ; value-test ← nil
    if a constant test t occurs in the "id" field of c then id-test ← t
    if a constant test t occurs in the "attribute" field of c then attr-test ← t
    if a constant test t occurs in the "value" field of c then value-test ← t
    {is there an existing memory like this?}
    am ← lookup-in-hash-table (id-test, attr-test, value-test)
    if am ≠ nil then return am
    {no existing memory, so make a new one}
    am ← allocate-memory()
    add am to the hash table for alpha memories
    am.successors ← nil ; am.items ← nil
    am.reference-count ← 0
    {initialize am with any current WMEs}
    for each WME w in working memory do
        if w passes all the constant tests in c then alpha-memory-activation (am ,w)
    return am
end
```

We modify the *build-or-share-beta-memory-node* function so it initializes the *all-children* field on newly-created beta memory nodes.

```
function build-or-share-beta-memory-node (parent: rete-node)
returning rete-node {revised from version on page 34}
    for each child in parent.children do {look for an existing node to share}
        if child is a beta memory node then return child
    new ← allocate-memory()
    new.type ← "beta-memory"
    new.parent ← parent; insert new at the head of the list parent.children
    new.children ← nil
    new.all-children ← nil
    new.items ← nil
    update-new-node-with-matches-from-above (new)
    return new
end
```

We make one minor change to *get-join-tests-from-condition*. We must now allow *earlier-conds* to contain negative conditions. Occurrences of variables inside negative conditions do not represent *bindings* of those variables — a negative condition tests for the *absence* of something in working memory, so there will be no binding. Thus, we modify the function so that it ignores negative conditions in *earlier-conds*.

```
function get-join-tests-from-condition (c: condition, earlier-conds: list of condition)
returning list of test-at-join-node {revised from version on page 35}
    result ← nil
    for each occurrence of a variable v in a field f of c do
        if v occurs anywhere in a positive condition in earlier-conds then
            let i be the largest i and f2 be a field such that v occurs in the f2 field of
                    the i^th condition (a positive one) in earlier-conds
            this-test ← allocate-memory()
            this-test.field-of-arg1 ← f
            this-test.condition-number-of-arg2 ← i
            this-test.field-of-arg2 ← f2
            append this-test to result
    return result
end
```

For right unlinking, we need to initialize the *nearest-ancestor-with-same-amem* fields on newly-created join and negative nodes. The *find-nearest-ancestor-with-same-amem* procedure finds the appropriate value. It starts at a given node and walks up the beta network, returning the first node it finds that uses a given alpha memory. Note that this node may be in the subnetwork for a conjunctive negation.

```
function find-nearest-ancestor-with-same-amem (node: rete-node, am: alpha-memory)
returning rete-node
    if node is the dummy top node then return nil
    if node.type = "join" or node.type = "negative" then
        if node.amem = am then return node
    if node.type = "NCC"
        then return find-nearest-ancestor-with-same-amem (node.partner.parent, am)
        else return find-nearest-ancestor-with-same-amem (node.parent, am)
end
```

When we create a new join node, we check whether either of its memories is empty; if so, we unlink it right away.

```
function build-or-share-join-node (parent: rete-node, am: alpha-memory,
                                   tests: list of test-at-join-node)
returning rete-node {revised from version on page 34}
    for each child in parent.all-children do {look for an existing node to share}
        if child is a join node and child.amem=am and child.tests=tests then
            return child
    new ← allocate-memory()
    new.type ← "join"
    new.parent ← parent; insert new at the head of the list parent.children
    insert new at the head of the list parent.all-children
    new.children ← nil
    new.tests ← tests; new.amem ← am
    insert new at the head of the list am.successors
    increment am.reference-count
    new.nearest-ancestor-with-same-amem ←
        find-nearest-ancestor-with-same-amem (parent, am)
    {Unlink right away if either memory is empty}
    if parent.items = nil then remove new from the list am.successors
    else if amem.items = nil then remove new from the list parent.children
    return new
end
```

The function for creating new negative nodes is similar to the ones for creating beta memories and join nodes. However, one additional consideration is important with negative conditions, and also with conjunctive negations. Any time there is a variable <v> which is tested in a negative condition and bound in one or more other (positive) conditions, at least one of these positive conditions must come *before* the negative condition. Recall that when we add a production to the network, the network-construction routines are given a list of its conditions *in some order*. If all conditions are positive, any order will work. Negative conditions require the aforementioned constraint on the order, though, because negative nodes need to be able to access the appropriate

variable bindings *in tokens*, and the tokens "seen" by negative nodes indicate only variable bindings from *earlier* conditions, i.e., conditions higher up in the network.

```
function build-or-share-negative-node (parent: rete-node, am: alpha-memory,
                                       tests: list of test-at-join-node)
returning rete-node
    for each child in parent.children do {look for an existing node to share}
        if child is a negative node and child.amem=am and child.tests=tests then
            return child
    new ← allocate-memory()
    new.type ← "negative"
    new.parent ← parent; insert new at the head of the list parent.children
    new.children ← nil
    new.items ← nil
    new.tests ← tests; new.amem ← am
    insert new at the head of the list am.successors
    increment am.reference-count
    new.nearest-ancestor-with-same-amem ←
        find-nearest-ancestor-with-same-amem (parent, am)
    update-new-node-with-matches-from-above (new)
    {Right-unlink the node if it has no tokens}
    if new.items = nil then remove new from the list am.successors
    return new
end
```

When adding a production that uses an NCC, we use the *build-or-share-ncc-nodes* function. Most of the work in this function is done by the helper function *build-or-share-network-for-conditions*, which builds or shares the whole subnetwork for the subconditions of the NCC. The rest of the *build-or-share-ncc-nodes* function then builds or shares the NCC and NCC partner nodes.

```
function build-or-share-ncc-nodes (parent: rete-node, c: condition {the NCC condition},
                                   earlier-conds: list of condition)
returning rete-node {returns the NCC node}
    bottom-of-subnetwork ← build-or-share-network-for-conditions (parent,
                               subconditions of c, earlier-conds)
    for each child in parent.children do {look for an existing node to share}
        if child is an NCC node and child.partner.parent=bottom-of-subnetwork
            then return child
    new ← allocate-memory(); new-partner ← allocate-memory()
    new.type ← "NCC"; new-partner.type ← "NCC-partner"
    new.parent ← parent
    insert new at the tail of the list parent.children {so the subnetwork comes first}
    new-partner.parent ← bottom-of-subnetwork
    insert new-partner at the head of the list bottom-of-subnetwork.children
    new.children ← nil ; new-partner.children ← nil
    new.partner ← new-partner; new-partner.ncc-node ← new
    new.items ← nil ; partner.new-result-buffer ← nil
    partner.number-of-conjuncts ← number of subconditions of c
    {Note: we have to inform NCC node of existing matches before informing the partner,
    otherwise lots of matches would all get mixed together in the new-result-buffer}
    update-new-node-with-matches-from-above (new)
    update-new-node-with-matches-from-above (partner)
    return new
end
```

The *build-or-share-network-for-conditions* helper function takes a list of conditions, builds or shares a network structure for them underneath the given *parent* node, and returns the lowermost node in the new-or-shared network. Note that the list of conditions can contain negative conditions or NCC's, not just positive conditions.

```
function build-or-share-network-for-conditions (parent: rete-node,
                                                conds: list of condition,
                                                earlier-conds: list of condition)
returning rete-node
    let the conds be denoted by c_1, ..., c_k
    current-node ← parent
    conds-higher-up ← earlier-conds
    for i = 1 to k do
        if c_i is positive then
            current-node ← build-or-share-beta-memory-node (current-node)
            tests = get-join-tests-from-condition (c_i, conds-higher-up)
            am ← build-or-share-alpha-memory (c_i)
            current-node ← build-or-share-join-node (current-node, am, tests)
        else if c_i is negative (but not NCC) then
            tests = get-join-tests-from-condition (c_i, conds-higher-up)
            am ← build-or-share-alpha-memory (c_i)
            current-node ← build-or-share-negative-node (current-node, am, tests)
        else {NCC's}
            current-node ← build-or-share-ncc-nodes (current-node, c_i,
                                                     conds-higher-up)
        append c_i to conds-higher-up
    return current-node
end
```

Finally, the *add-production* procedure can now be greatly simplified by having it use *build-or-share-network-for-conditions*.

```
procedure add-production (lhs: list of conditions) {revised from version on page 37}
    current-node ← build-or-share-network-for-conditions (dummy-top-node, lhs, nil )
    build a new production node, make it a child of current-node
    update-new-node-with-matches-from-above (the new production node)
end
```

We modify the *update-new-node-with-matches-from-above* procedure so it handles the case where the parent node is a negative or NCC node. One additional note: if the parent is a join node, this procedure will right-activate it. The *join-node-right-activation* procedure contains the line "if node.amem just became nonempty then ..." — depending on how this test is implemented (e.g., by checking whether the number of items in the alpha memory is exactly one), we may need to be careful to ensure that right-activations from the *update-new-node-with-matches-from-above* procedure do not make the alpha memory "look" like it just became nonempty.

```
procedure update-new-node-with-matches-from-above (new-node: rete-node)
{revised from version on page 38}
    parent ← new-node.parent
    case parent.type of
        "beta-memory":
            for each tok in parent.items do left-activation (new-node, tok)
        "join":
            saved-list-of-children ← parent.children
            parent.children ← [new-node] {list consisting of just new-node}
            for each item in parent.amem.items do
                right-activation (parent, item.wme)
            parent.children ← saved-list-of-children
        "negative":
            for each tok in parent.items do
                if tok.join-results = nil then left-activation (new-node, tok, nil )
        "NCC":
            for each tok in parent.items do
                if tok.ncc-results = nil then left-activation (new-node, tok, nil )
end
```

Finally, we modify the procedures for removing productions so they handle negative conditions and NCC's.

```
procedure remove-production (prod: production) {copied from page 38}
    delete-node-and-any-unused-ancestors (the p-node for prod)
end

procedure delete-node-and-any-unused-ancestors (node: rete-node)
{revised from version on page 39}
    {For NCC nodes, delete the partner node too}
    if node is an NCC node then
        delete-node-and-any-unused-ancestors (node.partner)

    {Clean up any tokens the node contains}
    if node is a beta memory, negative, or NCC node then
        while node.items ≠ nil do
            delete-token-and-descendents (first item on node.items)
    if node is an NCC partner node then
        while node.new-result-buffer ≠ nil do
            delete-token-and-descendents (first item on node.new-result-buffer)

    {Deal with the alpha memory}
    if node is a join or negative node then
        if node is not right-unlinked then
            remove node from the list node.amem.successors
        decrement node.amem.reference-count
        if node.amem.reference-count=0 then delete-alpha-memory (node.amem)

    {Deal with the parent}
    if node is not left-unlinked then
        remove node from the list node.parent.children
    if node is a join node then
        remove node from the list node.parent.all-children
        if node.parent.all-children=nil then
            delete-node-and-any-unused-ancestors (node.parent)
    else if node.parent.children=nil then
        delete-node-and-any-unused-ancestors (node.parent)

    deallocate memory for node
end
```

Appendix B

Static Characteristics of the Testbed Systems

This appendix presents measurements of various characteristics of each of the testbed systems. We have already discussed several important *dynamic* (run-time) characteristics of these systems in the body of the thesis. Our focus here is on *static* characteristics — characteristics which are independent of the run-time distribution of WMEs, e.g., statistics about the productions and the Rete network construction.

Table B.1 shows the number of productions, right-hand-side (RHS) actions, conditions, and Rete nodes in each testbed system, with 100,000 or more learned rules in each. The third row gives the total number of RHS actions in all rules. (We have not discussed RHS actions in this thesis, because their handling is highly system-dependent.) The fourth through sixth rows give the number of positive, negative, and conjunctive negative conditions, summed over all the rules; the seventh row gives the total number of all three types of conditions. Note that the subconditions inside conjunctive negations are not counted at all here — a conjunctive negation is counted as just one condition. (This makes little difference, since conjunctive negations are not very common in these systems, as the table shows.) The eighth row gives the number of alpha memories in the Rete network. This is particularly large in Assembler and Radar because they create many new constants ("gensym" slot identification tags or numeric timestamps) at run-time, leading to the creation of new alpha memories. The ninth through thirteenth rows give the number of beta nodes of various types (beta memories, join nodes, negative nodes, CN or CN partner nodes, and production nodes); the last row gives the total number of beta nodes in the Rete network. Note that the implementation used in this thesis sometimes merges beta memories and join nodes, as discussed in Section 2.9.2; the counts shown here are those for a network without merging, i.e., a single merged node is counted as one beta memory node plus one join node.

Table B.2 gives the same data as Table B.1, except it is normalized for the number of productions in each system, e.g., the second row gives the mean number of RHS actions per production. Two things are noteworthy here. First, the average number of conditions per production (the sixth row) ranges from 9.15 in Radar to 43.22 in Merle. As we mentioned in Section 6.1, the worst-case cost of matching a single production is exponential in the number of

	Assem.	Disp.	Merle	Radar	SCA-F.	SCA-R.	Sched
Productions	105,308	115,891	106,323	105,548	154,299	119,950	117,386
RHS actions	466,646	588,264	1,593,719	129,577	462,861	359,814	466,509
Cond's: pos.	2,260,030	2,108,656	4,425,998	943,257	3,871,885	1,780,686	3,932,271
neg.	45,139	8,010	168,552	22,697	9	9	57,899
conj. neg.	2	17	779	2	0	0	16
Total	2,305,171	2,116,683	4,595,329	965,956	3,871,894	1,780,695	3,990,186
α nodes: mem.	30,267	1,286	1,106	22,891	66	66	1,154
β nodes: mem.	1,372,918	474,956	2,132,104	453,453	484,926	328,955	2,113,667
join	1,463,140	582,185	2,213,855	558,871	638,361	445,337	2,230,962
neg.	95	6,920	49,769	22,625	8	8	965
CN or CNP	4	26	8	4	0	0	34
p-node	105,308	115,891	106,323	105,548	154,299	119,950	117,386
Total	2,941,466	1,179,979	4,502,060	1,140,502	1,277,595	894,251	4,463,015

Table B.1: Number of productions, actions, conditions, and Rete network nodes in each testbed system, with 100,000 or more learned rules.

conditions. If these systems actually exhibited this worst-case behavior, the match cost would be astronomical. This clearly demonstrates that the match cost a system incurs in practice may be much less than the theoretical worst case.[1] Second, the number of join nodes or beta memories per production is often much less than the number of conditions per production, due to sharing. For example, productions in Sched contain 33.50 positive conditions on average (row three). If there were no sharing, these conditions would be represented by 33.50 join nodes per production. But the actual number of join nodes per production is only 19.01 (row nine) — so on average, sharing saves 14.49 join nodes per production.

Table B.3 shows the static sharing factor for each type of Rete node in each system. The second row gives the static sharing factor for alpha memories, i.e., the number of alpha memories that would be needed if alpha memories were not shared between conditions, divided by the number of actual memories that actually are needed. The third through seventh rows give the static sharing factors for various types of beta nodes (beta memories, join nodes, negative nodes, CN and CN partner nodes, and production nodes). The last row gives the overall static sharing factor for beta nodes, i.e., the total number of beta nodes needed without sharing divided by the total number needed with sharing. Note that the *static* impact of sharing beta nodes is rather modest, compared with the *dynamic* impact discussed in Section 3.5. The difference arises because the static measurement gives equal weight to all nodes, while the dynamic measurement gives more weight to nodes activated more often, and many highly-shared nodes near the tops of the networks in these systems are activated quite often.

The data shown in Tables B.1, B.2, and B.3 is from the testbed systems *after learning*, i.e.,

[1] Note that this occurs despite the fact that *none* of the systems strictly adheres to the *unique attributes* restriction on working memory (Tambe, 1991, pages 36-37), since the presence of multiple goals and preferences violates this restriction; moreover, at least two of the systems (Assembler and Dispatcher) use large multi-attributes in one or more problem spaces.

	Assem.	Disp.	Merle	Radar	SCA-F.	SCA-R.	Sched
RHS actions	4.43	5.08	14.99	1.23	3.00	3.00	3.97
Cond's: pos.	21.46	18.20	41.63	8.94	25.09	14.85	33.50
neg.	0.43	0.07	1.59	0.22	0.00	0.00	0.49
conj. neg.	0.00	0.00	0.01	0.00	0.00	0.00	0.00
Total	21.89	18.26	43.22	9.15	25.09	14.85	33.99
α nodes: mem.	0.29	0.01	0.01	0.22	0.00	0.00	0.01
β nodes: mem.	13.04	4.10	20.05	4.30	3.14	2.74	18.01
join	13.89	5.02	20.82	5.29	4.14	3.71	19.01
neg.	0.00	0.06	0.47	0.21	0.00	0.00	0.01
CN or CNP	0.00	0.00	0.00	0.00	0.00	0.00	0.00
p-node	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Total	27.93	10.18	42.34	10.81	8.28	7.46	38.02

Table B.2: Number of actions, conditions, and Rete network nodes per production in each testbed system, with 100,000 or more learned rules.

	Assem.	Disp.	Merle	Radar	SCA-F.	SCA-R.	Sched
α nodes: mem.	76.16	1,645.96	4,155.61	42.20	58,665.06	26,980.23	3,457.72
β nodes: mem.	1.65	4.44	2.08	2.08	7.98	5.41	1.86
join	1.54	3.62	2.00	1.69	6.07	4.00	1.76
neg.	475.15	1.16	3.39	1.00	1.13	1.13	60.00
CN or CNP	1.00	1.31	194.75	1.00	NA	NA	1.00
p-node	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Overall	1.62	3.78	2.05	1.86	6.30	4.25	1.83

Table B.3: Static sharing factors for various types of nodes, with 100,000 or more learned rules.

with the initial rules plus 100,000 or more learned rules. Tables B.4, B.5, and B.6 give the corresponding data from the testbed systems *before learning*, i.e., with just the initial rules.

	Assem.	Disp.	Merle	Radar	SCA-F.	SCA-R.	Sched
Productions	293	1,953	624	341	48	48	418
RHS actions	1,607	9,414	1,814	1,374	108	108	1,259
Cond's: pos.	2,388	15,534	5,288	2,939	374	374	5,103
neg.	128	1,159	409	194	9	9	75
conj. neg.	2	17	3	2	0	0	16
Total	2,518	16,710	5,700	3,135	383	383	5,194
α nodes: mem.	209	918	378	266	30	30	238
β nodes: mem.	1,211	5,745	2,737	1,472	170	170	1,757
join	1,425	6,909	3,173	1,701	194	194	2,090
neg.	89	487	222	122	8	8	66
CN or CNP	4	26	6	4	0	0	34
p-node	293	1,953	624	341	48	48	418
Total	3,023	15,121	6,763	3,641	421	421	4,366

Table B.4: Number of productions, actions, conditions, and Rete network nodes in each testbed system, with just the initial productions (no learned rules).

	Assem.	Disp.	Merle	Radar	SCA-F.	SCA-R.	Sched
RHS actions	5.48	4.82	2.91	4.03	2.25	2.25	3.01
Cond's: pos.	8.15	7.95	8.47	8.62	7.79	7.79	12.21
neg.	0.44	0.59	0.66	0.57	0.19	0.19	0.18
conj. neg.	0.01	0.01	0.00	0.01	0.00	0.00	0.04
Total	8.59	8.56	9.13	9.19	7.98	7.98	12.43
α nodes: mem.	0.71	0.47	0.61	0.78	0.63	0.63	0.57
β nodes: mem.	4.13	2.94	4.39	4.32	3.54	3.54	4.20
join	4.86	3.54	5.08	4.99	4.04	4.04	5.00
neg.	0.30	0.25	0.36	0.36	0.17	0.17	0.16
CN or CNP	0.01	0.01	0.01	0.01	0.00	0.00	0.08
p-node	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Total	10.32	7.74	10.84	10.68	8.77	8.77	10.44

Table B.5: Number of actions, conditions, and Rete network nodes per production in each testbed system, with just the initial productions (no learned rules).

	Assem.	Disp.	Merle	Radar	SCA-F.	SCA-R.	Sched
α nodes: mem.	12.07	18.22	15.09	11.80	12.77	12.77	21.92
β nodes: mem.	1.98	2.71	1.93	2.00	2.20	2.20	2.93
join	1.68	2.25	1.67	1.73	1.93	1.93	2.46
neg.	1.44	2.38	1.84	1.59	1.13	1.13	1.15
CN or CNP	1.00	1.31	1.00	1.00	NA	NA	1.00
p-node	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Overall	1.82	2.40	1.81	1.86	2.03	2.03	2.57

Table B.6: Static sharing factors for various types of nodes, with just the initial productions (no learned rules).

Appendix C

Distribution of Sizes of Affect Sets

This appendix presents the distribution of the sizes of WME affect sets in the testbed systems, with 100,000 rules in each system. The mean sizes are shown in Figure 3.3 on page 72. The distribution is shown in Figure C.1. For various size ranges, it shows the percentage of WMEs whose affect set sizes fall into that range. (The scale on the horizontal axis is logarithmic.) Table C.1 shows the distribution broken down into size ranges with a finer granularity. As the table shows, the distribution varies widely across systems. In each system, it has several peaks; typically, a few peaks are at small sizes (less than 100 affected productions) and a few are at very large sizes (10,000 or more affected productions). The largest sizes (80,000–100,000 productions) are typically WMEs representing the current goal, problem space, state, or operator; such WMEs are a common Soar idiom. Although it might be possible to design special-case matching techniques to handle this idiom, this is neither necessary (since unlinking and sharing already handle it quite well) nor sufficient to avoid a linear increase in match cost (since many other WMEs also affect tens of thousands of productions).

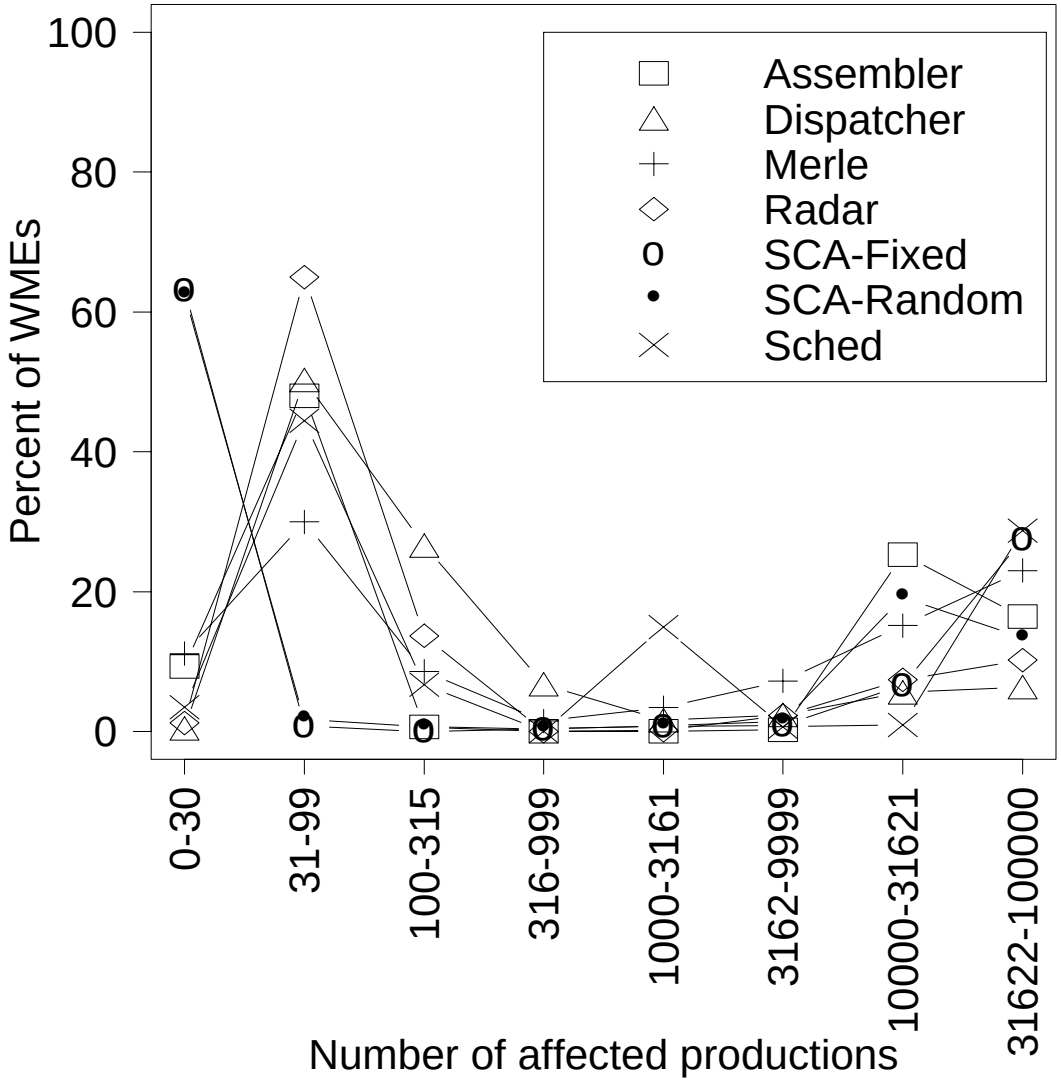


Figure C.1: Distribution of sizes of WME affect sets, with 100,000 rules in each system.

Size Range	Assem.	Disp.	Merle	Radar	SCA-F.	SCA-R.	Sched
0–9	0.16	0.23	2.49	0.62	14.98	13.97	2.16
10–19	0.16	0.21	8.66	0.62	45.34	42.53	1.28
20–29	8.99	0.00	0.00	0.00	2.80	5.90	0.00
30–39	0.00	0.00	4.63	3.56	0.37	1.01	0.00
40–49	32.50	0.00	9.98	0.00	0.00	0.00	2.83
50–59	9.78	0.00	6.20	47.68	0.00	0.68	24.71
60–69	0.91	0.00	4.45	9.60	0.00	0.00	1.77
70–79	0.16	26.06	0.96	0.71	0.00	0.00	7.92
80–89	0.11	5.24	1.27	3.44	0.37	0.00	1.28
90–99	4.57	19.01	2.49	0.00	0.00	0.00	5.95
100–199	0.64	22.00	8.22	7.51	0.00	0.00	6.47
200–299	0.00	4.60	0.37	6.18	0.00	0.70	0.22
300–399	0.00	0.30	0.20	0.00	0.37	0.00	0.08
400–499	0.00	0.18	0.05	0.00	0.00	0.00	0.01
500–599	0.00	0.05	0.08	0.02	0.00	0.00	0.00
600–699	0.00	0.07	0.00	0.00	0.00	0.36	0.00
700–799	0.00	6.16	1.13	0.00	0.00	0.00	0.00
800–899	0.00	0.00	0.03	0.00	0.00	0.00	0.00
900–999	0.00	0.00	0.00	0.00	0.00	0.00	0.00
1,000–1,999	0.01	1.52	1.27	0.07	0.37	0.80	14.90
2,000–2,999	0.01	0.07	1.95	0.11	0.38	0.00	0.06
3,000–3,999	0.01	0.00	2.32	0.48	0.00	0.00	0.00
4,000–4,999	0.22	0.00	0.88	0.25	0.00	0.00	0.00
5,000–5,999	0.01	0.17	0.81	0.00	0.41	0.95	0.00
6,000–6,999	0.01	0.00	1.86	0.34	0.00	0.00	0.00
7,000–7,999	0.01	0.00	0.56	0.99	0.46	0.00	0.00
8,000–8,999	0.01	0.00	0.91	0.21	0.00	0.00	0.00
9,000–9,999	0.01	2.21	0.05	0.00	0.00	0.52	0.68
10,000–19,999	11.83	4.31	11.86	7.40	5.47	1.54	0.00
20,000–29,999	13.49	1.22	3.30	0.00	1.17	4.52	0.95
30,000–39,999	0.00	0.10	1.56	0.00	0.48	17.33	0.00
40,000–49,999	9.20	3.06	1.74	5.27	3.33	1.10	16.01
50,000–59,999	0.00	0.14	0.99	0.44	5.04	0.00	4.20
60,000–69,999	0.00	0.23	1.74	0.18	2.31	0.00	0.00
70,000–79,999	0.00	0.96	3.12	0.00	1.11	0.00	0.00
80,000–89,999	2.49	1.89	1.78	0.00	1.04	0.00	0.00
90,000–100,000	4.73	0.00	12.07	4.33	14.18	8.09	8.51

Table C.1: Distribution of sizes of WME affect sets, with 100,000 rules in each system.

Appendix D

Reducing Unique-Attribute Trees to Atomic WMEs

In this appendix, we show that the problem of matching tree-structured productions against a tree-structured, unique-attribute working memory can be reduced to the problem of matching with atomic WMEs. Recall our definition of matching with atomic WMEs from Section 6.4: in the preprocessing phase, we are given P different sets $A_1, \ldots, A_P$, where each A_i is a set of symbols taken from some finite alphabet Σ. (Each A_i is like a production, testing for the presence of certain symbols in working memory.) In the query phase, we are given a set $W \subset \Sigma$ (this specifies the symbols in working memory), and we must output the set $\{i | A_i \subset W\}$ of matching productions.

Now consider the problem of matching tree-structured productions against a tree-structured, unique-attribute working memory. In this problem, working memory must form a tree: if we treat working memory as a directed, labeled graph, viewing each WME `(id ^attr value)` as an arc `id` $\stackrel{\texttt{attr}}{\longrightarrow}$ `value`, then this graph must be a tree. Working memory must also use unique-attributes: for each `id` and each `attr`, there is at most one `value` such that `(id ^attr value)` is in working memory. The LHS of each production must be tree-structured: if we treat LHS's as directed, labeled graphs, these graphs must be trees. More precisely, we first require the identifier field of each condition to contain a variable, and the attribute field to contain a constant; the value field may contain either. If we then view each condition `(<id> ^attr value)` as an arc `<id>` $\stackrel{\texttt{attr}}{\longrightarrow}$ `value`, and each condition `(<id> ^attr <value>)` as an arc `<id>` $\stackrel{\texttt{attr}}{\longrightarrow}$ `<value>`, the graph for each LHS must be a tree. We also require that the variable at the root of the LHS tree be restricted to match the "root" of the tree in working memory. (This restriction can be removed at a cost of an extra factor of $O(W)$ in the second phase of the reduction below — just run the atomic WME matching problem W times, each time pretending a different one of the at most W nodes in the working memory tree is the root.)

Theorem D.1 (Reducing Unique-Attribute Trees to Atomic WMEs) *The problem of matching tree-structured productions against a tree-structured, unique-attribute working memory is polynomial-time reducible to the problem of matching with atomic WMEs.*

Proof sketch: Since the LHS L of any given production forms a tree, it can be written as the union of some number of root-to-leaf paths p_j, where each p_j is a sequence of conditions

```
(<var-1> ^attr-1 <var-2>)
(<var-2> ^attr-2 <var-3>)
(<var-3> ^attr-3 <var-4>)
⋮
(<var-k> ^attr-k [last-v])
```

where `[last-v]` can either be a constant or some variable that does not occur anywhere else in the LHS. We will call these "condition-paths" to distinguish them from paths in working memory (sequences of WMEs), which we will call "WM-paths."

We claim that L has a complete match in working memory if and only if each p_j does. Clearly if L has a complete match, then each p_j does, since the conditions in p_j are a subset of those in L. If each p_j has a complete match, then the union of the WMEs matching the p_j's forms a complete match for L. (Note that the variable bindings from the matches for these different condition-paths will be consistent, due to the *unique-attribute* working memory — as explained in (Tambe et al., 1990), this restriction guarantees that each variable has at most one possible binding in WM at any given time.)

The basic idea in our reduction is to convert each LHS L into a set of root-to-leaf condition-paths, and represent each one as an atomic WME (i.e., symbol). Given a tree-structured working memory, we calculate all possible condition-paths that have a match in it, and represent each such path as an atomic WME. It follows from the above claim that this reduction is valid, i.e., it results in the correct set of matches being found.

We now give the reduction. In the preprocessing phase, we are given the LHS's L_i $(1 \leq i \leq P)$ of a set of P tree-structured productions. Initialize Σ to be empty. Express each L_i as the union of root-to-leaf condition-paths. Canonicalize the variable names in these condition-paths. For each such path p_j occurring in one or more productions, create a symbol σ_j and add it to Σ. For each LHS L_i, create the set $A_i = \{\sigma_j |$ path p_j is part of $L_i\}$. This concludes the preprocessing — we have constructed the A_i's and Σ for matching with atomic WMEs.

In the query phase, we are given a working memory WM containing |WM| WMEs. Since WM is tree-structured, it contains at most |WM| paths from the root to some other node. Each of these WM-paths is a sequence of WMEs

```
(id-1 ^attr-1 id-2)
(id-2 ^attr-2 id-3)
(id-3 ^attr-3 id-4)
⋮
(id-k ^attr-k last-v)
```

Note that there are two possible condition-paths that could match this WM-path, since the final value field in a condition-path can be either a variable or a constant:

```
(<var-1> ^attr-1 <var-2>)            (<var-1> ^attr-1 <var-2>)
(<var-2> ^attr-2 <var-3>)            (<var-2> ^attr-2 <var-3>)
(<var-3> ^attr-3 <var-4>)     and    (<var-3> ^attr-3 <var-4>)
⋮                                    ⋮
(<var-k> ^attr-k last-v)             (<var-k> ^attr-k <last-var>)
```

So for the working memory W for matching with atomic WMEs, we simply use the symbols σ_j corresponding to these two condition-paths p_j, for each of the at most $|\text{WM}|$ WM-paths in our given tree-structured working memory. □

Note that a reduction between these two problems holds in the other direction as well. Matching with atomic WMEs is trivially reducible to matching unique-attribute trees — we just represent a given set of symbols S as a one-level tree having just a root and $|S|$ leaves, each leaf labeled with a different element of S.

Appendix E

Detailed Results of Utility Problem Experiments

This appendix presents more information about the results of the utility problem experiments described in Chapter 7. Table E.1 shows, for each system, the number of problems in the training and test sets, and the number of rules it learned on the training set.[1] The training and test problems were created by random generators, as described in Sections 3.1 and 7.1. The sizes of the training sets were chosen by estimating how many problems would be required in order to have each system learn just over 100,000 rules. The sizes of the test sets were constrained by two factors. We wanted the sets to be large enough that the fastest experiments (those using Rete/UL) would take at least a few seconds, so that the granularity of the system clock would not significantly affect timing accuracy. On the other hand, we wanted the sets to be small enough that the slowest experiments (those using the basic Rete algorithm and 100,000 rules) would require only hours, not days. The test set sizes were chosen by estimating a number of problems which would satisfy these constraints.

System	Training Problems	Test Problems	Rules Learned on Training Problems
Assembler	300	10	105,015
Dispatcher	6,550	50	113,938
Merle	160	10	105,699
Radar	7,500	100	105,207
SCA-Fixed	200,000	500	154,251
SCA-Random	120,000	500	119,902
Sched	3,000	100	116,968

Table E.1: Number of problems in the training and test sets for, and number of rules learned by, each system.

[1] For the Radar system, the number of rules learned differed slightly from one run to the next, due to different random (guessed) decisions the system made when it lacked certain knowledge. In each run, the number was just over 105,200; the number in the table is from one particular run.

	Before Learning				After Learning			
System	DCs	Time per DC (msec.)	Total Time (sec.)	% Match	DCs	Time per DC (msec.)	Total Time (sec.)	% Match
Assembler	34,940	23.0	805.0	51.5	1,590	1,032.1	1,641.0	94.9
Dispatcher	46,615	50.0	2,330.2	37.7	7,782	418.0	3,253.0	84.2
Merle	49,048	25.6	1,255.5	27.7	27,730	478.8	13,278.3	95.4
Radar	4,957	8.2	40.5	37.7	5,840	276.4	1,614.1	90.9
SCA-Fixed	10,503	5.2	54.6	22.9	4,656	575.1	2,677.8	97.6
SCA-Random	10,503	5.4	56.6	23.5	8,582	262.7	2,254.3	95.6
Sched	28,765	11.9	342.7	38.3	12,388	446.4	5,530.6	96.7

Table E.2: Results of the experiments when using the basic Rete matcher.

	Before Learning				After Learning			
System	DCs	Time per DC (msec.)	Total Time (sec.)	% Match	DCs	Time per DC (msec.)	Total Time (sec.)	% Match
Assembler	34,940	21.6	756.3	44.4	1,590	109.8	174.6	55.8
Dispatcher	46,620	41.8	1,949.6	22.8	7,809	73.5	574.2	10.9
Merle	49,048	19.5	958.5	13.1	27,730	22.5	624.2	15.7
Radar	4,957	6.5	32.4	25.1	5,840	26.2	152.9	11.3
SCA-Fixed	10,503	5.3	55.4	22.8	4,656	14.3	66.5	17.8
SCA-Random	10,503	5.2	55.0	26.6	8,582	13.2	113.0	24.7
Sched	28,765	9.6	276.0	30.4	12,388	17.2	213.7	21.1

Table E.3: Results of the experiments when using Rete/UL.

Table E.2 shows the results of the experiments when the systems use the basic Rete match algorithm. The second column shows the total number of decision cycles each system required to solve all the test problems, before learning on the training set. (A decision cycle is one "basic step" in Soar's problem-solving, similar to the number of search nodes expanded in many search-based problem solvers.) The third column shows the average CPU time per decision cycle, and the fourth column shows the total CPU time required for the test problems. The fifth column shows what percentage of this total CPU time was spent in the matcher. The next four columns of the table give this same information, but for after learning on the training set — i.e., solving the test problems when using the 100,000 or more rules learned on the training set. Note that before learning, most of the systems spend only about 20–40% of their time in match, while after learning, most spend over 90% of their time in match. Thus, with the basic Rete algorithm, the cost of matching grows to dominate the overall system run time.

System	Basic Rete			Rete/UL		
	Cognitive Effect	Computational Effect	Net Effect	Cognitive Effect	Computational Effect	Net Effect
Assembler	21.97	0.02	0.49	21.97	0.20	4.33
Dispatcher	5.99	0.12	0.72	5.97	0.57	3.40
Merle	1.77	0.05	0.09	1.77	0.87	1.54
Radar	0.85	0.03	0.03	0.85	0.25	0.21
SCA-Fixed	2.26	0.00	0.02	2.26	0.37	0.83
SCA-Random	1.22	0.02	0.03	1.22	0.40	0.49
Sched	2.32	0.03	0.06	2.32	0.56	1.29

Table E.4: Summary of cognitive, computational, and net effects of learning when using different match algorithms.

Table E.3 is similar to Table E.2, but gives the results for when the systems use Rete/UL.[2] Note that in this case, even with 100,000 or more rules after learning, the match cost does not dominate the overall system run time, as the last column shows.

Table E.4 summarizes the effects of learning in these systems, when using different match algorithms. For each system and for each matcher (basic Rete and Rete/UL), it shows the cognitive, computational, and net effects of learning. The cognitive effect is the ratio of the number of decision cycles required to solve all the test problems *before learning* to the number required *after learning* — i.e., the factor by which the learning on the training set reduces the number of decision cycles required for the test set. The computational effect is the ratio of the time per decision cycle *before learning* to the time per decision cycle *after learning*. A number less than one here indicates that each decision cycle takes longer after learning than before. Finally, the net effect is the ratio of the total CPU time required to solve the test set before learning to the total CPU time required after learning. This is the overall speedup factor due to learning, and is equal to the product of the cognitive and computational effects. A number less than one here indicates that the system is slower after learning than before.

Most of the cognitive effects in Table E.4 are larger than one — learning generally reduces the number of decision cycles Soar takes to solve problems. The exception to this is Radar, where it increases the number of decision cycles slightly. Radar takes slightly more decision cycles to make a prediction when it has a relevant episode in its episodic memory than when it does not (in which case it predicts quickly by just guessing randomly). The training set causes the system to build up its episodic memory; this increases its classification accuracy, since it has to guess less often, but increases the number of decision cycles it requires on average.

The computational effects in Table E.4 are all less than one — the time per decision cycle is always larger after learning, at least in these systems. Several things can cause this:

[2]The number of decision cycles for Dispatcher is slightly different with Rete/UL than Rete. This is because Dispatcher sometimes makes decisions that depend on which of several complete production matches was found *first*. Even when they are all found in one match cycle, a sequential match algorithm always detects them in some order; changing the match algorithm can change the order. This difference could be avoided by having both matchers *sort* the matches they find each cycle.

System	Factor by which learning increases the: Total Time Per DC	Non-Match Time Per DC	Match Time Per DC	WMEs Per DC	Match Time Per WME
Assembler	5.07	4.03	6.38	1.22	5.25
Dispatcher	1.76	2.03	0.84	1.03	0.82
Merle	1.15	1.12	1.39	0.94	1.47
Radar	4.01	4.75	1.80	1.03	1.74
SCA-Fixed	2.71	2.88	2.12	1.71	1.24
SCA-Random	2.51	2.58	2.33	1.03	2.27
Sched	1.80	2.04	1.25	0.76	1.63

Table E.5: Factoring the computational effect into match and nonmatch components, and the match component into WMEs per DC and time per WME.

1. The larger number of rules in the system after learning can increase the match cost.
2. The learned rules can change the (external or internal problem-solving) behavior of the system. This can change the distribution of WMEs in working memory, thereby changing (either increasing or decreasing) the match cost.
3. The number of changes to working memory per decision cycle can change. This changes the number of times the match algorithm is invoked, thereby changing (either increasing or decreasing) the match cost per decision cycle.
4. The learned rules can change the cost of other (non-match) operations in the system.

With the basic Rete algorithm, (1) is obviously the crucial one which causes the time per decision cycle to increase dramatically after learning. What about with Rete/UL? Since our focus is on match cost, (4) is outside the scope of this thesis. We can factor this out by looking at just the time spent in the matcher per decision cycle, rather than the total time per decision cycle. The result of this is shown in Table E.5; its second, third, and fourth columns show the factor by which learning increases the total time per decision cycle, the non-match time per decision cycle, and the match time per decision cycle, respectively, when using Rete/UL. (The second column is the reciprocal of the computational effect shown in the next to last column of Table E.4.) The fifth column of Table E.5 shows the change in WME changes per decision cycle; this is effect (3) above. After factoring this out, we are left with the change in match time per change to working memory, shown in the last column of Table E.5; this is the result of effects (1) and (2).

Unfortunately, no distinct pattern emerges from this table. The computational effect is at least partly due to non-match costs in all the systems, but the magnitudes of the match and non-match components vary widely among the systems. Learning increases the number of working memory changes per decision cycle in five of the systems, but decreases it in the other two. In the Assembler system, the match cost per WME increases substantially after learning; this is probably mainly due to a change in the distribution of WMEs — if it were mainly due to the increase in the number of rules, then a similar ($\sim$5-fold) increase in match cost would probably have shown up when we held the working memory distribution fixed (see Figure 5.8

on page 112), but none did. On the other hand, in one system (Dispatcher), the match cost per WME actually *decreases* after learning; this cannot be due to effect (1) — additional rules can only increase match cost, not decrease it — so this must be due to a change in the distribution of WMEs in working memory.

Bibliography

Acharya, A. (1992). Private communication.

Acharya, A. and Tambe, M. (1992). Collection-oriented match: Scaling up the data in production systems. Technical Report CMU-CS-92-218, School of Computer Science, Carnegie Mellon University.

Allen, E. M. (1982). YAPS: Yet another production system. Technical Report CS-TR-1146, Department of Computer Science, University of Maryland.

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

Barachini, F. and Theuretzbacher, N. (1988). The challenge of real-time process control for production systems. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 705–709.

Barrett, T. (1993). Private communication.

Bouaud, J. (1993). TREE: The heuristic driven join strategy of a RETE-like matcher. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence*, pages 496–502.

Brownston, L., Farrell, R., Kant, E., and Martin, N. (1985). *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, Reading, MA.

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

Cohen, W. W. (1990). Learning approximate control rules of high utility. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 268–276.

Cooper, T. and Wogrin, N. (1988). *Rule-Based Programming with OPS5.* Morgan Kaufmann, San Mateo, CA.

Cruise, A., Ennis, R., Finkel, A., Hellerstein, J., Klein, D., Loeb, D., Masullo, M., Milliken, K., Van Woerkom, H., and Waite, N. (1987). YES/L1: Integrating rule-based, procedural, and real-time programming for industrial applications. In *Proceedings of the Third Conference on Artificial Intelligence Applications*, pages 134–139.

DeJong, G. and Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176.

Doorenbos, R., Tambe, M., and Newell, A. (1992). Learning 10,000 chunks: What's it like out there? In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 830–836.

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

Doorenbos, R. B. (1994). Combining left and right unlinking for matching a large number of learned rules. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 451–458.

Doorenbos, R. B. and Veloso, M. M. (1993). Knowledge organization and the utility problem. In *Proceedings of the Third International Workshop on Knowledge Compilation and Speedup Learning*, pages 28–34.

Etzioni, O. (1990). *A Structural Theory of Search Control.* PhD thesis, School of Computer Science, Carnegie Mellon University.

Etzioni, O. (1993). A structural theory of explanation-based learning. *Artificial Intelligence*, 60(1):93–139.

Forgy, C. L. (1979). *On the Efficient Implementation of Production Systems.* PhD thesis, Computer Science Department, Carnegie Mellon University.

Forgy, C. L. (1981). OPS5 user's manual. Technical Report CMU-CS-81-135, Computer Science Department, Carnegie Mellon University.

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

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

Francis, Jr., A. G. and Ram, A. (1993). Computational models of the utility problem and their application to a utility analysis of case-based reasoning. In *Proceedings of the Third International Workshop on Knowledge Compilation and Speedup Learning*, pages 48–55.

Ghallab, M. (1981). Decision trees for optimizing pattern-matching algorithms in production systems. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 310–312.

Gratch, J. and DeJong, G. (1992). COMPOSER: A probabilistic solution to the utility problem in speed-up learning. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 235–240.

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

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

Gupta, A., Forgy, C., and Newell, A. (1989). High-speed implementations of rule-based systems. *ACM Transactions on Computer Systems*, 7(2):119–146.

Gupta, A., Tambe, M., Kalp, D., Forgy, C., and Newell, A. (1988). Parallel implementation of OPS5 on the Encore multiprocessor: Results and analysis. *International Journal of Parallel Programming*, 17(2):95–124.

Hanson, E. N. (1993). Gator: A discrimination network structure for active database rule condition matching. Technical Report UF-CIS-TR-93-009, CIS Department, University of Florida.

Hanson, E. N. and Widom, J. (1993). Rule processing in active database systems. *International Journal of Expert Systems*, 6(1):83–119.

Hayes-Roth, F. and Mostow, D. J. (1975). An automatically compilable recognition network for structured patterns. In *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, pages 246–251.

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

Ishida, T. (1988). Optimizing rules in production system programs. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 699–704.

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

Knuth, D. E. (1973a). *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley, Reading, MA, second edition.

Knuth, D. E. (1973b). *The Art of Computer Programming, Volume 3: Sorting and Searching.* Addison-Wesley, Reading, MA.

Kuo, S. and Moldovan, D. (1992). The state of the art in parallel production systems. *Journal of Parallel and Distributed Computing*, 15:1–26.

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

Laird, J. E., Rosenbloom, P. S., and Newell, A. (1986). Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46.

Lee, H. S. and Schor, M. I. (1992). Match algorithms for generalized Rete networks. *Artificial Intelligence*, 54:249–274.

Lichtman, Z. L. and Chester, D. L. (1989). A family of cuts for production systems. In *Proceedings of the IEEE International Workshop on Tools for Artificial Intelligence*, pages 353–357.

Liu, R.-L. and Soo, V.-W. (1992). Augmenting and efficiently utilizing domain theory in explanation-based natural language acquisition. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 282–289.

Markovitch, S. and Scott, P. D. (1988). The role of forgetting in learning. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 459–465.

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

McDermott, J. and Forgy, C. (1978). Production system conflict resolution strategies. In Waterman, D. A. and Hayes-Roth, F., editors, *Pattern-Directed Inference Systems*, pages 177–199. Academic Press, New York.

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

Mertz, J. (1995). *Using a Cognitive Architecture to Design Instructions*. PhD thesis, Department of Engineering and Public Policy, Carnegie Mellon University.

Miller, C. S. (1993). *Modeling Concept Acquisition in the Context of a Unified Theory of Cognition*. PhD thesis, Department of Computer Science and Engineering, University of Michigan.

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

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

Minton, S. (1988b). Quantitative results concerning the utility of explanation-based learning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 564–569.

Minton, S., Carbonell, J. G., Knoblock, C. A., Kuokka, D. R., Etzioni, O., and Gil, Y. (1989). Explanation-based learning: A problem-solving perspective. *Artificial Intelligence*, 40(1-3).

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

Miranker, D. P., Brant, D. A., Lofaso, B., and Gadbois, D. (1990). On the performance of lazy matching in production systems. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 685–692.

Miranker, D. P., Burke, F., Kolts, J., and Steele, J. J. (1991). The C++ embeddable rule system. In *Proceedings of the Third International Conference on Tools for Artificial Intelligence*, pages 386–393.

Miranker, D. P. and Lofaso, B. J. (1991). The organization and performance of a TREAT-based production system compiler. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):3–10.

Mitchell, T. M., Keller, R. M., and Kedar-Cabelli, S. T. (1986). Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80.

Mooney, R. (1989). The effect of rule use on the utility of explanation-based learning. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 725–730.

Mostow, D. J. (1994). Private communication.

Naor, M. (1994). Private communication to Steven Rudich.

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

Nishiyama, S. (1991). Optimizing compilation of select phase of production systems. Master's thesis, University of Texas at Austin.

Oflazer, K. (1984). Partitioning in parallel processing of production systems. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 92–100.

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

Pasik, A. J., Miranker, D. P., Stolfo, S. J., and Kresnicka, T. (1989). User-defined predicates in OPS5: A needed language extension for financial expert systems. Technical Report CUCS-496-89, Columbia University.

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

Perlin, M. (1990a). Scaffolding the Rete network. In *Proceedings of the Second IEEE International Conference on Tools for Artificial Intelligence*, pages 378–385.

Perlin, M. (1990b). Topologically traversing the Rete network. *Applied Artificial Intelligence*, 4(3):155–177.

Perlin, M. (1991a). Incremental binding-space match: The linearized MatchBox algorithm. In *Proceedings of the Third International Conference on Tools for Artificial Intelligence*, pages 468–477.

Perlin, M. (1992). Constraint satisfaction for production system match. In *Proceedings of the Fourth IEEE International Conference on Tools with Artificial Intelligence*, pages 28–35.

Perlin, M. and Debaud, J.-M. (1989). Match box: fine-grained parallelism at the match level. In *Proceedings of the IEEE International Workshop on Tools for Artificial Intelligence*, pages 428–434.

Perlin, M. W. (1988). Reducing computation by unifying inference with user interface. Technical Report CMU-CS-88-150, School of Computer Science, Carnegie Mellon University.

Perlin, M. W. (1991b). *Automating the Construction of Efficient Artificial Intelligence Algorithms*. PhD thesis, School of Computer Science, Carnegie Mellon University.

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

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1):81–106.

Rivest, R. L. (1976). Partial-match retrieval algorithms. *SIAM Journal on Computing*, 5(1):19–50.

Rosenbloom, P. S. (1994). Private communication.

Rosenbloom, P. S. and Aasman, J. (1990). Knowledge level and inductive uses of chunking (EBL). In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 821–827.

Rosenbloom, P. S. and Laird, J. E. (1986). Mapping explanation-based generalization onto Soar. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 561–567.

Rosenbloom, P. S., Laird, J. E., Newell, A., and McCarl, R. (1991). A preliminary analysis of the Soar architecture as a basis for general intelligence. *Artificial Intelligence*, 47(1-3):289–325.

Samuelsson, C. and Rayner, M. (1991). Quantitative evaluation of explanation-based learning as an optimization tool for a large-scale natural language system. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 609–615.

Scales, D. J. (1986). Efficient matching algorithms for the Soar/Ops5 production system. Technical Report KSL-86-47, Knowledge Systems Laboratory, Department of Computer Science, Stanford University.

Schor, M. I., Daly, T. P., Lee, H. S., and Tibbitts, B. R. (1986). Advances in Rete pattern matching. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, pages 226–232.

Sellis, T., Lin, C.-C., and Raschid, L. (1988). Implementing large production systems in a DBMS environment: Concepts and algorithms. In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 404–412.

Smith, D. E. and Genesereth, M. R. (1986). Ordering conjunctive queries. *Artificial Intelligence*, 26:171–215.

Stolfo, S. J. (1993). Private communication.

Stolfo, S. J. and Miranker, D. P. (1986). The DADO production system machine. *Journal of Parallel and Distributed Computing*, 3:269–296.

Stolfo, S. J., Wolfson, O., Chan, P. K., Dewan, H. M., Woodbury, L., Glazier, J. S., and Ohsie, D. A. (1991). PARULEL: Parallel rule processing using meta-rules for redaction. *Journal of Parallel and Distributed Computing*, 13(4):366–382.

Tambe, M. (1991). *Eliminating Combinatorics from Production Match.* PhD thesis, School of Computer Science, Carnegie Mellon University.

Tambe, M., Kalp, D., Gupta, A., Forgy, C., Milnes, B., and Newell, A. (1988). Soar/PSM-E: Investigating match parallelism in a learning production system. In *Proceedings of the ACM/SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 146–160.

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

Tambe, M., Kalp, D., and Rosenbloom, P. S. (1992). An efficient algorithm for production systems with linear-time match. In *Proceedings of the Fourth IEEE International Conference on Tools with Artificial Intelligence*, pages 36–43.

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

Tambe, M. and Rosenbloom, P. S. (1994). Investigating production system representations for non-combinatorial match. *Artificial Intelligence*, 68(1):155–199.

Tzvieli, A. and Cunningham, S. J. (1989). Super-imposing a network structure on a production system to improve performance. In *Proceedings of the IEEE International Workshop on Tools for Artificial Intelligence*, pages 345–352.

Veloso, M. M. (1992). *Learning by Analogical Reasoning in General Problem Solving.* PhD thesis, School of Computer Science, Carnegie Mellon University.

Wogulis, J. and Langley, P. (1989). Improving efficiency by learning intermediate concepts. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 657–662.

Yost, G. R. (1992). *TAQL: A Problem Space Tool for Expert System Development.* PhD thesis, School of Computer Science, Carnegie Mellon University.

Yost, G. R. and Newell, A. (1989). A problem space approach to expert system specification. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 621–627.