Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem*

Charles L. Forgy

Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213, U.S.A.

Recommended by Harry Barrow

ABSTRACT

The Rete Match Algorithm is an efficient method for comparing a large collection of patterns to a large collection of objects. It finds all the objects that match each pattern. The algorithm was developed for use in production system interpreters, and it has been used for systems containing from a few hundred to more than a thousand patterns and objects. This article presents the algorithm in detail. It explains the basic concepts of the algorithm, it describes pattern and object representations that are appropriate for the algorithm, and it describes the operations performed by the pattern matcher.

1. Introduction

In many pattern/many object pattern matching, a collection of patterns is compared to a collection of objects, and all the matches are determined. That is, the pattern matcher finds every object that matches each pattern. This kind of pattern matching is used extensively in Artificial Intelligence programs today. For instance, it is a basic component of production system interpreters. The interpreters use it to determine which productions have satisfied condition parts. Unfortunately, it can be slow when large numbers of patterns or objects are involved. Some systems have been observed to spend more than nine-tenths of their total run time performing this kind of pattern matching [5]. This

*This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

article describes an algorithm that was designed to make many pattern/many object pattern matching less expensive. The algorithm was developed for use in production system interpreters, but since it should be useful for other languages and systems as well, it is presented in detail.

This article attends to two complementary aspects of efficiency: (1) designing an algorithm for the task and (2) implementing the algorithm on the computer. The rest of Section 1 provides some background information. Section 2 presents the basic concepts of the algorithm. Section 3 explains how the objects and patterns should be represented to allow the most efficient implementations. Section 4 describes in detail a very fast implementation of the algorithm. Finally, Section 5 presents some of the results of the analyses of the algorithm.

1.1. OPS5

The methods described in this article were developed for production system interpreters, and they will be illustrated with examples drawn from production systems. This section provides a brief introduction to the language used in the examples, OPS5. For a more complete description of OPS5, see [6].

A production system program consists of an unordered collection of If-Then statements called *productions*. The data operated on by the productions is held in a global data base called *working memory*. By convention, the If part of a production is called its *LHS* (left-hand side), and its Then part is called its *RHS* (right-hand side). The interpreter executes a production system by performing the following operations.

(1) *Match.* Evaluate the LHSs of the productions to determine which are satisfied given the current contents of working memory.
(2) *Conflict resolution.* Select one production with a satisfied LHS; if no productions have satisfied LHSs, halt the interpreter.
(3) *Act.* Perform the actions in the RHS of the selected production.
(4) Goto 1.

OPS5 working memories typically contain several hundred objects, and each object typically has between ten and one hundred associated attribute-value pairs. An object together with its attribute-value pairs is called a *working memory element.* The following is a typical, though very small, OPS5 working memory element; it indicates that the object of class Expression which is named Expr17 has 2 as its first argument, '*' as its operator, and X as its second argument.

(Expression ↑Name Expr17 ↑Arg1 2 ↑Op * ↑Arg2 X)

The ↑ is the OPS5 operator that distinguishes attributes from values.

The LHS of a production consists of a sequence of patterns; that is, a sequence of partial descriptions of working memory elements. When a pattern P describes an element E, P is said to *match* E. In some productions, some of

the patterns are preceded by the negation symbol, –. An LHS is satisfied when

(1) Every pattern that is not preceded by – matches a working memory element, and

(2) No pattern that is preceded by – matches a working memory element.

The simplest patterns contain only constant symbols and numbers. A pattern containing only constants matches a working memory element if every constant in the pattern occurs in the corresponding position in the working memory element. (Since patterns are partial descriptions, it is not necessary for every constant in the working memory element to occur in the pattern.) Thus the pattern

(Expression ↑ Op * ↑ Arg2 0)

would match the element

(Expression ↑ Name Expr86 ↑ Arg1 X ↑ Op * ↑ Arg2 0)

Many non-constant symbols are available in OPS5 for defining patterns, but the two most important are variables and predicates. A variable is a symbol that begins with the character '⟨' and ends with the character '⟩'—for example ⟨X⟩. A variable in a pattern will match any value in a working memory element, but if a variable occurs more than once in a production's LHS, all occurrences must match the same value. Thus the pattern

(Expression ↑ Arg1 ⟨VAL⟩ ↑ Arg2 ⟨VAL⟩)

would match either of the following

(Expression ↑ Name Expr9 ↑ Arg1 Expr23 ↑ Op * ↑ Arg2 Expr23)

(Expression ↑ Name Expr5 ↑ Arg1 0 ↑ Op – ↑ Arg2 0)

but it would not match

(Expression ↑ Name Expr8 ↑ Arg1 0 ↑ Op * ↑ Arg2 Expr23)

The predicates in OPS5 include = (equal), <> (not equal), < (less than), > (greater than), <= (less than or equal), and >= (greater than or equal). A predicate is placed between an attribute and a value to indicate that the value matched must be related in that way to the value in the pattern. For instance,

(Expression ↑ Op <>*)

will match any expression whose operand is not *. Predicates can be used with variables as well as with constant values. For example, the following pattern

(Expression ↑ Arg1 ⟨LEFT⟩ ↑ Arg2 <> ⟨LEFT⟩)

will match any expression in which the first argument differs from the second argument.

The RHS of a production consists of an unconditional sequence of actions. The only actions that need to be described here are the ones that change working memory. MAKE builds a new element and adds it to working memory. The argument to MAKE is a pattern like the patterns in LHSs. For example,

```
(MAKE Expression   ↑ Name Expr1   ↑ Arg1 1)
```

will build an expression whose name is Expr1, whose first argument is 1, and whose other attributes all have the value NIL (the default value in OPS5). MODIFY changes one or more values of an existing element. This action takes as arguments a pattern designator and a list of attribute-value pairs. The following action, for example

```
(MODIFY 2   ↑ Op NIL   ↑ Arg2 NIL)
```

would take the expression matching the second pattern and change its operator and second argument to NIL. The action REMOVE deletes elements from working memory. It takes pattern designators as arguments. For example

```
(REMOVE 1 2 3)
```

would delete the elements matching the first three patterns in a production.

An OPS5 production consists of (1) the symbol P, (2) the name of the production, (3) the LHS, (4) the symbol -->, and (5) the RHS, with everything enclosed in parentheses. The following is a typical production.

```
(P Time 0x
    (Goal   ↑ Type Simplify   ↑ Object ⟨X⟩)
    (Expression   ↑ Name ⟨X⟩   ↑ Arg1 0   ↑ Op *)
-->
    (MODIFY 2   ↑ Op NIL   ↑ Arg2 NIL))
```

1.2. Work on production system efficiency

Since execution speed has always been a major issue for production systems, several researchers have worked on the problem of efficiency. The most common approach has been to combine a process called *indexing* with direct interpretation of the LHSs. In the simplest form of indexing, the interpreter begins the match process by extracting one or more features from each working memory element, and uses those features to hash into the collection of productions. This produces a set of productions that might have satisfied LHSs. The interpreter examines each LHS in this set individually to determine whether it is in fact satisfied. A more efficient form of indexing adds memory to the process. A typical scheme involves storing a count with each pattern. The counts are all zero when execution of the system begins. When an element enters working memory, the indexing function is executed with the new

element as its only input, and all the patterns that are reached have their counts increased by one. When an element leaves working memory, the index is again executed, and the patterns that are reached have their counts decreased by one. The interpreter performs the direct interpretation step only on those LHSs that have non-zero counts for all their patterns. Interpreters using this scheme—in some cases combined with other efficiency measures—have been described by McCracken [8], McDermott, Newell, and Moore [9], and Rychener [10].

The algorithm that will be presented here, the Rete Match Algorithm, can be described as an indexing scheme that does not require the interpretive step. The indexing function is represented as a network of simple feature recognizers. This representation is related to the graph representations for so-called structured patterns. (See for example [2] and [7]). The Rete algorithm was first described in 1974 [3]. A 1977 paper [4] described some rather complex interpreters for the networks of feature recognizers, including parallel interpreters and interpreters which delayed evaluation of patterns as long as possible. (Delaying evaluation is useful because it makes it less likely that patterns will be evaluated unnecessarily.) A 1979 paper [5] discussed simple but very fast interpreters for the networks. This article is based in large part on the 1979 paper.

2. The Rete Match Algorithm—Basic Concepts

In a production system interpreter, the output of the match process and the input to conflict resolution is a set called the *conflict set.* The conflict set is a collection of ordered pairs of the form

⟨Production, List of elements matched by its LHS⟩

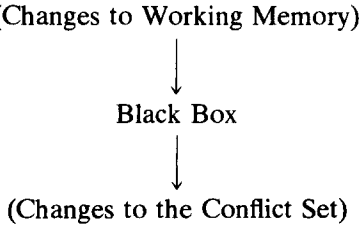
The ordered pairs are called *instantiations.* The Rete Match Algorithm is an algorithm for computing the conflict set. That is, it is an algorithm to compare a set of LHSs to a set of elements in order to discover all the instantiations. The algorithm can efficiently process large sets because it does not iterate over the sets.

2.1. How to avoid iterating over working memory

A pattern matcher can avoid iterating over the elements in working memory by storing information between cycles. The step that can require iteration is determining whether a given pattern matches any of the working memory elements. The simplest interpreters determine this by comparing the pattern to the elements one by one. The iteration can be avoided by storing, with each pattern, a list of the elements that it matches. The lists are updated when working memory changes. When an element enters working memory, the interpreter finds all the patterns that match it and adds it to their lists. When an

element leaves working memory, the interpreter again finds all the patterns that match it and deletes it from their lists.

Since pattern matchers using the Rete algorithm save this kind of information, they never have to examine working memory. The pattern matcher can be viewed as a black box with one input and one output.



The box receives information about the changes that are made to working memory, and it determines the changes that must be made in the conflict set to keep it consistent. For example, the black box might be told that the element

(Goal ↑ Type Simplify ↑ Object Expr19)

has been added to working memory, and it might respond that production TimexN has just become instantiated.

2.1.1. *Tokens*

The descriptions of working memory changes that are passed into the black box are called *tokens*. A token is an ordered pair of a *tag* and a list of data elements. In the simplest implementations of the Rete Match Algorithm, only two tags are needed, + and −. The tag + indicates that something has been added to working memory. The tag − indicates that something has been deleted from working memory. When an element is modified, two tokens are sent to the black box; one token indicates that the old form of the element has been deleted from working memory, and the other that the new form of the element has been added. For example, if

(Expression ↑ Name Expr41 ↑ Arg1 Y ↑ Op + ↑ Arg2 Y)

was changed to

(Expression ↑ Name Expr41 ↑ Arg1 2 ↑ Op * ↑ Arg2 Y)

the following two tokens would be processed.

⟨−(Expression ↑ Name Expr41 ↑ Arg1 Y ↑ Op + ↑ Arg2 Y)⟩
⟨+(Expression ↑ Name Expr41 ↑ Arg1 2 ↑ Op * ↑ Arg2 Y)⟩

2.2. How to avoid iterating over production memory

The Rete algorithm avoids iterating over the set of productions by using a tree-structured sorting network or index for the productions. The network,

which is compiled from the patterns, is the principal component of the black box. The following sections explain how patterns are compiled into networks and how the networks perform the functions of the black box.

2.2.1. *Compiling the patterns*

When a pattern matcher processes a working memory element, it tests many features of the element. The features can be divided into two classes. The first class, which could be called the intra-element features, are the ones that involve only one working memory element. For an example of these features, consider the following pattern.

(Expression ↑ Name ⟨N⟩ ↑ Arg1 0 ↑ Op + ↑ Arg2 ⟨X⟩)

When the pattern matcher processes this pattern, it tries to find working memory elements having the following intra-element features.

- The class of the element must be Expression.
- The value of the Arg1 attribute must be the number 0.
- The value of the Op attribute must be the atom +.

The other class of features, the inter-element features, results from having a variable occur in more than one pattern. Consider Plus0x's LHS.

```
(P Plus0x
    (Goal ↑ Type Simplify ↑ Object ⟨N⟩)
    (Expression ↑ Name ⟨N⟩ ↑ Arg1 0 ↑ Op + ↑ Arg2 ⟨X⟩)
-->···)
```

The intra-element features for the second pattern are listed above. A similar list can be constructed for the first pattern. But in addition to those two lists, the following inter-element feature is necessary because the variable ⟨N⟩ occurs twice.

- The value of the Object attribute of the goal must be equal to the value of the Name attribute of the expression.

The pattern compiler builds a network by linking together nodes which test elements for these features. When the compiler processes an LHS, it begins with the intra-element features. It determines the intra-element features that each pattern requires and builds a linear sequence of nodes for the pattern. Each node tests for the presence of one feature. After the compiler finishes with the intra-element features, it builds nodes to test for the inter-element features. Each of the nodes has two inputs so that it can join two paths in the network into one. The first of the two-input nodes joins the linear sequences for the first two patterns, the second two-input nodes joins the output of the first with the sequence for the third pattern, and so on. The two-input nodes test every inter-element feature that applies to the elements they process. Finally, after the two-input nodes, the compiler builds a special terminal node to represent the production. This node is attached to the last of the two-input

nodes. Fig. 1 shows the network for Plus0x and the similar production Time0x. Note that when two LHSs require identical nodes, the compiler shares parts of the network rather than building duplicate nodes.

2.2.2. *Processing in the network*

The root node of the network (at the top in Fig. 1) is the input to the black box. This node receives the tokens that are sent to the black box and passes copies of the tokens to all its successors. The successors of the top node, the nodes to perform the intra-element tests, have one input and one or more outputs. Each node tests one feature and sends the tokens that pass the test to its successors. The two-input nodes compare tokens from different paths and join them into bigger tokens if they satisfy the inter-element constraints of the LHS. Because of the tests performed by the other nodes, a terminal node will receive only tokens that instantiate the LHS. The terminal node sends out of the black box the information that the conflict set must be changed.

For an example of the operation of the nodes, consider what happens in the network in Fig. 1 when the following two elements are put into an empty working memory.

(Goal ↑ Type Simplify ↑ Object Expr17)
(Expression ↑ Name Expr17 ↑ Arg1 0 ↑ Op * ↑ Arg2 X)

First the token

⟨+(Goal ↑ Type Simplify ↑ Object Expr17)⟩

is created and sent to the root of the network. This node sends the token to its successors. One of the successors (on the right in Fig. 1) tests it and rejects it because its class is not Expression. This node does not pass the token to its successor. The other successor of the top node accepts the token (because its class is Goal) and so sends it to its successor. That node also accepts the token (since its type is Simplify), and it sends the token to its successors, the two-input nodes. Since no other tokens have arrived at the two-input nodes, they can perform no tests; they must just store the token and wait.

When the token

⟨+(Expression ↑ Name Expr17 ↑ Arg1 0 ↑ Op * ↑ Arg2 X)⟩

is processed, it is tested by the one-input nodes and passed down to the right input of Time0x's two-input node. This node compares the new token to the earlier one, and finding that they allow the variable to be bound consistently, it creates and sends out the token

⟨+(Goal ↑ Type Simplify ↑ Object Expr17)
(Expression ↑ Name Expr17 ↑ Arg1 0 ↑ Op * ↑ Arg2 X)⟩

```
(P Plus0x
        (Goal   ↑Type Simplify   ↑Object ⟨N⟩)
        (Expression   ↑Name ⟨N⟩   ↑Arg1 0   ↑Op +   ↑Arg2 ⟨X⟩)
-->     ...)

(P Time0x
        (Goal   ↑Type Simplify   ↑Object ⟨N⟩)
        (Expression   ↑Name ⟨N⟩   ↑Arg1 0   ↑Op *   ↑Arg2 ⟨X⟩)
-->     ...)
```

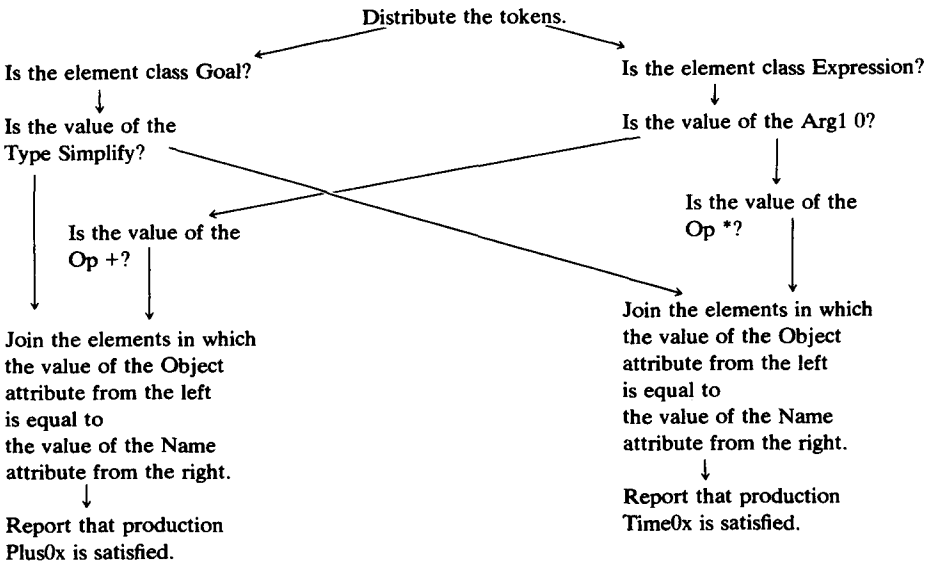


FIG. 1. The network for Plus0x and Time0x.

When its successor, the terminal node for Time0x, receives this token, it adds the instantiation of Time0x to the conflict set.

2.2.3. *Saving information in the network*

As explained above, the black box must maintain state information because it must know what is in working memory. In simple Rete networks all such state is stored by the two-input nodes. Each two-input node contains two lists called its left and right memories. The left memory holds copies of the tokens that arrived at its left input, and the right memory holds copies of the tokens that arrived at its right input. The tokens are stored as long as they are useful. The next section explains how the nodes determine when the tokens are no longer useful.

2.2.4. *Using the tags*

The tag in a token indicates how the state information is to be changed when the token is processed. The + and − tokens are processed identically except:

- The terminal nodes use the tags to determine whether to add an instantiation to the conflict set or to remove an existing instantiation. When a + token is processed, an instantiation is added; when a − token is processed, an instantiation is removed.
- The two-input nodes use the tags to determine how to modify their internal memories. When a + token is processed, it is stored in the internal memory; when a − token is processed, a token with an identical data part is deleted.
- The two-input nodes use the tags to determine the appropriate tags for the tokens they build. When a new output is created, it is given the tag of the token that just arrived at the two-input node.

2.3. Completing the set of node types

The network in Fig. 1 contained four kinds of nodes: the root node, the terminal nodes, the one-input nodes, and the two-input nodes. Certainly one could define many more kinds of nodes, but only a few more are necessary to have a complete and useful set. In fact, only two more kinds of nodes are necessary to interpret OPS5.

A second kind of two-input node is needed for negated patterns (that is, patterns preceded by −). The new two-input node stores a count with each token in its left memory. The count indicates the number of tokens in the right memory that allow consistent variable bindings. The tokens in its right memory contain the elements that match the negated pattern—or, more precisely, the tokens contain the elements that have the intra-element features that the negated pattern requires. The node allows the tokens with a count of zero to pass.

The last node type that needs to be defined is a variant of the one-input nodes described earlier. Those nodes tested working memory elements for constant features (testing, for example, whether a value was equal to a given atomic symbol). The new one-input nodes compare two values from a working memory element. These nodes are used to process patterns that contain two or more occurrences of a variable. The following, for example, would require one of these nodes because ⟨X⟩ occurs twice.

(Expression ↑ Arg1 ⟨X⟩ ↑ Op + ↑ Arg2 ⟨X⟩)

3. Representing the Network and the Tokens

This section describes representations for tokens and nodes that allow very fast interpreters to be written.

3.1. Working memory elements

The representation chosen for the working memory elements should have two properties.

- The representation should make it easy to extract values from elements because every test involves extracting one or more values.
- The representation should make it easy to perform the tests once the values are available.

To make extracting the values easy, each element should be stored in a contiguous block in memory, and each attribute should have a designated index in the block. For example, if elements of class Ck had seventeen attributes, A1 through A17, they should be stored as blocks of eighteen values. The first value would be the class name (Ck). The second value would be the value of attribute A1. The third would be the value of attribute A2, and so on. The particular assignment of indices to attributes is unimportant; it is important only that each attribute have a fixed index, and that the indices be assigned at compile time. This allows the compiler to build the indices into the nodes. Thus instead of a node like the following:

Is the value of the Status attribute Pending?

the compiler could build the node

Is the value at location 8 Pending?

With this representation, each value can be accessed in one memory reference, regardless of the number of attributes possessed by an element.

To make the tests inexpensive, the representation should have explicit type bits. One obvious way to represent a value is to use one word for the type and one or more words for the value proper. But more space-efficient representations are also possible. For example, consider a production system language that supports three data types, integers, floating point numbers, and atoms. A representation like the following might be used: One word would be allocated to each value. For integers and atoms, the low order sixteen (say) bits would hold the datum and the seventeenth bit would be a type bit. For floating point numbers, the entire word would be used to store a normalized floating point number. A floating point number would be recognized by having at least one non-zero in the high order bits.

3.2. The network

This section explains how to represent nodes in a form similar to von Neumann machine instructions. This representation was chosen because it allows the network interpreter to be organized like the interpreters for conventional von Neumann architectures.

3.2.1. *An assembly language notation*

To make it easier to discuss the representation for the nodes, an assembly language notation is used below. A one-input node like

Is the value of locating 8 Pending?

becomes

```
TEQA 8, Pending
```

The T, which stands for test, indicates that this is a one-input node. The EQ indicates that it is a test for equality. (It is also necessary to have NE for not equals, LT for less than, etc.). The A indicates the node tests data of type atom. (There is also a type N for integer values, a type F for floating point, and a type S for comparing two values in the same working memory element). Two-input nodes are indicated by lines like the following.

```
L001  AND (2) = (1)
```

L001 is a label. AND indicates that this is a two-input node for non-negated patterns. The sequence (2) = (1) indicates that the node compares the second value of elements from the left and the first value of elements from the right; the = indicates that it performs a test for equality. The terminal nodes contain the type TERM and the name of the production. For example

```
TERM Plus0x
```

As will be explained below, the ROOT node is not needed in this representation.

3.2.2. *Linearizing the network*

To make the nodes like the instructions for a von Neumann machine, it is necessary to eliminate the explicit links between nodes. Many of the explicit links can be eliminated simply by linearizing the network, placing a node and its successor in contiguous memory locations. However, since some nodes have more than one successor, and others (the two-input nodes) have more than one predecessor, linearizing is not sufficient in itself: two new node types must be defined to replace some of the links. The first of the new nodes, the FORK, is used to indicate that a node has more than one successor. The FORK node contains the address of one of the successors. The other successor is placed immediately after the FORK. For example, the FORK in the following indicates that the node L003 has two successors.

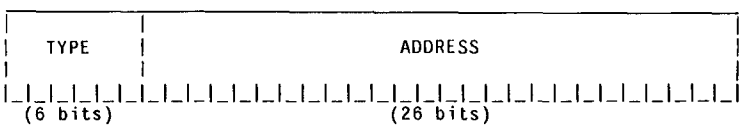
```
L003  TEQA 0, Expression
      FORK L004
      TEQA 3, +
      . . .
L004  TEQA 3, *
```

The other new node type, the MERGE, is used where the network has to grow back together—that is, before two-input nodes. The two-input node is placed after one of its predecessors (say its left predecessor) and the MERGE is placed after the other. The MERGE, which contains the address of the two-input node, functions much like an unconditional jump. Fig. 2 shows the effect of the linearization process; it contains the productions from Fig. 1 and the linearized network for their LHSs.

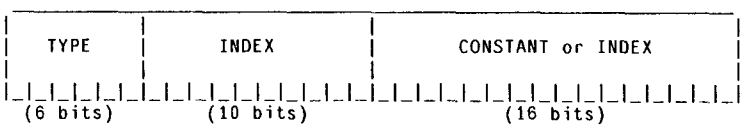
3.2.3. *Representing the nodes in memory*

This section shows how the nodes could be represented on a computer which has a thirty-two bit word length. The thirty-two bit word length was chosen because it is typical of today's computers; the precise word length is not critical, however. Since the network can be rooted at a FORK (see the example in Fig. 2) it is not necessary to have an explicit root node for the network. Hence only seven classes of nodes are needed; FORKs, MERGEs, the two kinds of one-input nodes, the two kinds of two-input nodes, and the terminal nodes.

FORKs and MERGEs could be represented as single words. Six bits could be used for a type field (that is, a field to indicate what the word represents) and the remaining twenty-six bits could be used for the address of the node pointed to. FORKs and MERGEs would thus be represented:



Both kinds of one-input nodes could be represented as single words that are divided into three fields. The first field would hold the type of the node. The second field would hold the index of the value to test. The third field would hold either a constant or a second index. The bits in a word could be allocated as follows.



A sixteen-bit field is required to represent an integer or an atom using the format of Section 3.1. Since a floating point number cannot be represented in sixteen bits, in nodes that test floating point numbers, this field would hold not the number, but the address of the number.

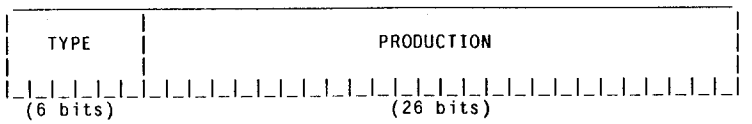
```
(P Plus0x
        (Goal   ↑Type Simplify   ↑Object ⟨N⟩)
        (Expression   ↑Name ⟨N⟩   ↑Arg1 0   ↑Op +   ↑Arg2 ⟨X⟩)
-->     ...)

(P Time0x
        (Goal   ↑Type Simplify   ↑Object ⟨N⟩)
        (Expression   ↑Name ⟨N⟩   ↑Arg1 0   ↑Op *   ↑Arg2 ⟨X⟩)
-->     ...)
```

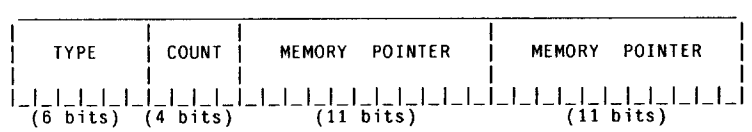
```
ROOT    FORK L003               ; Root node of the network
        TEQA 0, Goal            ; Is the element class Goal?
        TEQA 1, Simplify        ; Is the Type Simplify?
        FORK L002
L001    AND (2) = (1)           ; Two-input node for Plus0x
        TERM Plus0x             ; Report Plus0x is satisfied
L002    AND (2) = (1)           ; Two-input node for Time0x
        TERM Time0x             ; Report Time0x is satisfied
L003    TEQA 0, Expression      ; Is the element class Expression?
        TEQN 2, 0               ; Is the Arg1 0?
        Fork L004
        TEQA 3, +               ; Is the Op +?
        MERGE L001
L004    TEQA 3, *               ; Is the Op *?
        MERGE L002
```

FIG. 2. A compiled network.

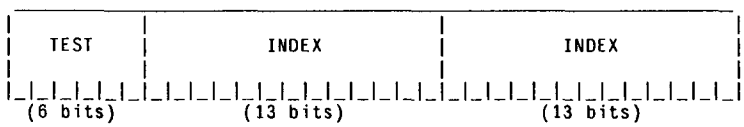
The terminal nodes could also be stored in single words. These nodes contain two fields, the usual type field plus a longer field for the index or address of the production that the node represents.



The length of a two-input node would depend on the number of value pairs tested by the node. Each node could have one word of basic information plus one word for each value pair. The first word would contain a type field, a pointer to the memory for the left input, a pointer to the memory for the right input, and a field indicating how many tests are performed by the node. The bits in the word could be allocated as follows.



The word for each test would contain three fields. Two fields would hold the indices of the two elements to test. The remaining field would indicate the test to perform; that is, it would indicate whether the node is to test for equality of the two elements, for inequality, or for something else. The bits in the word might be allocated as follows.



Note that the index fields here are longer than the index fields in the one-input nodes. This is necessary because the indices in the two-input nodes must designate elements in the tokens as well as values in the elements.

3.3. The tokens

This section describes a space-efficient representation for tokens. This representation is not suitable for all interpreters; it requires the interpreter to process only one working memory change at a time, and it requires that certain parts of the network be traversed depth first. Fortunately, these are not serious restrictions. The simplest way to perform the match is process one token at a time, traversing the entire network depth first. Section 4 describes an interpreter that operates in this manner.

If the interpreter operates this way, then it can use a stack to represent its tokens. When a token has to be built, first the tag for the token is pushed onto the stack, and then the working memory elements are pushed onto the stack in order. When tokens have to be extended (a very common operation—see the code in Section 4) the additional working memory elements are just pushed onto the stack.

The one-input nodes will be more efficient if they do not use this stack. Since all the one-input nodes will process the same working memory element—the element that was just added to or deleted from working memory—the element should be made easily available. The element could be copied into a dedicated location in memory, or the address of the element could be loaded into a dedicated base register. Either of these would make it possible for the one-input nodes to access the element without going through the stack.

3.4. The interpreter's state

In addition to the stack for tokens, the interpreter must maintain another stack for its state information. One reason for the stack is to allow the interpreter to find its way about in the network. When the interpreter passes a FORK, it pushes the pointer it does not follow onto the stack. Then when it reaches the

end of a path it pops a pointer from the stack and follows it. Another reason for the stack is to provide a place for the two-input nodes to keep their local information. As will be seen in the next section, the two-input nodes sometimes have to suspend themselves while their successors are processed. The stack holds the information that is needed to resume processing the two-input nodes.

4. The Network Interpreter

This section provides a concrete description of the operations performed by the network interpreter. One node from each class has been selected, and the code to interpret the nodes has been written. It might be noted that since the code sequences are short and simple, they could easily be written in microcode.

The code is written in a PASCAL-like language which has literal labels and field extraction. Field extraction is indicated by putting two numbers within angle brackets; the first number is the index of the high order bit in the field, and the second number is the index of the low order bit. The assumption will be made that the bits are numbered from right to left, with the low order bit being bit zero. Thus the expression SELF⟨31:26⟩ indicates that the high order six bits of the value of the variable SELF are to be extracted and right justified.

The main loop of the interpreter is very simple: the interpreter fetches the next node from memory and dispatches on its type field. Let the segment of memory that holds the nodes be called NODE_MEMORY and let the pointer to the current node be called NC. The main loop is then:

```
MAIN:  SELF := NODE_MEMORY[NC];
       CASE SELF⟨31:26⟩ OF    !Type field is high order 6 bits
         0: GOTO FORK;
         1: GOTO MERGE;
         2: GOTO TERM;
         3: GOTO TEQA;
         . . .
         END;
```

The node is copied into the variable SELF so that the node programs can examine it. The assignment of numbers to the various node types is arbitrary. Goto's are used instead of procedure calls because these examples make all the state of the interpreter explicit, and not hidden in PASCAL's stack.

TEQA is typical of the one-input nodes for testing constants. If the segment of memory that holds the working memory element being processed is called CURRENT, then TEQA is as follows.

```
TEQA:  TEMP := CURRENT[SELF⟨25:16⟩]:     !Get the word pointed to
                                         !by the index field
```

```
IF (TEMP⟨31:16⟩ = 0) AND          !Test type bits
   (TEMP⟨15:0⟩ = SELF⟨15:0⟩)      !Test value
THEN GOTO SUCC
ELSE GOTO FAIL:
```

Either SUCC or FAIL is executed after each one-input node. SUCC is executed when the test succeeds, and FAIL is executed when the test fails. SUCC increments the node counter to point to the next node.

```
SUCC:  NC := NC + 1;
       GOTO MAIN;
```

FAIL tries to get a node from the stack of unprocessed nodes; if it cannot, it halts the match. Assuming the stack is named NS and the pointer to the top of the stack is called NSTOP, the code is:

```
FAIL:  IF NSTOP < 0 THEN GOTO EXIT_MATCH;
       NC := NS[NSTOP];
       NSTOP := NSTOP - 1;
       GOTO MAIN;
```

The one-input nodes for comparing pairs of values are similar to the other one-input nodes. TEQS is typical of these nodes.

```
TEQS:  IF CURRENT[SELF⟨25:16⟩] = CURRENT[SELF⟨9:0⟩]
       THEN GOTO SUCC
       ELSE GOTO FAIL;
```

FORK pushes an address onto NS and then passes control to the following node.

```
FORK:  NSTOP := NSTOP + 1;
       NS[NSTOP] := SELF⟨25:0⟩;
       GOTO SUCC;
```

A two-input node must be able to determine whether it was reached over its left input or its right input. This can be indicated to the node by a global variable which usually has the value LEFT, but which is temporarily set to RIGHT when a MERGE passes control to a two-input node. If this global variable is called DIRECTION, the code for the MERGE is

```
MERGE: DIRECTION := RIGHT;
       NC := SELF⟨25:0⟩;
       GOTO MAIN;
```

The two kinds of two-input nodes are very similar, so only AND is shown here. In order not to obscure the more important information, some details of

the program are omitted. The code does not show how the variables are tested, nor does it show how tokens are added to and removed from the node's memories. Assuming the token stack is called TS and the pointer to the top element is called TSTOP, the program is as follows.

```
! Control can reach this point many times during the processing
! of a token. The node needs to update its state and put
! information on NS only once, however.
!
AND:IF NS[NSTOP] <> NC                    !If the state is not in NS
    THEN                                  !Then put it there
      BEGIN
      NSTOP := NSTOP + 4;
      NS[NSTOP] := NC;
      NS[NSTOP - 1] := DIRECTION;
      NS[NSTOP - 2] := MEMORY_CONTENTS
                       (OPPOSITE(DIRECTION));
      NS[NSTOP - 3] := TSTOP;
      MODIFY_MEMORY(DIRECTION);       !Store the token
      DIRECTION := LEFT;              !Reset to the default
      END;
!
! Go process the tokens
!
    IF NS[NSTOP] - 1] = RIGHT THEN GOTO RLOOP
      ELSE GOTO LLOOP;
! Compare the token to the elements in the right memory
!
LLOOP:  REPEAT
          TEMP := NEXT_POSITION(NS[NSTOP - 2]);
          IF TEMP = NIL                !If right memory is empty
          THEN                         !Then clean up and exit
            BEGIN
            TSTOP := NS[NSTOP - 3];
            NSTOP := NSTOP - 4;
            GOTO FAIL;
            END
        UNTIL PERFORM_AND_TEST(TEMP,LEFT);
!
! Fall out of the loop when the test succeeds so that
! the successors of this node can be activated
!
```

```
            !Extend the token
            TSTOP := NS[NSTOP - 3] + 1; TS[NSTOP] := TEMP;
            !Prepare NS so that control will return to this node
            NSTOP := NSTOP + 1; NS[NSTOP] := NC;
            !Pass control to the successors of this node
            NC := NC + SELF⟨25:22⟩ + 1; GOTO MAIN;
!
!  Compare the token the elements in the left memory
!
RLOOP:
```

This is similar to LLOOP.

The only remaining node type is the TERM node. Since updating the conflict set is a language-dependent operation, that detail of the TERM node cannot be shown. The rest of the processing of the node is as follows.

```
TERM:  UPDATE_CONFLICT_SET(SELF⟨25 : 0⟩);
       GOTO FAIL;
```

5. Performance of the Algorithm

Extensive studies have been made of the efficiency of the Rete Match Algorithm. Both analytical studies (which determined the time and space complexity of the algorithm) and empirical studies have been made. This section presents some of the results of the analytical studies. Because of space constraints, it was not possible to present the empirical results or the proofs of

TABLE 1. Space and time complexity

Complexity measure	Best case	Worst case
Effect of working memory size on number of tokens	O(1)	$O(W^C)$
Effect of production memory size on number of nodes	$O(P)$	$O(P)$
Effect of production memory size on number of tokens	O(1)	$O(P)$
Effect of working memory size on time for one firing	O(1)	$O(W^{2C-1})$
Effect of production memory size on time for one firing	$O(\log_2 P)$	$O(P)$

C is the number of patterns in a production.
P is the number of productions in production memory.
W is the number of elements in working memory.

the analytical results. The proofs and detailed results of some empirical studies can be found in [5].

Table 1 summarizes the results of the analytical studies of the algorithm. The usual notation for asymptotic complexity is used in this table [1].Writing that a cost is $O(f(x))$ indicates that the cost varies as $f(x)$ plus perhaps some smaller terms in x, The smaller terms are ignored because the $f(x)$ term will dominate when x is large. Writing that a cost is O(1) indicates that the cost is unaffected by the factor being considered. It should be noted that all the complexity results in Table 1 sharp; production systems achieving the bounds are described in [5].

6. Conclusions

The Rete Match Algorithm is a method for comparing a set of patterns to a set of objects in order to determine all the possible matches. It was described in detail in this article because enough evidence has been accumulated since its development in 1974 to make it clear that it is an efficient algorithm which has many possible applications.

The algorithm is efficient even when it processes large sets of patterns and objects, because it does not iterate over the sets. In this algorithm, the patterns are compiled into a program to perform the match process. The program does not have to iterate over the patterns because it contains a tree-structured sorting network or index for the patterns. It does not have to iterate over the data because it maintains state information: the program computes the matches and partial matches for each object when it enters the data memory, and it stores the information as long as the object remains in the memory.

Although the Rete algorithm was developed for use in production system interpreters, it can be used for other purposes as well. If there is anything unusual about the pattern matching of production systems, it is only that the pattern matching takes place on an unusually large scale. Production systems contain rather ordinary patterns and data objects, but they contain large numbers of them, and invocations of the pattern matcher occur very frequently during execution. If programs of other kinds begin to use pattern matching more heavily, they could have the same efficiency problems as production systems, and it could be necessary to use methods like the Rete Match Algorithm in their interpreters as well. Certainly the algorithm should not be used for all match problems; its use is indicated only if the following three conditions are satisfied.

- The patterns must be compilable. It must be possible to examine them and determine a list of features like the lists in Section 2.2.1.
- The objects must be constant. They cannot contain variables or other non-constants as patterns can.
- The set of objects must change relatively slowly. Since the algorithm maintains state between cycles, it is inefficient in situations where most of the data changes on each cycle.

ACKNOWLEDGMENT

The author would like to thank Allen Newell and Robert Sproull for many useful discussions concerning this work, and Allen Newell, John McDermott, and Michael Rychener for their valuable comments on earlier versions of this article.

REFERENCES

1. Aho, A.V., Hopcroft, J.E., and Ullman, J.D., *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, MA, 1974).
2. Cohen, B.L., A powerful and efficient structural pattern recognition system, *Artificial Intelligence* **9** (1977) 223–255.
3. Forgy, C.L., A network match routine for production systems, Working Paper, 1974.
4. Forgy, C.L., A production system monitor for parallel computers, Department of Computer Science, Carnegie-Mellon University, 1977.
5. Forgy, C.L., On the efficient implementation of production systems, Ph.D. Thesis, Carnegie-Mellon University, 1979.
6. Forgy, C.L., OPS5 user's manual, Department of Computer Science, Carnegie-Mellon University, 1981.
7. Hayes-Roth, F. and Mostow, D.J., An automatically compilable recognition network for structured patterns, *Proc. Fourth Internat. Joint Conference on Artificial Intelligence* (1975) 246–251.
8. McCracken, D., A production system version of the Hearsay-II speech understanding system, Ph.D. Thesis, Carnegie-Mellon University, 1978.
9. McDermott, J., Newell, A., and Moore, J., The efficiency of certain production system implementations, in: Waterman, D.A. and Hayes-Roth, F. (Eds.), *Pattern-Directed Inference Systems* (Academic Press, New York, 1978) 155–176.
10. Rychener, M.D., Production systems as a programming language for Artificial Intelligence applications. Ph.D. Thesis, Carnegie-Mellon University, 1976.

Received May 1980; revised version received April 1981