/ 1995 [illegible]

N95- 19755

[illegible]

Optimal Pattern Distributions In Rete-based Production Systems

p. 10

Stephen L. Scott

Hughes Information Technology Corporation
1768 Business Center Drive, 4th Floor
Reston, VA 22090
(703) 759-1356

sscott@mitchell.hitc.com

ABSTRACT

Since its introduction into the AI community in the early 1980's, the Rete algorithm has been widely used. This algorithm has formed the basis for many AI tools, including NASA's CLIPS. One drawback of Rete-based implementations, however, is that the network structures used internally by the Rete algorithm make it sensitive to the arrangement of individual patterns within rules. Thus while rules may be more or less arbitrarily placed within source files, the distribution of individual patterns within these rules can significantly affect the overall system performance. Some heuristics have been proposed to optimize pattern placement, however, these suggestions can be conflicting.

This paper describes a systematic effort to measure the effect of pattern distribution on production system performance. An overview of the Rete algorithm is presented to provide context. A description of the methods used to explore the pattern ordering problem area are presented, using internal production system metrics such as the number of partial matches, and coarse-grained operating system data such as memory usage and time. The results of this study should be of interest to those developing and optimizing software for Rete-based production systems.

INTRODUCTION

The Rete algorithm was developed by Charles Forgy at Carnegie Mellon University in the late 1970's, and is described in detail in [Forgy, 1982]. Rete has been used widely in the expert system community throughout the 1980's and 1990's, and has formed the basis for several commercial and R&D expert system tools [Giarratano & Riley, 1989] [ILOG, 1993]. Recent enhancements have been proposed based on parallel processing [Miranker, 90] and matching enhancements [Lee and Schor, 1992]. Rete provides an efficient mechanism for solving the problem of matching a group of facts with a group of rules, a basic problem in a production system.

In this section, an overview of the Rete algorithm is given in order to provide context for the discussion to follow. This presentation, however, is not intended to be a rigorous analysis of the Rete algorithm.

Rete based systems assume a working memory that contains a set of facts and a network of data structures that have been compiled from rule definitions. The rules contain a set of condition elements (CE's) that form the left-hand-side (LHS), and a right-hand-side

(RHS) that performs actions. The RHS actions may be side-effect free, such as performing a computation, invoking an external routine, performing I/O to the input or output streams or file. Other actions on the RHS may cause changes in the working memory, such as insertions, deletions, or modifications of facts. The Rete network actually contains two main structures: a pattern network, and a join network. The pattern network functions to identify which facts in working memory are associated with which patterns in the rules. The join network is used to identify which variables are similarly bound within a rule across CE's.

Within the pattern network, elements of the individual CE's are arranged along branches of a tree, terminating in a leaf node that is called an alpha node. The join network consists of groups of beta nodes, each containing two nodes as inputs and one output that can be fed to subsequent beta nodes. Finally, the output of the join network may indicate that one or more rules may be candidates for firing. Such rules are called activations, and constitute inputs to the conflict set, which is a list of available rules that are ready for execution. Typically, some arbitration mechanism is used to decide which rules of equal precedence are fired first. When a rule fires, it may of course add elements to or delete elements from the working memory. Such actions will repeat the processing cycle described above, until no more rules are available to be fired.

Consider the following small set of facts and a rule. For simplicity, additional logical constructs, such as the TEST, OR, or NOT expressions are not considered, and it is assumed that all CE's are ANDed together, as is the default. Note that myRule1 has no RHS, as we are focusing only on the LHS elements of the rule.

```
(deffacts      data
        (Group        1 2 3)
        (Int 1)
        (Int 2)
        (Int 3))

(defrule       myRule1
        (Group        ?i ?j ?k)
        (Int ?i)
        (Int ?j)
        (Int ?k)
=>)
```

This rule can be conceptualized in a Rete network as follows (see Figure 1). There are two branches in the pattern network, corresponding to the facts that begin with the tokens "Group" and "Int", respectively. Along the "Group" branch of the tree, there are nodes for each of the tokens in the fact, terminating with an alpha node that contains the identifier "f-1" corresponding to the first fact in the deffacts data defined above. Similarly, along the "Int" branch, there is one node for all the facts that have "Int" as a first token, and then additional nodes to show the various values for the second token. Alpha nodes along this branch also contain references to the appropriate facts that they are associated with, numbered in the diagram as "f-2" through "f-4". Note that the "Int" branch has shared nodes for structurally similar facts, i.e. there is only on "Int" node even though there are three facts with "Int" as a first token.

On the join network, myRule1 has three joins to consider. The first CE of myRule1 requires a fact consisting of a first token equal to the constant "Group" followed by three additional tokens. The alpha node of the "Group" branch of the pattern network supplies one such fact, f-1. The second CE of myRule1 requires a fact consisting of a first token equal to the constant "Int" followed by another token, subject to the constraint that this token must be the same as the second token of the fact satisfying the first CE. In this case, the fact f-2 meets these criteria, hence the join node J1 has one partial activation. This is because there is one set of facts in the working memory that satisfy its constraints. Continuing in this fashion, the output of J1 is supplied as input to J2, which requires a satisfied join node as a left input and a fact of the form "Int" followed by a token (subject to the constraint that this token must be equal to the third token of the first CE). The fact f-3 meets these criteria, so join node J2 has one partial activation as well. This process continues until we finish examining all CE's in myRule1 and determine that there are indeed facts to satisfy the rule. The rule is then output from the join network with the set of facts that satisfied its constraints and sent on to the agenda, where it is queued up for execution.

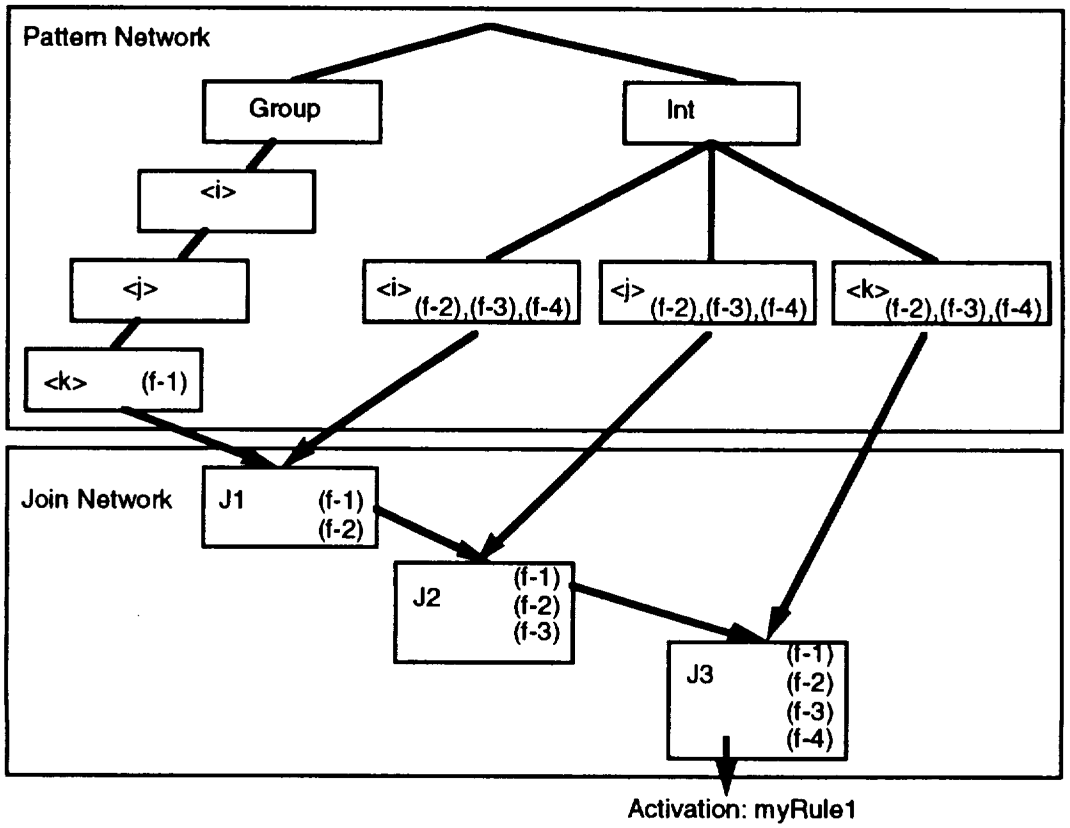

Figure 1. Rete Network for myRule1. This diagram depicts the Rete pattern and join networks for a rule with four CE's.

Consider the same set of data and a rule with the CE's arranged in a different order. Semantically, myRule1 and myRule2 are the same, however, the number of partial matches generated by myRule2 is much greater than that generated by myRule1.

```
(defrule        myRule2
        (Int ?i)
        (Int ?j)
        (Int ?k)
        (Group          ?i ?j ?k)
=>)
```

With this rule, we have three facts that match the first CE. Examining the first two CE's, there are three possible facts that can match the second CE, hence there are nine possible ways to satisfy the first two CE's. Moving to the third CE, there are again three ways to satisfy the third CE, but each of these must be considered with the nine possibilities that preceded it, hence there are 27 possible ways to satisfy the first three CE's. Fortunately, the fourth CE is satisfied by only one fact, so the number of partial activations for CE's one through four is only one; it is the fact that matches the fourth CE (f-1), coupled with exactly one set of the 27 possibilities available for CE's one through three. Summarizing, there are 40 partial activations for this rule (3 + 9 + 27 + 1).

From the above discussion, we have seen that pattern ordering in the LHS of rules can have significant impact on performance. Unfortunately, there are a large number of possible orderings one can try in even a small rule. Since in general, in a rule with N CE's, there are N ways to place the first CE, N-1 ways to place the second CE, and so on, the number of possible pattern arrangements is given by N!. As there may be many rules in an expert system, each with a large number of possible CE orderings, it should be clear that it is prohibitively expensive to do an exhaustive search of all possible arrangements of all rules in an attempt to optimize performance.

There may be some reduction in the number of arrangements if one considers that from the pattern network point of view, some arrangements produce an equal number of partial activations and thus can be considered together for analysis purposes. For example, the rule aRule1

```
(defrule aRule1
        (Group ?i ?j)
        (Int ?i)
        (Int ?j)
=> )
```

has the same number of partial activations as aRule2,

```
(defrule aRule2
        (Group ?i ?j)
        (Int ?j)
        (Int ?i)
=> )
```

because the CE's in slots 2 and 3 are similar with respect to effect on the join network. So even though there are 6 possible arrangements of CE's in this rule, only 3 actually produce different numbers of partial activations in the join network. This property is used extensively in the Analysis section that follows, as it allows valid results to be obtained using a manageable subset of the possible pattern orderings.

ANALYSIS AND RESULTS

In an effort to better understand the effects of pattern ordering on production system performance, a series of tests were conducted. This section describes the various experiments and the results obtained.

Partial Activations

From the discussion above, it is evident that some CE orderings are considerably more efficient than others with respect to partial activations. A test suite was developed using a rule with N CE's and a data set of N facts, where N-1 CE's are syntactically similar and one CE joins across the remaining N-1 CE's. This is the configuration used in the example myRule1 and myRule2 above; note that in that case N=4 because there are four CE's.

To interpret the data in the following table, match the number of CE's in the rule LHS (identified by the row labeled N = <n>) with the position of the constraining fact (the fact that has elements to match all other CE's in the rule). For the example myRule1 cited above, the row "N=4" is matched with "Fact Pos 1", giving 4 partial activations. Similarly, the example myRule2 cited above has a constraining fact in position 4, hence for N=4, the number of partial activations is 40. The following table shows the results of the number of partial activations for rules with the number of CE's varying from N=2 to N=8.

	Fact Pos 1	Fact Pos 2	Fact Pos 3	Fact Pos 4	Fact Pos 5	Fact Pos 6	Fact Pos 7	Fact Pos 8
N = 2	2	2						
N = 3	3	4	7					
N = 4	4	6	14	40				
N = 5	5	8	23	86	341			
N = 6	6	10	34	158	782	3906		
N = 7	7	12	47	262	1557	9332	55987	
N = 8	8	14	62	404	2804	19610	137258	960800

Table 1. Partial Activations in Rule Sets. This table shows the increase in partial activations observed in rules with various numbers of CE's, where a constraining fact is located at the position indicated by the column heading.

From this, at least two observations may be made. First, it is clear that the number of partial activations grows very rapidly. For this example set of rules and data, the number of partial activations for a rule with N CE's is given by

$$\sum_{i=0}^{N-1} (N-1)^i \qquad (1.0)$$

With such growth, on small computer systems, this may result in unexpected termination of a program, and even on large systems, performance may be degraded as the system attempts to accommodate the memory demands through paging or swapping. The second observation is the smaller the number of CE's on the LHS, the smaller the upper limit on partial activations. This suggests that a system with a larger number of smaller rules is better, at least from the vantage point of partial activations, than a system with a smaller number of larger rules.

Memory Usage

Within the Rete network implementation, data is maintained about partial activations. This data requires memory allocation, and as expected, the required memory grows in proportion to the number of partial activations. To examine this, the same suite of rules used above for partial activation testing was used, however, in this case, calls were made to the CLIPS internal function (mem-used) in order to calculate the memory required to store a network. The following table shows the results of these tests.

	Fact Pos 1	Fact Pos 2	Fact Pos 3	Fact Pos 4	Fact Pos 5	Fact Pos 6	Fact Pos 7	Fact Pos 8
N = 2	376	376						
N = 3	548	560	560					
N = 4	596	620	620	960				
N = 5	836	872	872	1912	8008			
N = 6	960	1008	1008	3228	18180	105652		
N = 7	1016	1076	1076	5076	36132	253832	1746792	
N = 8	1292	1364	1364	7864	65440	536008	4300744	33947716

Table 2. Memory Requirements for Various Rule Sets. This table shows the increase in memory requirements observed in rules with various numbers of CE's, where a constraining fact is located at the position indicated by the column heading. Memory allocation values are in bytes.

As expected, the amount of memory required to represent a rule varies in proportion to the number of partial activations. The two observations given for partial activations also hold here: some rule LHS orderings will require much less memory than others, and it is in general more memory efficient to have more small rules than a few large rules.

Reset Time

After rules and data are read into the system, the network must be updated to reflect the state required to represent these constructs. Data must be filtered through the network in order to determine facts are available, and comparisons must be made across CE's to determine which rules are eligible for firing. In order to investigate the time these processes take, the same test suite describe above was used, however, in this case, an operating system call was used to time the execution of the load and reset operations for the various rules. The "timex" command, available on many systems, gives operating system statistics about the real time, system time and user time required to execute a process. The following table shows the results of this test, giving real time in seconds, for the test suite.

	Fact Pos 1	Fact Pos 2	Fact Pos 3	Fact Pos 4	Fact Pos 5	Fact Pos 6	Fact Pos 7	Fact Pos 8
N = 2	0.1	0.1						
N = 3	0.1	0.1	0.1					
N = 4	0.1	0.1	0.1	0.1				
N = 5	0.1	0.1	0.1	0.1	0.11			
N = 6	0.1	0.11	0.11	0.11	0.11	0.17		
N = 7	0.1	0.1	0.13	0.11	0.12	0.25	0.96	
N = 8	0.11	0.1	0.11	0.11	0.15	0.45	2.51	17.88

Table 3. Reset Time for Rule Sets. This table shows the increase in reset time observed in rules with various numbers of CE's, where a constraining fact is located at the position indicated by the column heading.

As the reset times do not grow as rapidly as N increases, these results suggest that reset time is not as great a consideration as memory or number of partial activations. Also the granularity of timex is only 1/100 of a second, making more precise measurements difficult.

Placement of Volatile Facts

One heuristic that has been proposed concerns the placement of volatile facts in a rule. In data sets where a particular type of pattern is frequently asserted or retracted (or modified if the tool supports this), it is best to put these patterns at the bottom of the LHS of the rule. A typical example is a control fact containing constants, typically used to govern processing phases. The justification given is that because Rete attempts to maintain the state of the system across processing cycles, by placing the volatile fact at the bottom of the LHS, Rete does not need to check most of the rest of the network and can realize some performance gain. To test this, the following scenario was used. The data set consisted of a set of facts of the form

```
(Int     val <n>          isPrime Yes)
```

where <n> contained a prime number in the range 1 <= n <= 1000. A volatile counter fact of the form

```
(Counter <n>)
```

was used, where n again ranged from 1 <= n <= 1000. This fact was asserted and retracted for each value of n in the range. The rules to test whether or not n was prime were

```
(defrule        isPrime
        (Int     val ?n  isPrime Yes)
        ?x <- (Counter ?n)
=>
        (retract ?x)
        (assert (Counter =(+ ?n 1))
        (printout t ?n " is a prime " crlf))
```

```
(defrule        notPrime
        (Int     val ?n  isPrime Yes)
        ?x <- (Counter ?ctrVal&:(!= ?n ?ctrVal))
=>
        (retract ?x)
        (assert (Counter =(+ ?ctrVal 1))
        (printout t ?ctrVal " is not a prime " crlf))
```

The results below indicate run times in seconds for systems that searched for primes up to size K. The column 100, for example, indicates that primes between 1 and 100 were sought by using the volatile fact (Counter <n>) 100 times.

The example rules isPrime and notPrime given above correspond the rules used for the "volatile fact at bottom" row of the table. The "volatile fact at top" rules are virtually the same, except that the (Counter <n>) fact appears as the first CE instead of the second as illustrated above.

	100	250	500	750	1000
volatile fact at top	1.67	4.74	8.17	15.01	20.31
volatile fact at bottom	1.39	3.92	8.06	12.19	16.43

Table 4. Run Times for Rules with Volatile Facts. This table shows the differences in run times observed in rules with volatile facts placed at the top or bottom of the LHS. Times are in seconds.

This example shows that placing volatile facts at the bottom of a rule improves runtime performance, even for a small rule set and small amounts of data. The improvement is more obvious as the problem size grows, as the observed difference for K=100 is slight, whereas the difference for K=1000 is almost 4 seconds.

Placement of Uncommon Facts

Another heuristic suggests that facts that are relatively rare in the system should be placed first on the LHS. To test this, the following scenario was used. A data set contained three classes of facts: sensor facts, unit facts, and controller facts. These facts were distributed in the system in various proportions. Two rules were compared, one organized so that its CE's matched the distribution of the facts, and the other exactly opposite. In the following rules, rareFirst is tailored to perform well when the number of Ctrl facts is less than the number of Unit facts and the number of Unit facts is less than the number of Sensor facts. Conversely, rareLast is not expected to perform as well under this arrangement of data.

```
(defrule        rareFirst
        (Ctrl        Id ?cid        Status ?cStat)
        (Unit        Id ?uid        Ctrl ?cid        Status ?ustat    Value ?uVal)
        (Sensor      Id ?sid        Unit ?uid        Value ?sVal)
=>)
```

```
(defrule     rareLast
        (Sensor     Id ?sid     Unit ?uid     Value ?sVal)
        (Unit       Id ?uid     Ctrl ?cid     Status ?ustat   Value ?uVal)
        (Ctrl       Id ?cid     Status ?cStat)
=>)
```

The following table shows the number of partial activations generated for these rules given various distributions of matching Ctrl, Unit, and Sensor facts. The nomenclature i:j:k indicates that there were i Ctrl facts, j Unit facts, and k Sensor facts.

	Ctrl:Unit: Sensor 3:10:20	Ctrl:Unit: Sensor 5:20:50	Ctrl:Unit: Sensor 10:50:100	Ctrl:Unit: Sensor 25:125:500	Ctrl:Unit: Sensor 50:200:1000
rarest fact at top	33	75	160	650	1250
rarest fact at bottom	60	150	300	1500	3000

Table 5. Partial Activations for Rules with Rare Facts. This table shows the differences in partial activations observed in rules with patterns that match rarest facts at the top or bottom of the LHS.

This test shows that placing less common facts at the top of the LHS reduces the number of partial activations for the rule. Another point is worthy of mention here: had the distribution of facts been different, rareLast might have outperformed rareFirst rule. This points out a potential problem, as attempting to optimize a system based on one set of data may not have optimal results on other sets of data. Given that expert systems are typically much more data driven than other forms of software, this kind of optimization may not be effective if the data sets vary widely.

CONCLUSIONS

This paper has described a number of tests performed to investigate the effects of pattern ordering on production system performance. The results have borne out widely held heuristics regarding pattern placement on the LHS of rules. The results have quantified various aspects of the problem of partial activation growth by measuring the number of partial activations, memory requirements, system reset and run time for a variety of pattern configurations.

In general, the conclusions that can be drawn are as follows. Partial activations can vary exponentially as a result of pattern ordering. This suggests that (1) rules should be written with some regard to minimizing partial activations, and (2) systems should use larger numbers of small rules rather than smaller numbers of large rules. The second suggestion helps to reduce the risk of having potentially large numbers of partial activations. The growth of partial activations as a result of pattern ordering affects memory requirements, and, to a lesser extent, reset time. As the number of partial activations increases, the memory required and the reset time also increase.

Placing patterns that match volatile facts at the bottom of a rule LHS improves run-time performance. Placing patterns that match the least common facts in a system at the top of a rule LHS reduces the number of partial activations observed. It may be difficult to use these methods in practice, however, since both of them depend on knowing the frequency with which certain facts appear in the system. In some cases, this may be readily apparent, but in other cases, especially where the form of the data may vary widely, these may not be practical. Long term statistical analysis of the system performance may be required to make use of these optimizations.

REFERENCES

1. "CLIPS PROGRAMMER'S GUIDE, VERSION 6.0, JSC-25012, NASA Johnson Space Center, Houston, TX, June 1993.
2. "CLIPS USER'S GUIDE, VERSION 6.0", JSC-25013, NASA Johnson Space Center, Houston, TX, May 1993.
3. "ILOG Rules C++ User's Guide, Version 2.0", ILOG Corporation, 1993.
4. Forgy, Charles, "Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem", ARTIFICIAL INTELLIGENCE, vol 19, 1982, pg 17-37.
5. Giarratano, Joseph and Gary Riley, EXPERT SYSTEMS: PRINCIPLES AND PROGRAMMING, PWS-Kent Publishing Company, Boston, MA, 1989.
6. Lee, Ho Soo, and Schor, Marshall, "Match Algorithms for Generalized Rete Networks", ARTIFICIAL INTELLIGENCE, Vol 54, No 3, 1992, pg 249-274.
7. Miranker, Daniel, TREAT: A NEW AND EFFICIENT MATCH ALGORITHM FOR AI PRODUCTION SYSTEMS, Morgan Kaufmann Publishers, Inc, San Mateo, CA, 1990.
8. Schneier, Bruce, "The Rete Matching Algorithm," AI EXPERT, December 1992, pg 24-29.