The CLIPS Implementation of the Rete Pattern Matching Algorithm

Gary Riley

Introduction

Recent changes to the pattern matching algorithm used by CLIPS have dramatically improved performance on the classic waltz and manners benchmarks, which are frequently cited as measures of performance for rule-based systems. While the usefulness of these benchmarks is disputed, particularly for product comparisons, many believe that waltz and manners exhibit characteristics of some ‘real world’ applications and that generalized optimizations for these characteristics are therefore desirable. For this reason, interest has been expressed in the changes made to CLIPS that improved performance.

This paper will discuss those changes along with some of the differences between the CLIPS implementation of the rete algorithm and a more traditional approach. It is assumed that the reader has a basic understanding of the rete algorithm. The bibliography contains a number of references which go into great detail of this algorithm. This paper will not attempt to prove that the optimizations used by CLIPS are in fact the best implementation. It is also believed that all of the changes described are prior art.

Joins from the Right

Previous versions of CLIPS implemented the nesting of multiple CEs within a not CE by forking at the join at the join preceding the not CE and then rejoining the forks after the last join in the nested group of CEs. For example, the following rule would be represented with the topology shown in Figure A.

```
(defrule not-example
   (a ?x)
   (not (and (b ?x) (c ?x)))
   =>)
```

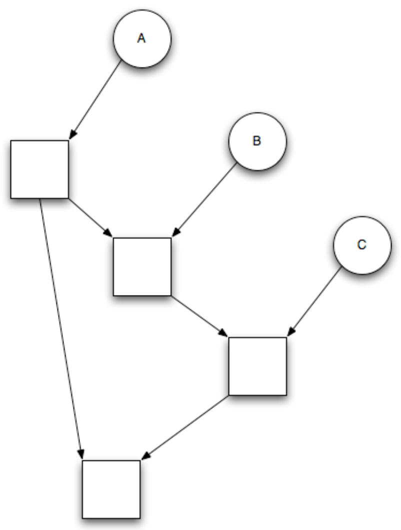

Figure A: CLIPS 6.24 Rete Topology

At the join at which the forks merge, it is necessary to check whether a partial match entering from the right is associated with the partial match entering from the left (i.e. when the join forks along these two paths it is necessary to recombine the partial matches sent along the two paths). This solution works, but is not very elegant.

Another drawback to this solution is that the network topology does not produce the best combination of partial matches for display by the matches command, which is very useful for debugging. For example,

```
CLIPS> (reset)
CLIPS> (assert (a 1) (b 2) (c 2))
<Fact-3>
CLIPS> (matches not-example)
Matches for Pattern 1
f-1
Matches for Pattern 2
f-2
Matches for Pattern 3
f-3
Partial matches for CEs 1 - 2
f-1,
Activations
f-1,
CLIPS> (facts)
f-0     (initial-fact)
f-1     (a 1)
f-2     (b 2)
f-3     (c 2)
For a total of 4 facts.
CLIPS>
```

It’s much more useful to see the partial matches generated for patterns 2 and 3 combined (patterns b and c), than the partial matches for patterns 1 and 2 (patterns a and b).

CLIPS 6.3 removes the fork in the join topology and supports a “join from the right” topology which more closely matches the layout of the rule as shown in Figure B.

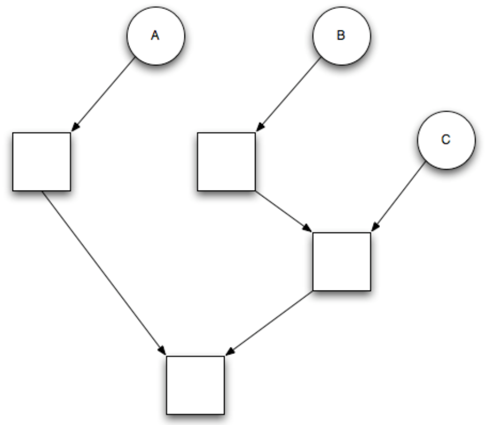

Figure B: CLIPS 6.3 Rete Topology

Algorithmically, it is no longer necessary to have separate code specifically handling the case of joining partial matches which forked. Instead the join entered from the right by the C pattern unifies the partial matches based on the value of the variable ?x and the join entered from the right with the partial matches from the patterns b and c also unifies based on the variable ?x with the partial matches generated by pattern a.

Using this topology, the output of the matches command is more informative:

```
CLIPS> (reset)
CLIPS> (assert (a 1) (b 2) (c 2))
<Fact-3>
CLIPS> (matches not-example)
Matches for Pattern 1
f-1
Matches for Pattern 2
f-2
Matches for Pattern 3
f-3
Partial matches for CEs 2 - 3
f-2,f-3
Partial matches for CEs 1 - 3
f-1,*
Activations
f-1,*
CLIPS>
```

As a historical note, the Automated Reasoning Tool (ART) included this functionality over two decades ago in part to improve the efficiency of their schema system. While the CLIPS functionality automatically generates ‘joins from the right’ for groups of patterns nested within a not CE, ART also provided the functionality for the user to explicitly group patterns with a ‘join from the right.’

Adding Patterns

Prior versions of CLIPS automatically added the (initial-fact) pattern to rules in a number of situations (a rule without a pattern, a rule or nested group of patterns that begins with a not CE, and a rule or nested group of patterns that begins with a test CE). When a (reset) command is issued, the (initial-fact) fact is asserted which in turn causes the rules matching this pattern to evaluate the pattern following the (initial-fact) pattern.

CLIPS used this implementation for compatibility with ART, which in turn it's assumed used this rather simple and clever approach to remove the OPS5 restriction that the first pattern of a production could not be negated. This also simplified the rete algorithm implementation somewhat as it could always be assumed that the first pattern in a rule was not negated.

It became apparent over the years, however, that adding the (initial-fact) pattern to rules was essentially "exposing the plumbing" of the internal algorithms to the end users and caused more confusion than it was worth in simplifying the number of cases that needed to be checked. In addition, adding the patterns also made it more difficult to debug rules using the matches command as the user can no longer determine the index of a pattern solely by its position in the original rule.

In CLIPS 6.3, a list is kept of all the joins that must be 'primed' which includes the join for a rule without patterns and the first join of a group of joins that is negated. When created, each of this joins has a partial match created for it that is used to pass to its child joins when the join is primed. When a (reset) command is issued the joins which need to be primed are visited and the pre-created partial match is propagated to any child joins.

There are still some situations in which patterns must be added to a rule. For example, this rule:

```
(defrule add-pattern-example
   (a ?x)
   (not (and (b ?y)
             (not (and (c ?x) (d ?y)))))
   =>)
```

is automatically converted to this rule:

```
(defrule add-pattern-example
   (a ?x)
   (not (and (a ?x)
             (b ?y)
             (not (and (c ?x) (d ?y)))))
   =>)
```

The manner used to implement joins from the right does not allow the ?x variable from the a pattern to be unified with the ?x variable from the c pattern without including it as part of the intermediate 'join from the right' group. It's believed that this situation is less common than the situations for which the (initial-fact) pattern is currently being added.

Hashed Memories

The alpha memories use a large shared hash table for storage. The rationale for this decision is that the size of the table is an attribute that could be easily tuned by the user based on the size of their system. The beta memories each use their own hash table which is dynamically resized. It is believed that the hash tables would be more difficult for the user to tune and more likely to be more affected by the actual data than the alpha memories would be.

The beta memory resizing has minimal effect on manners, waltz, and sudoku. It can be enabled and disabled. The initial hash table size is set to a small size (17) and when the average number of partial matches per bucket exceeds a specified value (11), the hash table is resized. Beta memories are reset to their initial size when the hash tables contains no partial matches.

When a partial match is added to the memory, the hash value is computed based on the variable bindings that must be unified between the left and right memories. This hash value is stored with the partial match. When partial matches are joined, it's not necessary to evaluate the expression stored with each partial match if the hash values are the same. In other words, each bucket will likely have a number of partial matches with different hash values even though they share the same bucket index.

Figure G shows an example hashed beta memory containing several partial matches.

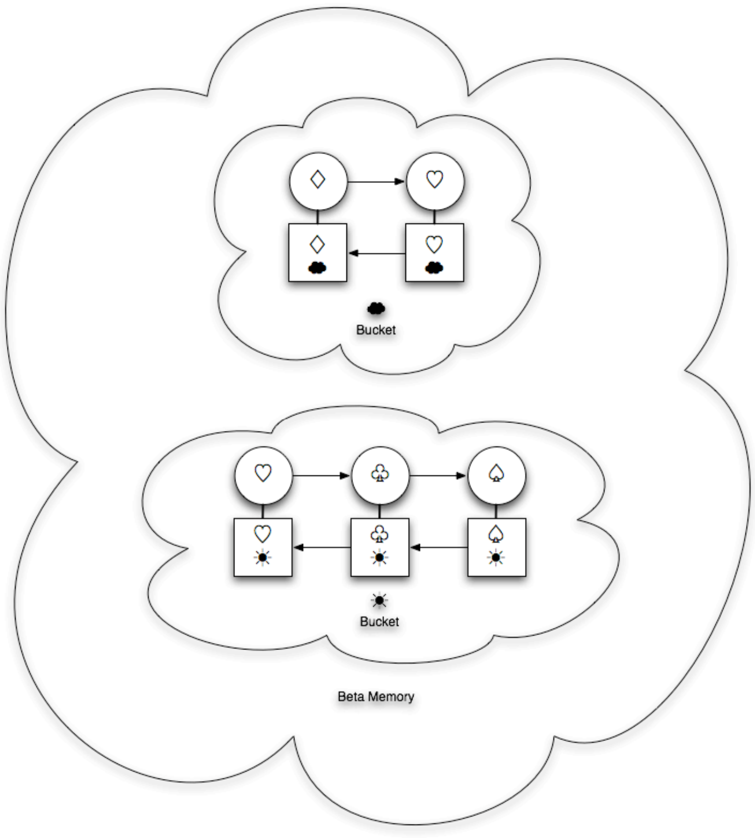

Figure G: Example Beta Memory

Salience

On each cycle, manners128 has a mean number of activations of 138 (with a maximum number of 9490) and waltz50 has a mean number of activations of 2298 (with a maximum of 12616). Manners does not utilize any salience values in the rules whereas waltz employs four difference salience values (-10, 5, 10, and the default value of 0). Because the depth strategy is used when running manners, new activations are inserted at the beginning of the agenda and so no search for the insertion point is required. Waltz activations, however, often had to be inserted within the interior of the activation list, a process that slowed down overall performance. Figure C shows an example of several activations of varying saliences stored using the CLIPS 6.24 represention.

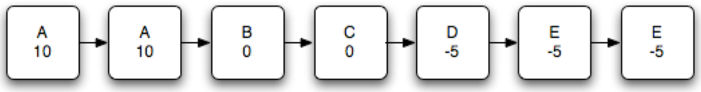

Figure C: CLIPS 6.24 Agenda

The solution to this problem was to define a salience group data structure that held all activations containing the same salience. For a modest number of salience values, this means that it is only necessary to search through the salience groups until the one of the appropriate value was found. If one did not exist, then it was inserted at the appropriate point. For the waltz benchmark, this means that at most four salience groups would need to be examined before the activation could be asserted.

Each salience group also contains pointers to the first and last activations in its list allowing activations to be easily added for the breadth as well as depth strategy. Figure D shows how the activations shown in Figure C for CLIPS 6.24 would be represented in CLIPS 6.3.

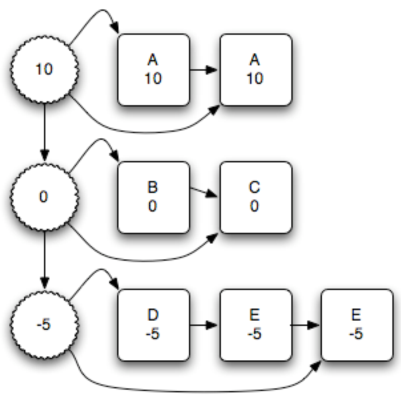

Figure D: CLIPS 6.3 Agenda

The new CLIPS 6.3 representation does not address the issue of having programs with large numbers of different saliences (perhaps from auto-generated rules). One approach to this could be to restrict the number of allowed salience values (perhaps from 20,000 to 200) so that an array could be used more appropriately to maintain the list for each salience value.

Exists Conditional Element

Prior versions of CLIPS represented the exists CE

```
(exists <CE>)
```

as

```
(not (not <CE>))
```

The exists CE is now represented using a single join rather than the two joins previously required. In addition, occurrences of (not (not <CE>) are automatically detected by the CLIPS parser and converted to (exists <CE>).

When a new partial match enters an exists join from the LHS, it is compared to all of the partial matches stored in the memory of the data structure that feeds the RHS. If a match is found, a new partial match is generated and sent to any child joins. In addition a link is added to the RHS partial match that links the partial match to the RHS partial match that satisfies it. Once a match is found, no further examination of the RHS partial matches is performed as it is unnecessary at this point.

When a partial match is deleted from the RHS memory feeding the exists join, the partial match is examined to determine if satisfies a partial match from the beta memory of the exists join. If it does, then the link between the two data structures is removed and the RHS memory is checked for a partial match following the one to be deleted. If one is found, then no further action is needed other than to update the links between the LHS and RHS partial matches. If on other satisfying partial match is found, then all of the partial matches associated with the partial match being deleted are also deleted. This process is repeated for each LHS partial match that was satisfied by the RHS partial match being deleted.

As an example of how the linkages between the two memories feeding the exists join are interconnected, the following rule could have the example linkages between the two memories associated with the join as shown in Figure E.

```
(defrule exists-example
  (circle (suit ?suit))
  (exists (square (suit ?suit) (weather ?)))
  =>)
```

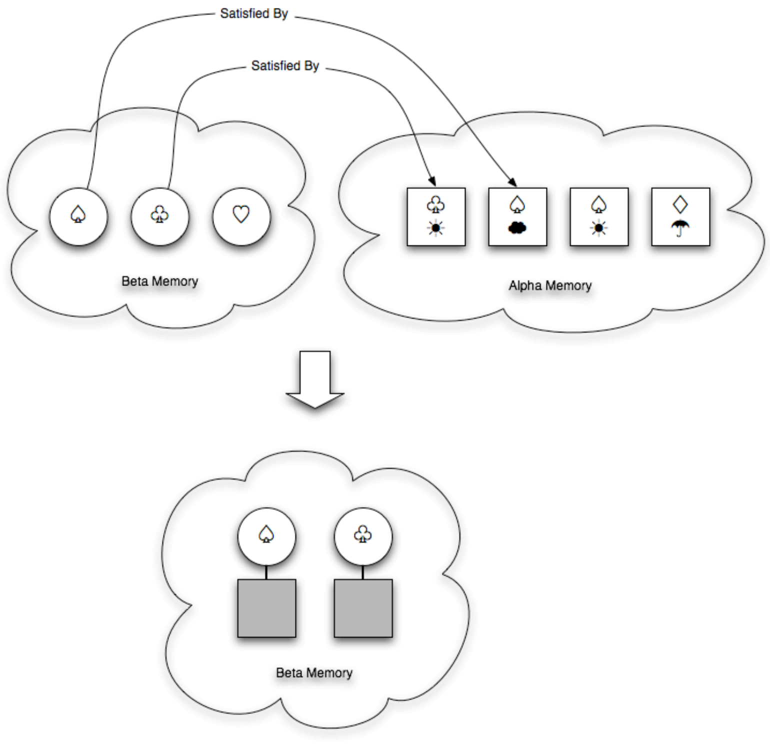

Figure E: Example Exists Conditional Element Linkages

Not Conditional Element

CLIPS does not make use of counts when processing a not conditional element. For example, each partial match in the beta memory of a join could keep a count of every partial match in the RHS memory that prevents it from being satisfied. As partial matches are added to or removed from the RHS memory, the count could be respectively incremented or decremented if appropriate. When the count changed to 0, this would indicate that a new partial match should be generated and set to any child joins. If the count changed from 0, then partial matches associated with this partial match should be removed.

CLIPS uses a lazy approach in evaluating conditions associated with the not CE. It is only necessary to take action when the count goes from 0 to 1 or 1 to 0. Rather than perform tests that would increment or decrement the count to a value greater than 2, CLIPS simply remembers the first RHS partial match that prevents a LHS partial match from being propagated to its child joins.

When a RHS partial match is removed, succeeding RHS partial matches are checked to determine whether they would prevent the LHS partial match from being propagated. If one is found, no further action is needed other than setting up a linkage between the two partial matches to indicate that one is causing the other not to be propagated. If no other partial match is found, then the LHS partial match can be propagated to its child joins.

As an example of how the linkages between the two memories feeding the not join are interconnected, the following rule could have the example linkages between the two memories associated with the join as shown in Figure F.

```
(defrule not-example
   (circle (suit ?suit))
   (not (square (suit ?suit) (weather ?)))
   =>)
```

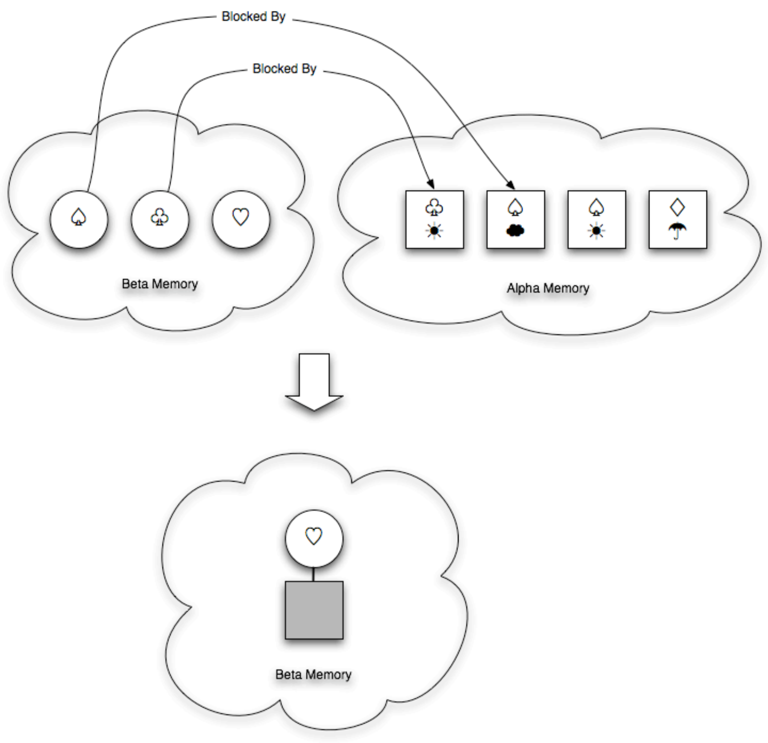

Figure F: Example Not Conditional Element Linkages

Asymmetric Retract

In a traditional implementation of the rete algorithm, the process of retraction can be considered to be symmetric to the process of assertion. Tokens (containing a partial match and an operation to be performed) are passed from one join to another and the test for each join is evaluated for each combination of the partial match in the token and the partial matches in the opposite memory. The difference between the two operations is whether partial matches are added or removed from the appropriate memories.

CLIPS uses an asymmetric approach to handling retractions. Assertions are handled in a traditional manner, although a partial match rather than a token is passed from one join to another. The partial matches, however, are heavily linked as part of the assertion process, so that a partial match's lineage can be determined. Figures G, H, and I show illustrations of the links that generated between partial matches. Figure G shows the links that allow a partial match to determine its left and right parents. Figure H shows the links that allow a partial match to determine the first child partial match associated with it. Figure I shows the links that group the sibling partial matches of a common parent partial match.

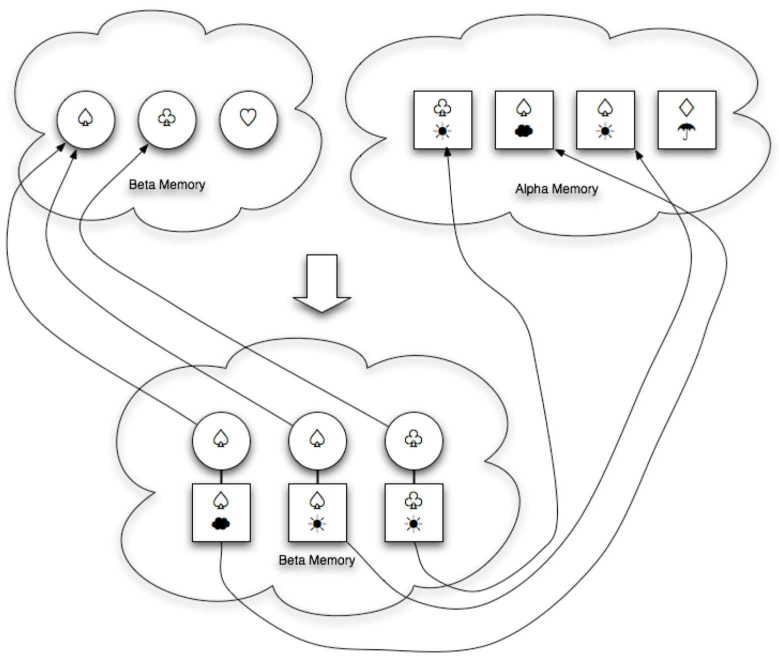

Figure G: Parent Links in the Partial Matches

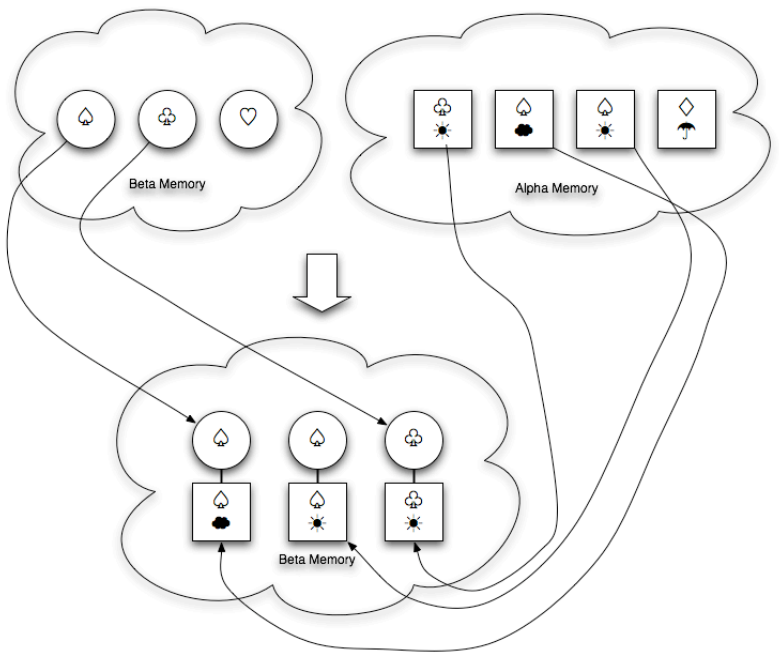

Figure H: Child Links in the Partial Matches

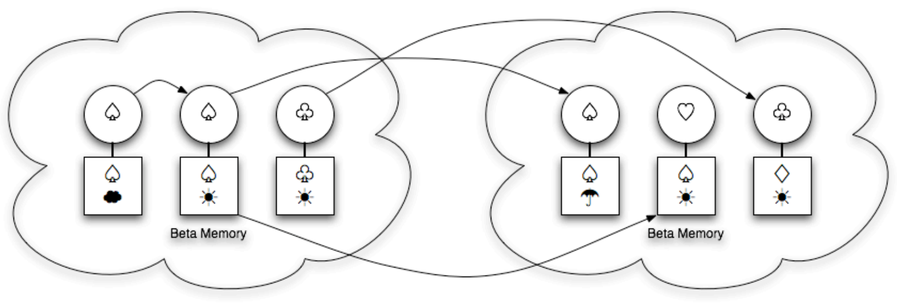

Figure I: Left and Right Parent Sibling Links across Beta Memories

The retraction process begins in the alpha memory, but uses a bottom-up approach for removing partial matches from the join memories. The child links of the partial matches are followed until a partial match is found that has no children. Once this partial match has been found, the partial matches can be deleted by following the sibling links and any child links encountered until none remain. At that point, the parent link of the bottom most partial match should have no more children and that partial match can be deleted along with its siblings.

Pattern Network Constants Hash Tables

The pattern network has a new node type that improves performance when a large number of constants are used in patterns. The child nodes of a constant selector node are all constant values. When the selector node is entered, a hash value is computed from the slot value to which the selector is being applied and the memory address of the selector itself. A shared hash table contains mappings between the selector and its constant child nodes. The hash value is used to locate the appropriate bucket in the hash table which is then searched for a mapping between the selector and the specified slot value. If one is found, then the constant child node has been satisfied and pattern matching can continue from that node.

Benchmark Results

Table 1 shows a comparison of the performance for several programs run with versions 6.24 and 6.3 of CLIPS.

Benchmark	Rules	Rules/Second 6.24	Rule/Second 6.3
MAB	32	118,900	100,500
Events	703	111,100	222,200
Sudoku	80	1,158	10,439
SICM	462	947	2,334
Sudoku Stress	80	300	5,038
Waltz 50	32	190	13,800
Manners 128	8	14	2,600

Table 1: Benchmark Performance Comparisons between CLIPS 6.24 and 6.3

The Manners program is a constraint satisfaction problem solver that finds an acceptable seating arrangement for guests at a dinner party, seating everyone next to a member of the opposite sex with a similar hobby. The data to the program (the number of guests) can be scaled to observe the effect on performance. It is often cited as a measure of rule-based performance, but is considered by many to be flawed as a benchmark.

The Waltz program processes a collection of lines and labels them according to a set of constraints. Like manners the data to the program can be scaled. It too is often cited, but considered by many to be flawed.

The Monkey and Bananas program is a simple planning problem in which a monkey must perform a sequence of actions in order to retrieve and eat a bunch of bananas. As a benchmark, it's useful only as a measure of best case performance.

The Events program is derived from an existing system and contains 703 similar rules each containing a single pattern matching two to five constant slot values. The input data is the only data utilized by the program (i.e. no chaining is involved as the rules simply identify events and do not assert any data).

The Shipping Instruction Comparison Module (SICM) is an expert system that automatically processes changes to cargo container shipping instructions. It's been in production use since 2005 and processes data sets that can range up to 10,000 or more facts.

The Sudoku program is a puzzle solver that's specifically designed as a benchmark. It's designed to fire the same number of rules regardless of the conflict resolution strategy for any engine that supports salience/priority values. Solutions are unique and easily verified. There are test cases for each of the 18 supported solution techniques. The design also allows the benchmark to be incrementally developed and tested as the more complex solution techniques are not employed until the easier ones fail. Although it's a bit more complex than manners, the problem domain is easily understood and there's considerable information online describing the various solution techniques used. Like manners and waltz, the amount of data can be scaled by selecting the size of the puzzle. It can be run in non-stressed and stressed modes to demonstrate how implementation techniques can effect performance.

Bibliography

Doorenbos, R. *Production Matching for Large Learning Systems*, Ph.D. Thesis, CMU CS-95-113, 1995.

Forgy, C. *On the Efficient Implementation of Production Systems*, Ph.D. Thesis, CMU SDL-425, 1979.

Forgy, C. *Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem*, Pages 17-37, Artificial Intelligence 19, 1982.

Friedman-Hill, E. *Jess in Action*, Manning Publications Co., Greenwich, 2003.

Giarratano, J., and Riley, G. *Expert Systems: Principles and Programming*, 4th Edition, Boston, Course Technology, 2005.

Schor, M. I., Daly, T. P., Lee, H.S., and Tibbitts, B. R. "Advances in Rete Pattern Matching," *Proceedings of the 1986 National Conference on Artificial Intelligence*, 1986.

http://en.wikipedia.org/wiki/Rete_algorithm