An Introduction to the Rete Algorithm

The audience should already understand the basics of inference rule-based systems. In particular,
- the LHS (left-hand side), or IF part of a rule
- the RHS (right-hand side), or THEN part of a rule
- Working Memory or WM, the objects the rule engine is observing
- the Agenda, or the list of rule instances awaiting execution. Rule instances reference a rule and the tuple(s) of data matching the condition of that rule.
- understand the syntax of Drools DRL, JRules IRL, or similar syntax

This presentation is intended to cover the 'basics' of the Rete algorithm. It does not address optimizations, though opportunities will make themselves obvious during this discussion. It also does not address 'advanced' conditions, such as tests for existence (IF THERE IS NO CAR WHERE THIS CAR IS BLUE...), alternate search sets (IF THE CAR IS BLUE FROM THE CARS IN LOT A...), or the size of a search set (IF THERE ARE 4 CARS WHERE THE CAR IS BLUE...).
What is Rete?

- A public domain, efficient pattern matching algorithm
- Initially published by Dr. Charles Forgy in his 1979 Ph.D. thesis
- The basis of most forward-chaining rule engines (CLIPS, Jess, JBoss Drools, IBM ILOG JRules, FICO's Blaze Advisor, etc.)
- Pronunciation: 'REET', 'REE-tee', or more commonly in Europe 're-tay', from the Latin 'rete', meaning network (from Wikipedia)

I pronounce it 'REE-tee' as does Dr. Forgy.
Purpose of Rete

From Dr. Forgy's Thesis: "Production systems have historically operated from one to two orders of magnitude slower than conventional programs, due in large part to the difficulty of performing the match."

Matching inefficiency in previous algorithms increased as a factor of both the number of rule conditions and the number of objects in WM making large systems impractical

Shorten execution times of pattern matching when updating the agenda

The purpose of the Rete algorithm is to improve on the execution times over alternative pattern matching schemes used in 'production systems' (a term referring to systems who logic is based on interacting IF–THEN rule statements)

In the second point, I'm alluding to the fact that previous matching algorithms scaled linearly (or worse) with increasing number of rules and objects in WM. A goal of Rete was to make this relationship less than a linear one.

In the third point, the main goal of Rete. It's all about the matching.
Assumptions of Rete

- Desired rule execution behavior is 'inference' or 'rule-chaining' rather than 'sequential'
- Working Memory changes slowly compared to pattern matching cycle times
- Pattern matching involves comparisons that are expensive to repeatedly reproduce. Rules tend to share conditional comparisons

All algorithms make assumptions about the problem space they model. Rete assumes:
1) We care about changes to observed objects (those in Working Memory or WM) during execution. If not, Rete can be slower than a simple sequential scan of the rules.
2) Each WM change requiring an update to the agenda involves a relatively small percentage of the total objects in WM.
3) Rules commonly involve symbolic comparisons (string comparisons, object equality, list evaluation, etc.) that are time consuming to repeat.
Assumptions of Rete

- Our rule systems are sufficiently complex that network set-up time will be compensated by improved matching performance.
- We are willing to trade additional memory consumption for execution time reduction.

Establishing the Rete data structure is not without its overhead cost. Using Rete for very small rule sets or very few objects in WM does not always result in improved performance due to this overhead.

The memory required by the Rete network is part of the cost for the improved pattern match speed.
Basics of Rete

- A directed acyclic graph (DAG) or dataflow model

- A stateful network of interconnected nodes (stateful of both with regard to WM and to rule conditions)

- Represents the entire, active rule set and 'current state' of objects in WM that may induce a change in the agenda

In computer science and mathematics, a directed acyclic graph, also called a DAG, is a directed graph with no directed cycles; that is, for any vertex \( v \), there is no nonempty directed path that starts and ends on \( v \). DAGs appear in models where it doesn't make sense for a vertex to have a path to itself. (from Wikipedia)

The network contains the result (state) of the evaluations of the LHS (the IF conditions) of all the rules against all of the objects' state at they are currently represented. Thus it is stateful of both working memory and the rule conditions across objects that can induce an agenda change and all active rules.
The Basics of Rete:

Two distinct parts to the network:

- **Alpha network (left side):** a discrimination network. Conditions involving only individual attributes of WM elements.
  
  The Alpha network is often referred to as the 'left side' of the Rete network. This should not be confused with the LHS or left-hand side of a rule. The Left side of the Rete network contains references to discrimination tests on WM objects. Discrimination tests are conditions which only involve attributes of a single WM object. "IF THE CAR IS BLUE..." represents a discrimination test on a single instance of 'car', for example.

- **Beta network (right side):** implements join conditions between attributes of different WM elements.
  
  The Beta network or right side of the Rete network stores the results of join tests. These are tests involving two or more objects, such as "IF THE COLOR OF A CAR IS THE COLOR OF A SEAT...". This test involved comparing all of the instances of 'car' to all of the instances of 'seat' to find a match.

I have also seen Rete network portrayed in a vertical fashion, with the alpha network on the top and the beta network on the bottom.
Basics of Rete

Tokens enter the network at a single entry point when changes to Working Memory occur.

Insertion of new WM objects is represented by a positive token; retraction by a negative token. Update is logically a retract (old) and insert (new). Tokens may split at forks.

Each path terminates at a node representing a single rule in the rule set. If the token reaches one or more terminal node, an agenda change occurs.

The Rete network is stateful of both the WM objects and the rule conditions. When objects are inserted, retracted, or updated in WM, the state of the network must be updated to reflect these changes.

For inserted objects, a positive token moves through the network. Nodes are updated to reflect the addition of this object.
For retracted objects, a negative token moves through the network.

Tokens only reach terminal nodes if matches are found that update all intermediate nodes. This results in adding new matching data tuples or removing old matching data tuples that no longer match. This causes an rule agenda change for rule represented by that terminal node.

Tokens can split at forks in the network and take both exiting paths simultaneously.
Rule Set Example

```
rule "rule_1"
  when
    A( a1 == 1, $x: a2 )
    B( b1 == 2, $y: b2, b3 == $x )
    C( c1 == $y )
  then
    System.out.println( "rule_1" );
end

rule "rule_2"
  when
    B( b1 == 2, $y: b2 )
    D( d1 == 300, d2 == $y )
  then
    System.out.println( "rule_2" );
end

rule "rule_3"
  when
    B( b1 == 2, $y: b2, $z: b3 )
    D( d1 == 300, d2 == $y )
    E( e1 == $z )
  then
    System.out.println( "rule_3" );
end
```

The syntax here is Drools DRL. But the syntax is very similar to ILOG JRules IRL and others.

Rule 1, for example, says: Print "rule_1" when you find a instance of Class A who's a1 attribute equals 1 and who's a2 attribute equals the b3 attribute of an instance of Class B (as a2 is assigned to variable $x and the Class B test says b2 == $x), and who's b1 attribute equals 2 and who's b2 attribute is equals to the c1 attribute of an instance of Class C; where each of the instances of Class A, B, and C are in Working Memory (WM).

NOTE:
- rule_1, rule_2, and rule_3 share a discrimination test on instances of class B, its b1 attribute must equal 2.
- rule_2 and rule_3 share a discrimination test for instances of class D, its d1 attribute must equal 300; and a join test between instances of Class B and Class D, Class B's b2 attribute must equal Class D's d2 attribute.
Example Facts

For our example, assume the following objects are have already been inserted into WM

Note that the first entry refers to an instance of Class A in which its a1 attribute equals 1, its a2 attribute equals 100 and its name attribute equals "a_1".

In the following digram, this notation is shortened from
A (a1=1, a2=100, "a_1")
to
A(1, 100, a_1)
This is the resultant Rete network or data structure:
- single entry node for tokens representing changing in WM is labeled 'Fact' on the left.
- one discrimination branch for each class represented in the LHS conditions of all rules.
- one terminal node exists for each (active) rule in the rule set.

The select nodes implement discrimination tests. The Alpha memory reference those objects in WM which passed the discrimination tests. The beta memory stores the 'current' join condition matches. And the terminal nodes represent each rule in the rule set along with their matching tuples.

Note the 'sharing' of both alpha and beta memory for rules which share conditions. This means that changes in WM involving elements in those nodes only need to be evaluated once, regardless of the number of rules which share that test.

Also note that A(2,100,a_2) is not present in Alpha Memory although it was inserted into WM. This is because it can never induce an agenda change as it failed the discrimination test on Class A where its a1 attribute must equal 1.
rule_1: A(1, 100, a_1), B(2, 10, 100, b_1), C(10, c_1)
rule_2: B(2, 11, 100, b_2), D(300, 11, d_1)
rule_2: B(2, 11, 200, b_3), D(300, 11, d_1)
rule_3: B(2, 11, 200, b_3), D(300, 11, d_1), E(200, e_1)

This is the state of the agenda, based of the data in the previous slide having been inserted in WM with the rule set previously defined
Now let's insert a new object in WM and see how the network is updated in response to this change in WM.
Here we are inserting a new instance of Class E into WM named 'e_2', where its e1 attribute equals 100. A positive token is passed through the network beginning at the entry node and traversing down to the path for the tests on class E, since it was an instance of Class E which was inserted. If we were retracting this instance of Class E, the token would have been a negative token.

Note the change in beta memory in the terminal node for rule_3. This induces an agenda change, adding an instance of rule_3 since a new matching tuple was generated, objects: B(2,11,100,b_2), D(300,11,d_1), E(100,e_2)
Updated Agenda

- **rule_3**: B(2,11,100,b_2), D(300,11,d_1), E(100,e_2)
- **rule_1**: A(1,100,a_1), B(2,10,100,b_1), C(10,c_1)
- **rule_2**: B(2,11,100,b_2), D(300,11,d_1)
- **rule_2**: B(2,11,200,b_3), D(300,11,d_1)
- **rule_3**: B(2,11,200,b_3), D(300,11,d_1), E(200,e_1)

This is the new state of the agenda, after inserting the new instance of Class E. Note the new rule instance for Rule 3 on the top of the agenda this insertion induced. This order assumes a conflict resolution strategy that places more recent WM changes higher on the agenda for rules of equal priority.
Insert Fact

D (d1 = 300, d2 = 10, "d_2")

For an additional example, assume the following object has just been inserted into WM
Here we are inserting a new object into WM, an instance of class D, named 'd_2', where $d_1=300$, $d_2=10$. A positive token is passed through the network beginning at the entry node and traversing down the discrimination path for class D. It passes all of the Class D discrimination tests ($D_1$ must equal 300) and continues into the Beta Network to evaluate join conditions. After updating the beta memory for successful matches for the join condition $b_2 == d_2$, the token splits, traversing both paths exiting that join test. This result in the addition of two rules instance, one for rule_2 and one for rule_3.
Updated Agenda

rule_2: B(2,10,100,b_1), D(300,10,d_2)

rule_3: B(2,10,100,b_1), D(300,10,d_2), E(100,e_2)

rule_3: B(2,11,100,b_2), D(300,11,d_1), E(100,e_2)

rule_1: A(1,100,a_1), B(2,10,100,b_1), C(10,c_1)

rule_2: B(2,11,100,b_2), D(300,11,d_1)

rule_2: B(2,11,200,b_3), D(300,11,d_1)

rule_3: B(2,11,200,b_3), D(300,11,d_1), E(200,e_1)

This is the state of the agenda, based on the data in the previous slide having been inserted in WM. The actual order of the two new rule instances on the agenda is determined by the rule engine's conflict resolution strategy, which varies by implementation.

Retraction of an object is simply the reverse process. A negative token is passed for the removed instance. Alpha memory and Beta memory entries involving the retracted object are removed. Any rule instance on the agenda involving the retracted object are removed from the agenda, even if they have not yet fired.
Practical Implications

- Multiple rules sharing the same condition only require the condition to be re-evaluated once, and only when the observed attribute changes.

- Traversal depth is a factor of the number of conditions on a given rule, not the number of rules within the rule set.

- Failed tests short-circuit network traversal. Place the most discriminating tests early in the network (first in your conditions).

Some practical implications for rule authors...

1) Create atomic conditions and code identical conditions the same way in different rules (don't use 'myInt > 1' in one rule and 'myInt >= 2' in another, for example). Obvious opportunities for multi-processing exists when network paths split (when different rules share conditions).

2) Rete network traversal path length does not increase linearly with rule set size or number of objects in WM. It increases with greater numbers of distinct conditions in the same rule. Join conditions are particularly costly.

3) Failed tests short-circuit Rete network traversal. Place the most discriminating tests first in your conditions.
Practical Implications

WM change notification is critical. Unnecessary WM change notification is not expensive, but should be avoided when possible

Rule conditions may be re-evaluated many, many times

Some practical implications for rule authors...

4) Be diligent in your WM change notification mechanism. Err toward over notification rather than missing notification. Some rule engines provide for some form of automatic notification (such as Drool’s support for Java Bean property change listener, for example). Use these if you can to ensure notification.

5) Rule conditions involving attributes of a specific class are re-evaluated each time an instance of that class changes. This may be many, many times. Only use light-weight conditions and ensure all data elements are in-memory. Validate the need for inference behavior and use sequential processing if available and it meets the needs of your rule set.
Questions?

Larry Terrill
EBDX.COM
rules@ebdx.com

http://ebdx.com/presentations/rete.pdf
If you found this presentation useful and would like similar training, please contact the address above. I provide consultive training, architectural, and implementation services for rule-based system.

This presentation is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License.
Copyright 2009-2011, EBDX.COM, LLC. All other rights reserved.