An Extension to the RETE Match Algorithm:
Supporting both Forward and Backward Chaining

Tamás Mészáros, Balázs Vadász
Technical University of Budapest,
Department of Measurement and Instrument Engineering
Budapest Hungary, H-1521, Mûegyetem rkp 9.
email: meszaros@mmt.bme.hu
vadasz@mmt.bme.hu
tel. & fax: (36) 166-4938

Content areas: *Expert System Design, Integrating AI and Conventional Systems, Knowledge Representation*

Abstract
The Rete match algorithm is widely accepted as one of the best solutions for the many pattern/many object pattern matching problem [Forgy79]. It is mainly applied in production systems to support the forward chaining inference. One of its main advantages is that it saves state information between the forward cycles. In every cycle only the working memory changes are processed. Moreover, this algorithm exploits the structural redundancy of the rules.
However, it has some drawbacks. It does not allow the run-time change of the rule base. It does not maintain a concentrated working memory. The working memory elements are stored in the node memories of the Rete network. Finally, the support for the backward chaining has not been worked out.
This paper presents a solution for this latest problem. It shows the extensions needed to provide for the backward chaining operation on the same rule base. The paper describes the detailed operation of the network during the backward chaining. Based on the considerations presented in the paper a complete inference engine has been implemented.

The authors declare that the paper is unpublished, original work, substantially different from papers currently under review and will not be submitted elsewhere.

Overview of the backward chaining

The backward chaining operation of production systems, also called hypothesis checking, tries to prove a statement using the rule base and the current content of the working memory. In general this hypothesis is a missing but required working memory element.

The state of a rule-based system can be characterised by the content of the working memory. The goal is to find a way from a state in which the working memory contains the required element to the current state.

The backward chaining operation is performed in a cyclic manner very much like the forward chaining. Each cycle receives a set of required working memory elements as an input and tries to reduce this set to a smaller one which can serve for a subsequent cycle as its input. These intermediate sets are not necessarily smaller than the initial one. Nevertheless, the goal to be achieved is an empty set. If it is reached then the elements of the original set of hypotheses are proved. In other words the way from the required state of the production system to the current one is found.

The starting point of the first cycle is the need for a given working memory element. Speaking in general this element is a set of hypotheses with only one element.

The steps of the cycle are the followings.

First the members of the input set are checked if they are present in the working memory. Those elements which are present will be removed from the hypothesis set. The second step determines the rules of the rule base which can produce at least one hypothesis. Some of these rules may be in the conflict set. In the third step these rules can be fired. Those elements of the hypothesis set which are produced by this action are eliminated from the set. The fourth step takes place if the hypothesis set is still not empty. Working memory elements satisfying condition parts of those rules which are able to create remaining elements of the hypothesis set are put together in a new set. This new set replaces the original hypothesis set and serves as the input for the next backward cycle.

The sequence of these cycles terminates if either the hypothesis set becomes empty or remains the same between two cycles[1].

Extension to the Rete match algorithm: support for the backward chaining inference

The Rete match is an efficient pattern matching algorithm for forward chaining. However, it was inherently designed to support the matching phase of the forward chaining. It does not support the bi-directional usage of the rules, which was the original idea of the rule-based systems.

Our aim was to design a Rete match based inference engine that is able to utilise the same Rete network for both forward and backward chaining. This implicitly means that the same working memory is used in both cases.

In the following the above discussed steps of the backward chaining cycle will be presented in the case of extended Rete match network. They are as follows:

- check if the current hypothesis exists in the working memory
- select the rules which can produce it (i.e. which contain a statement on their right-hand-side, producing the desired working memory element)
- check the existence of such a rule in the conflict set
- search for the working memory elements which are required to satisfy at least one rule being able to produce the current hypothesis

---

[1]Not concerning the implementational issues, e.g. time limits.

Search in the working memory

One of the most characteristic feature of the Rete match algorithm is the absence of a concentrated working memory. The working memory is "spread" over the Rete network. Only the working memory elements referenced by the left-hand-side of the rules are stored in the memories of the relevant nodes of this network.

Those working memory elements that do not even partially match any condition element are not stored at all.

The rule base may contain rules left-hand-side of which consists of only one condition element. The working memory elements matching only such kind of rules are not stored in the Rete network, since traversing the graph they do not pass join nodes. However, they can be found examining the rule instances in the conflict set. To avoid the search over the conflict set we extended the terminal nodes with a memory. The other kinds of working memory elements are stored in join nodes according to [Forgy79].

Therefore, all the working memory elements passing the network can be found in at least one of the node memories.

The search for a desired working memory element is very similar to the normal operation of the Rete network. The searched working memory element is sent to the input node that creates a token containing this element. The node programs examining this token are different from those which are in effect during the normal operation of the network.

The program of the single nodes is the very same.

The join nodes look up for the arriving token in their corresponding memory. If it can be found the search is successful. If not then the token is not forwarded toward the successor nodes since the they cannot contain more working memory elements. It means that the search over this path terminates after examining the first join node.

The terminal nodes work the same way. Of course they have only one memory. The arriving token is matched against the content of this memory. The criteria of the successful matching is the same as in the case of join nodes. The terminal nodes play role only if they are not preceded by at least one join node. During the normal operation, supporting the forward chaining, they do not influence the matching.

The search is over if either the working memory element is found or whole network has been traversed.

Searching for rules

The aim of this step is to find all those rules which potentially can produce the desired working memory element.

To perform this we have to search over the right-hand-side of the rules. The result of this search is a set of rules, which rules contain a statement making the desired element. We call this statement MAKE function according to [Forgy79].

The MAKE function produces an instance of a working memory element type. The parameters of this function are this type designator and some of the attributes defined by the type. According to this the MAKE statement can be treated as a pattern to be matched against the desired working memory element. If they match the rule containing this statement is able to produce that element. To perform this matching we can use a separate Rete network. This network is built by examining the right-hand-side of the rules and called MAKE-Rete network.

The MAKE-Rete network does not contain join nodes since one MAKE statement creates only one working memory element. The desired working memory element enters this network trough an input node. If this element satisfies all the tests embodied in the nodes on a path and reaches one of the terminal nodes in this way then this terminal node selects the corresponding rule from the rule base. Of course more than one rule can be selected at the same time. These rules can produce the desired working memory element.

Checking the conflict set

The purpose of this step is to check if one the rules selected in the previous step can be found in the conflict set. This subset of the conflict set consists of rule instances (unless it is empty).

A rule instance is generally much more constrained than the rule itself without the variable bindings. There can be several variables on the right-hand-side of a rule which obtain an actual binding from the left-hand-side during the instantiation. It is because variables can appear on both sides of a rule with the same name. They carry over information from the left-hand-side to the right-hand-side. According to this if we find an appropriate rule instance in the conflict set then the variable bindings have to be checked. If the bindings do not correspond to the attribute values of the desired element the instance cannot be used to produce that element. If it does by firing this rule we can obtain the desired working memory element.

Since the Rete match algorithm does not deal with the implementation of the conflict set the details of this search are beyond the scope of this paper.

Search for missing working memory elements

This step takes place if the previous ones did not succeed. In this case we have to designate those working memory elements that are not yet in the working memory. If they were in the working memory they could produce such a rule instance in the conflict set which could make the desired working memory element. For this purpose we use the same Rete network built to support the forward reasoning. However, the node programs are different in this case.

In order to have an instance of a specific rule in the conflict set we have to have the working memory elements satisfying all the condition elements on the left-hand-side of the rule (not concerning negated condition elements). So if we do not have the instance of the desired rule in the conflict set we are missing one or more working memory elements from the node memories. These working memory elements should be produced. The task is twofold: on the one hand all the missing elements have to be found out and the other hand their types and attribute values have to be determined. Since the condition elements of the rules and the working memory elements satisfying them are stored in the Rete network this network is used to find out which elements are missing. Moreover the network can be used to determine the types and the attribute values of the missing elements.

The types and the attribute values are constrained by the tests performed by the nodes on a given path in the network. The starting points of this last step are the terminal nodes selected by the MAKE-Rete network. Traversing the network from a terminal node up to the input node one can collect all the tests performed by the nodes on that path. Sending such a working memory element to the input node which satisfies all these tests this element will reach the terminal node we started the backward traversal from. According to this an empty token with slots to be filled is sent upward[2] trough the network. Due to the possibly shared variables between the condition elements and the MAKE statements some of these slots may be filled according to the variable bindings. All the nodes receive this token from their output and send it upward through their inputs.

As a node receives this token from its output fills some of the slots according to the tests performed by the given node. For example if a node tests a given attribute then it sets such a

---

[2]Naming the inputs and outputs of a node we follow the naming convention used to describe the normal forward operation, according to [Forgy79].

value for that attribute in the token that would satisfy the given test. The type checking nodes set the type fields of the working memory elements in the tokens. However, the above outlined process is a bit more complicated in the case of join nodes.
A token received by a join node always contains more than one working memory element (since it joins two tokens during the forward operation). In the first step every token is split into two parts. The structure and the content of these parts correspond to the structure and the content of the tokens joined by the node during the forward operation. Both node memories are searched for the corresponding token part. If no token is found in the memories then the two tokens are sent upward without change. If one of these tokens is found then only the other token will be propagated further. However, some of the attribute slots of it possibly will be filled according to the variable binding set by the token found in the memory. One token is emitted for every token found in the memory of the opposite side. If tokens are found on both sides then this procedure is executed for both sides. The arrival of one token on the output may result in sending out of several tokens on both inputs.

Conclusion

The work discussed in the paper has shown that the original ideas the Rete match algorithm is based on can be extended to support the backward chaining operation of production systems as well. Since in the case of the Rete match algorithm the working memory is spread over the Rete network there was a strict constraint to use the same network during the backward chaining phase. This paper presented a method that fulfils this requirement. This solution uses the very same network in a bi-directional way.
This extended Rete match algorithm was implemented as a part of a complete inference engine. This implementation provides flexible tools to interface other problem solving techniques. Since it was design following the object-oriented paradigm there are no restrictions on the complexity of the working memory elements. Keeping in mind the efficiency issues the programs were implemented in C++ [Mészáros93].

References

Forgy79 Forgy, C.L.: On the Efficient Implementation of Production Systems, Ph.D. dissertation, Carnegie-Mellon University, 1979.

Forgy82 Forgy, C.L.: Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem, Artificial Intelligence 19, 1982, pp. 17-32.

Mészáros93 Mészáros, T.: A Rete Match Based Interpreter in Object-Oriented Environment, MSc thesis, Technical University of Budapest, 1993. (in Hungarian)

Perlin90 Perlin, M.: Topologically Traversing the Rete Network, Applied Artificial Intelligence 4, 1990, pp.155-177.