From: AAAI-87 Proceedings. Copyright ©1987, AAAI (www.aaai.org). All rights reserved.

TREAT: A Better Match Algorithm for AI Production Systems

Daniel P. Miranker
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Abstract

This paper presents the TREAT match algorithm for AI production systems. The TREAT algorithm introduces a new method of state saving in production system interpreters called *conflict-set support*. Also presented are the results of an empirical study comparing the performance of the TREAT match with the commonly assumed best algorithm for this problem, the RETE match. On five different OPS5 production system programs TREAT outperformed RETE, often by more than fifty percent. This supports an unsubstantiated conjecture made by McDermott, Newell and Moore, that the state saving mechanism employed in the RETE match, condition-element support, may not be worthwhile.

I. Introduction

Production systems are the basis of many expert systems [Brownston et al., 1985]. The growing use of expert systems is well known as is their large computational requirements. Thus it is important to search for more efficient ways to execute production system programs.

In general, a *production system* is defined by a set of rules, or *productions*, that form the *production memory* together with a database of current assertions, called the *working memory* (WM). Each production has two parts, the *left-hand side* (LHS) and the *right-hand side*, (RHS). The LHS contains a conjunction of *pattern elements* that are matched against the working memory. The RHS contains directives that update the working memory by adding or removing facts, and directives that affect external side effects, such as reading or writing an I/O channel.

In operation, a production system interpreter repeatedly executes the following cycle of operations:

1. Match. For each rule, compare the LHS against the current WM. Each subset of WM elements satisfying a rule's LHS is called an *instantiation*. All instantiations are enumerated to form the *conflict set*.
2. Select. From the conflict set, choose a subset of instantiations according to some predefined criteria. In practice a single instantiation is selected from the conflict set on the basis of the *recency* of the matched data in the WM.
3. Act. Execute the actions in the RHS of the rules indicated by the selected instantiations.

In general much of the WM of a production system remains unchanged across production system cycles. Therefore it is worthwhile for the production system interpreter to incrementally compute the contents of the conflict set. The RETE match [Forgy, 1982], briefly outlined in section III , has often been assumed to be the best algorithm for this problem. However, the literature contains no comparitive analysis of the RETE match with any other algorithm and a conjecture made by McDermott, Newell and Moore[McDermott, Newell and Moore, 1978], suggests that the state saving mechanism employed in the RETE match, condition-element support, may not be worthwhile. Section II describes several methods for introducing state into a production system interpreter, including a new method incorporated into the TREAT algorithm called *conflict-set support*. Section IV describes the TREAT algorithm. Section V presents the results of an empirical study comparing the performance of RETE and TREAT for the execution of five different OPS5 programs. For all five programs TREAT required fewer comparisons to do variable binding than RETE. In two instances TREAT required fewer than half.

II. The Development of Matching Algorithms

Figure 1 illustrates an OPS5 rule and WM. In the LHS of the rule the capital letters represent constants, the characters in brackets, pattern variables. Though not illustrated in the example, condition elements may be negated.

A. Relational Database Analogy

A convenient way to describe the primitive operations of a production system algorithm is to make an analogy to relational database terminology. If the WM elements of a production system are considered to be tuples of some universal relationship in a relational database, then it becomes apparent that the LHS of a rule in a production system is analogous to a query in a relational database language.

The constants in a single-condition element may be

```
Rule:                          Initial Working Memory:

(P example-rule                (A 1)
   (A < x >)                   (B 1 2)
   (B < x > < y >)             (B 2 3)
   (C < y >)                   (B 2 4)
   -- >                        (C 3)
   ; no RHS actions)           (C 2)
```

Figure 1: Example Rule System

viewed as a relational selection over a database of WM. We say a WM element *partially matches* a condition element if it satisfies the select operators or the intra-condition element pattern contraints. Consistent bindings of pattern variables between distinct condition elements may be regarded as a database join operation on the relations formed by the selections. The conflict set is the union of the query results of each of the rules in the system.

B. Methods for Introducing State

A difference between database systems and production systems is that database systems usually compute queries one at a time over a large database. In terms of the analogy, a production system continuously computes many queries, as many as there are rules, over a slowly changing, modest size database. To minimize recalculating comparisons on different production system cycles production systems algorithms retain state across cycles. McDermott, Newell and Moore [McDermott, Newell and Moore, 1978] have identified three types of knowledge or state information that may be incorporated into a production system algorithm. A fourth type, conflict-set support is exploited by the TREAT algorithm. In detail these are:

- *Condition Membership*: Associated with each condition element in the production system is a running count indicating the number of WM elements partially matching the condition element. A match algorithm that uses condition membership may ignore those rules that are *inactive*. A rule is *active* when all of its positive condition elements are partially satisfied.
- *Memory Support*: An indexing scheme indicates precisely which subset of WM partially matches each condition element. By analogy, memory support systems explicitly maintain a representation of the relations resulting from the select operations. Later this representation will be called an *alpha-memory*.
- *Condition Relationship*: Provides knowledge about the interaction of condition elements within a rule. By analogy this corresponds to explicitly maintaining the intermediate results of a multiway join.
- *Conflict Set Support*: The conflict set is explicitly retained across production system cycles. By doing so, it is possible to limit the search for new instantiations to those instantiations that contain newly asserted WM elements.

C. McDermott et al.'s Conjecture

McDermott, Newell and Moore conjected that the cost of maintaining the state required for condition relationship exceeds the cost of the comparisons that otherwise would have to be recomputed.

> "It seems highly likely that for many production systems, the retesting cost will be less than the cost of maintaining the network of sufficient tests."[McDermott, Newell and Moore, 1978]

III. The RETE Algorithm

The RETE match[Forgy, 1982] incorporates memory support and condition relationship. Until now, no work has been done to repudiate or confirm McDermott et. al.'s conjecture. Despite that conjecture and a lack of any comparative studies of the RETE match with any other production system algorithm, the RETE match is commonly assumed to be the best algorithm for production system matching.

Briefly, the RETE algorithm compiles the LHSs of the production rules into a discrimination network in the form of an augmented dataflow network. (See Figure 2.) Database operators are used as the operators in the dataflow network. The top portion of the RETE network contains chains of tests that perform the select operations. Tokens passing through those chains *partially match* a particular condition element and are stored in *alpha-memory nodes*, thus forming the memory support part of the algorithm. Following the alpha-memories are two-input test nodes that test for consistent variable bindings between condition elements. By analogy, the two-input nodes incrementally compute the join of the memories on their input arcs. When a token enters a two-input node, it is compared against the tokens in the memory on the opposite arc. Paired tokens with consistent variable bindings are stored in *beta-memories*. Tokens that propagate from the last beta-memory in the network reflect changes to the conflict set. The reader is encouraged to see [Miranker, 1987b] for a more detailed explaination.

A. Tradeoffs in RETE

The advantages of RETE are; the large amount of stored state minimizes the number of times two WM elements will be repeatedly compared and similar rules will compile to similar networks, allowing sharing of network structures.

The primary disadvantage of RETE is that when a WM element is removed the stored state must be unwound, often requiring the repetition of the precise sequence of operations that were performed upon its addition. Other disadvantages are; the size of the beta-memories may be combinatorially explosive, sharing

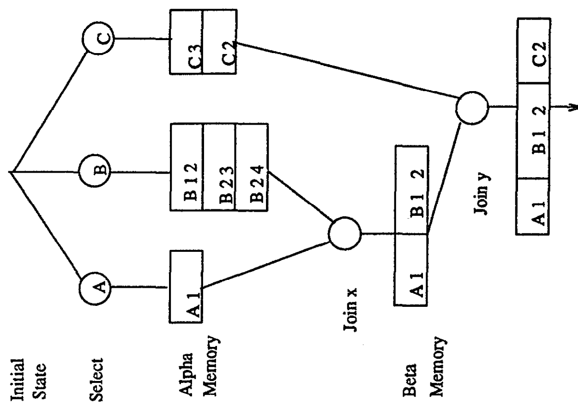


Figure 2: RETE Illustration

network structure is not advantagous in a parallel environment due to contention and/or communication costs, to maintain consistent state in the network RETE must perform extensive computation for rules that are inactive, thus not exploiting condition support.

The incentive to develop TREAT was created by the difficulties associated with using RETE on parallel computers. [Stolfo and Miranker, 1984, Gupta, 1984]. In a sequential computer RETE tokens may be manipulated by simple memory accesses. In a parallel computer manipulating tokens can involve contention and costly communication steps.

IV. The TREAT Algorithm

A. Conflict Set Support

To exploit conflict set support two observations must be made. Assume for the moment that there are no negated condition elements in the production system. If the only action of a fired rule is to add a new WM element, then the conflict set remains the same except for the addition of new instantiations that contain the new WM element. In the example below, adding (A 2) results only in instantiations containing (A 2). The second observation is that if the only action of a fired rule is to delete a WM element, then no new rules will be instantiated. Some instantiations may become invalide. These will contain the removed WM element.

The essence of the TREAT algorithm is to exploit these observations. Additions to WM may be used as seeds to initiate a constrained search for new instantiations. Deletions are processed by examining the conflict set directly and removing any instantiation that contain a deleted WM element. (See Figure 4.)

B. Negated Condition Elements

Allowing negated condition elements slightly complicates the algorithm. The TREAT algorithm must consider four cases, the addition or deletion of WM elements that partially match both positive or negated condition elements. The cases concerning positive condition elements remain unchanged from the previous section. The handling of negated condition elements is described in the abstract algorithm and in detail in [Miranker, 1987b]. Briefly, in the other two cases when changed elements partially match negated condition elements the condition is temporarily considered to be positive and the change acts as a seed to create possible instantiations. If the change is an addition, instantiations are removed from the conflict set. If the change is a deletion, instantiation may be entered into the conflict set.

C. Detailed TREAT Algorithm

The TREAT algorithm exploits condition membership, memory support and conflict set support. All the condition elements in a production system are numbered. The number associated with a condition element is called the condition element number (CE-num). Information relevant to condition elements is stored in arrays indexed by CE-num. Alpha-memories similar to those used in RETE are used to form the memory support part of the algorithm, but rather than existing amorphously in a network they are formed explicitly as a vector, each entry containing an alpha-memory. The alpha-memories are broken into three partitions: old, new-delete and new-add.[1] The old partition, (old-mem), contains the partially-matched elements that have already been processed. During the act phase, elements are not added to the old-mem but to the memories in the add and delete partitions, (new-add-mem and new-del-mem). The calculation of the contents of the alpha-memory could be done by building the top portion of a RETE network. The implementation reported here used a hash function whose argument is the value of the first attribute in an OPS5 WM element.

To incorporate condition support, whenever an old-mem is updated a test is made to see if its size has become zero or nonzero. If the critical change is detected, the size of each of the old-mems for the rule is examined and the set of active rules is updated accordingly.

When an alpha-memory of an active rule is altered and the change corresponds to one of the three cases where a search for instantiations is required, then the search takes place among the changed (new) alpha-memory, the old-memories that correspond to the remaining condition elements in the rule. Figure 3 contains an abstract program for the TREAT algorithm.

D. Join Optimization

The join operation is commutative and associative. Thus when searching for consistent variable bindings the alpha-memories may be considered in any order. There are many multiway join optimizations[Ullman,

[1] In the implementations reported here, these are formed by three separate vectors. However, a vector of structures would probably have resulted in better paging characteristics.

1. Act: Set CHANGES to the WM updates required by the RHS.
2. For each WM change in CHANGES do;
   (a) For each condition element, CE_i do;
      - If the partial match of the element against CE_i is successful
        and if addition to working memory
        then add WM-element to new-add-mem[CE_i].
        else add WM-element to new-del-mem[CE_i].

   end for;

   end for;
3. Match: Process deletes.
4. For each nonempty del-mem do;
   (a) Set cur-ce = CE-num of the selected memory.
   (b) Set old-mem[cur-ce] = old-mem[cur-ce] - new-del-mem[cur-ce].
   (c) If size of old-mem[cur-ce] = 0 then update-rule-active.
   (d) Case: If CE corresponding to the new-del-mem is positive or negated.
      i. Positive: Search conflict set for instantiations containing the deleted WM-elements. If found remove them.
      ii. Negative: If the affected rule is active, then perform search for new instantiations by searching new-del-mem[cur-ce] and the old-mems that correspond to the remaining condition elements that are part of the affected rule. Check that the new instantiations are not invalidated by elements in old-mem[cur-ce].

   end for;
5. Match: Process adds.
6. For each nonempty add-mem do;
   (a) Set cur-ce = CE-num of the selected memory.
   (b) Set old-size = the size of old-mem[cur-ce].
   (c) Set old-mem[cur-ce] = old-mem[cur-ce] + add-mem[cur-ce].
   (d) If size of old-mem[cur-ce] = 0 then update-rule-active.
   (e) If the rule is active, then perform search for new instantiations by searching new-add-mem[cur-ce] and the old-mems that correspond to the CEs of the remaining CEs that are part of the affected rule.
   (f) Case: If CE corresponding to the del-mem is positive of negated.
      i. Positive: Add these new instantiations to the conflict set
      ii. Negative: Search the conflict set for each of the new instantiations and remove them if found.

   end for;

Figure 3: Abstract Algorithm Illustrating TREAT

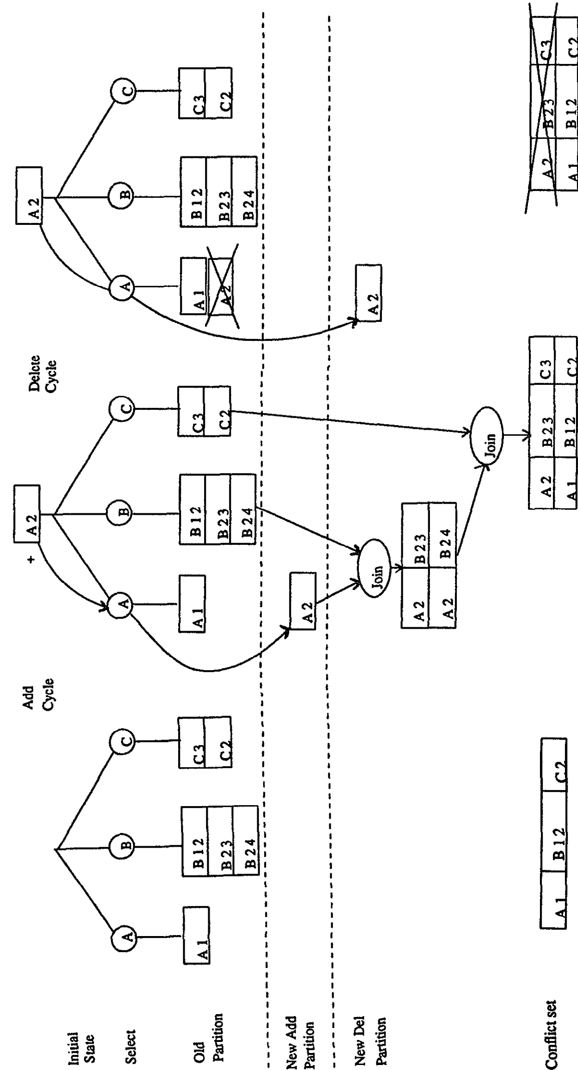


Figure 4: TREAT Illustration

1982]. However, in OPS5 the small size of the alpha-memories and the very small number of WM changes per cycle, (an average of 2.5), dictates that for an optimization to be useful it must be simple to compute and result in a deterministic ordering of the alpha-memories. Three orderings were studied. Static-ordering, where the alpha-memories where considered in the lexical order of condition elements. Seed-ordering, where the changed alpha-memory is considered first, since in OPS5 these changes are almost always small and considering them first will greatly constrain the search. The third method based on semi-join reductions was not successful and will not be detailed. Note that the use of join optmizations allows TREAT to be used effectively for other production system languages. If a system is temporally-nonredundant the search for instantiations may still be performed in a different but still optimal order.

E. An Example using TREAT

Figure 4 shows the initial state created by the TREAT algorithm as well as the activity during the addition and deletion of a WM element (A 2). The activities of TREAT and RETE in this case are identical except that TREAT does not maintain beta-memories. However, the beta-memories do not contribute constructively to the computation of the new instantiation. To be fair, note that for the add example had the WM element partially matched the "C" branch of the network RETE would have searched only a beta-memory while TREAT would have had to search both remaining alpha-memories. For a delete, the RETE match must recompute the tokens stored in the beta-memories and then delete them. TREAT outperforms RETE during deletetions by directly updating the alpha-memories and the conflict-set. The key issue is; does the number extra comparisons performed by TREAT while searching for instantiations exceed the number of comparisons performed by RETE while processing deletions? The results of an empirical study of this question are presented in the next section.

V. TREAT vs. RETE

This section presents quantitative measurements of identical runs of OPS5 programs on several different OPS5 interpreters. The RETE-based OPS5 interpreter is the familiar one distributed by Forgy from Carnegie Mellon University. The TREAT-based OPS5 interpreters were written at Columbia University.

A. Synopsis of the Benchmarks

Five OPS5 programs representing a wide variety of characteristics were obtained from diverse sources. Some characteristics of these systems are summarized in Figure 5.

- MAB: The familiar Monkeys and Bananas program[Brownston et al., 1985].
- Waltz: A set of rules that perform Waltz constraint propagation[Winston, 1979].
- Mapper: The Mapper is program that will assist a tourist to navigate Manhattan's public transportation system. The Mapper has an extremely large WM. The maps for nearly the entire Manhattan bus and subway systems are stored as 1124 WM elements.
- Mud: A system written at Carnegie Mellon University to to analyze the castings from oil wells. It should also be noted that this is precisely the same system used by Gupta [Gupta, 1986] in his study of parallelism in OPS5.
- Mesgen: A natural-language program written by Karen Kukich at the Univ. of Pennsylvania that takes Dow Jones figures and converts them into text describing the course of a trading day.

	Number of rules	Number of conditions	Average WM Size	Cycles in test run	Average CS Size
MAB	13	34	11	14	21
Mud	884	2134	241	972	
Waltz	33	130	42	71	193
Mesgen	155	442	34	138	149
Mapper	237	771	1153	84	595

Figure 5: Summary of the Gross Characteristics of the Studied Systems

B. Counting Comparisons for Variable Bindings

It has been reported that 90% of the execution time of a production system is spent in the match phase. Evidence indicates that in the RETE-OPS5 implementation the majority of the match time is spent in performing variable binding and in maintaining the beta-memory nodes [Gupta, 1986]. The critical difference between the algorithms is the method used to handle variable binding.

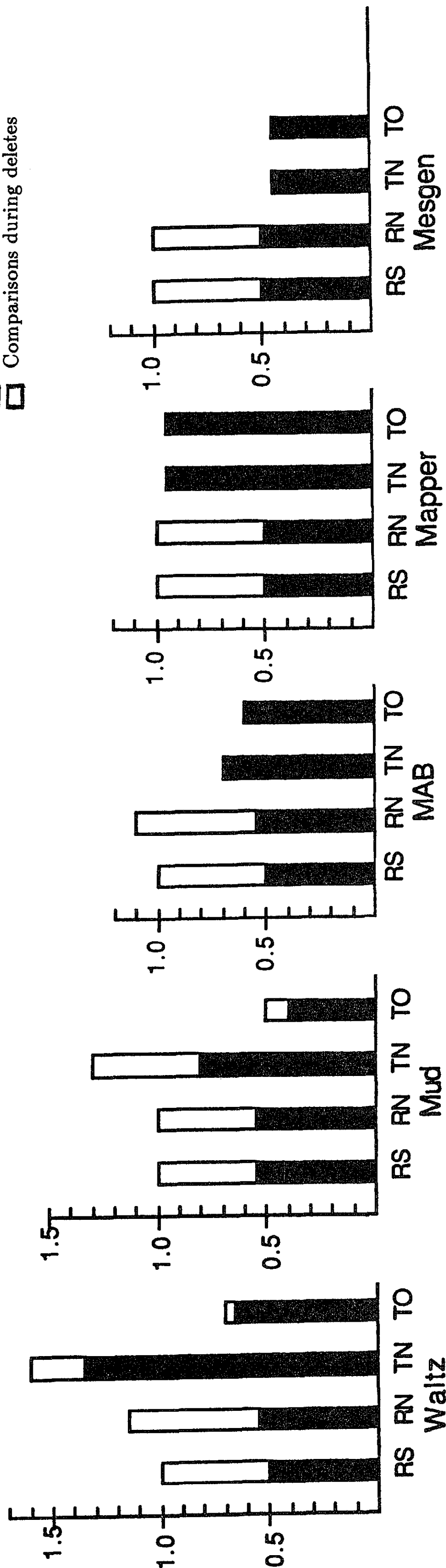
The graphs in Figure 6 show the number of comparisons required to do variable binding for each of the OPS5 programs for two variations of each algorithm. The bars are normalized to the number of comparisons required by execution of the stardard RETE implementation. The dark portion of the bars indicates the number of comparisons required during the add cycles, the light portion, the number for the delete cycles.

The RS bars represent the performance of the standard release of RETE-based OPS5. The RN bars indicate the performance of RETE without sharing. We see that sharing does not contribute significantly, if at all, to the variable binding phase of the RETE match.

The TN bars represent the performance of TREAT without any optimizations. Search is performed in lexical order. Depending on the system this version of algorithm may perform better or worse than the RETE match. Thus, some run-time optimization is necessary.

The TO bars represent the performance of TREAT using the seed-ordering heuristic. Inspection of the graphs shows that TREAT with seed-ordering always performed better than RETE even on a sequential computer. Except for the Mapper[2] with it's very large WM the algorithm requires roughly half of the comparisons required of the RETE match. Note for each successful comparison performed by the RETE match there is the additional expense of maintaining a beta-memory.

[2] There is evidence that with the introduction of hashing the performance of the Mapper would be closer to that of the other systems.



VI. Conclusion

In many cases TREAT without any optimization outperforms the RETE match. With seed-ordering optimization, TREAT always outperforms RETE. In two intances TREAT required less than half of the comparisons to perform variable bindings than RETE. This does not consider the additional cost of maintaining the beta-memories. Since the algorithms are nearly identical in all other respects, it may be concluded that TREAT is a bettër production system algorithm in both time and space. Further this study supports the conjecture made by McDermott, Newell and Moore that condition-element support many not be worthwhile.

References

[Brownston et al., 1985] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5.* Addison Wesley, Reading, Mass., 1985.

[Forgy, 1982] Charles L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Patern Matching Problem. *Artificial Intelligence*, 19:17-37, 1982.

[Gupta, 1984] Anoop Gupta. Implementing OPS5 Production Systems on DADO In *Proceedings International Conference on Parallel Processing*, IEEE Computer Society Press, 1984.

[Gupta, 1986] Anoop Gupta. *Parallelism in Production Systems.* Ph.D. Thesis. Carnegie Mellon University, Department of Computer Science 1986.

[Miranker, 1987a] Daniel P. Miranker. *TREAT:A Better Match Algorithm for AI Production Systems; Long Version.* Technical Report, Department of Computer Sciences, University of Texas at Austin, April 1987.

[Miranker, 1987b] Daniel P. Miranker. *TREAT:A New and Iefficient Match Algorithm for AI Production Systems* Ph.D. Thesis, Computer Science Dept. Columbia University. Available as Technical Report TR-87-03, Dept. of Computer Sciences, University of Texas at Austin, Jan. 1987.

[Stolfo and Miranker, 1984] Salvatore J. Stolfo and Daniel P. Miranker. DADO:A Parallel Processor for Expert Systems. In *Proceedings International Conference on Parallel Processing.* IEEE Computer Society Press, 1984.

[Ullman, 1982] J.D. Ullman. *Principles of Database Systems.* Computer Science Press, 1982.

[McDermott, Newell and Moore, 1978] J. McDermott, A. Newell and J. Moore. The Efficiency of Certain Production System Implementations. In *Pattern-directed Inference Systems.* Academic Press, 1978.

[Winston, 1979] P. H. Winston. *Artificial Intelligence.* Addison Wesley, 1979.