Constraint Satisfaction for Production System Match

Mark Perlin
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
perlin@cs.cmu.edu

Abstract*

RETE is a classical AI algorithm for incremental production system match. This task may be viewed as a constraint satisfaction problem (CSP). RETE employs a backtracking augmentation strategy, applying its constraints strictly from left-to-right. Since no lookahead is employed, costly intermediate tuples are often constructed that provably cannot lead to a valid rule instantiation.

Arc Consistency (AC) is another classical AI algorithm. AC is a preprocessing step, performed prior to CSP backtrack search, that provides lookahead. Specifically, AC detects and removes those variable domain values that cannot contribute to a complete, valid solution. This filtering can reduce the effective size of each domain, hence the effective size of their product; this product bounds the CSP cost.

This paper improves production system match by integrating these two classical algorithms, incorporating AC into RETE. This novel approach combines the constraint graphs of RETE and AC into a single network, which is then incrementally updated. Empirical studies show the technique to be most efficacious with expensive rules. Thus, by employing the lookahead from AC preprocessing, in many cases costly RETE computation can be effectively reduced.

1. Introduction

Production systems (PS) [13] are a forward-chaining rule-based computational architecture, often used in AI to model human reasoning. A typical PS interpreter, such as OPS5 [2], operates in a series of Match-Select-Apply cycles. The match step, which compares all rules against all the Working Memory (WM) data, entails a combinatorial constraint satisfaction problem (CSP) and is thus the computational bottleneck.

RETE [3] is a classical AI algorithm for incremental production system match. RETE employs a backtracking augmentation strategy, applying its constraints strictly from left-to-right. Since no lookahead is employed, costly intermediate tuples are often constructed that provably cannot lead to a valid rule instantiation.

Arc Consistency (AC) [8] is another classical AI algorithm. AC is a preprocessing step, performed prior to CSP backtrack search, that provides lookahead. Specifically, AC detects and removes those variable domain values that cannot contribute to a complete, valid solution. This filtering can reduce the effective size of each domain, hence the effective size of their product; this product bounds the CSP cost. AC finds application, for example, in machine vision scene labelling problems, where the filtering procedure [27] can often solve the CSP without further backtrack search.

To what extent can these two efficient CSP methods, RETE and AC, be integrated into a single, more efficient, algorithm? How can such a combination provide useful insights into the general understanding and implementation of efficient AI algorithms? And how can such a combination be effectively applied in improving the performance of AI systems?

This paper describes how incremental arc consistency can be incorporated into RETE. This RETE/AC integration is enabled by recent results on factoring arc consistency [21] that reduce the AC preprocessing costs from quadratic in the domain set size to linear. The construction and update of the integrated RETE/AC network is described, and a directed arc consistency update is introduced that allows for more efficient incremental operation. Empirical studies are presented, comparing alternative RETE/AC constructions, and demonstrating the efficacy of the approach. The comparative advantages and disadvantages of incorporating AC into RETE match are discussed, and possibilities for future research are explored.

* This research was sponsored by the National Library of Medicine of the National Institutes of Health under Contract 5R29 LM04707-05, and by the Pittsburgh NMR Institute.
2. Preliminaries and Nomenclature

A variable \( X_i \) has a domain \( D_i \) of possible values.

A constraint on one or more variables determines allowable combinations of variable values. Specifically, a unary constraint on \( X_i \) determines the allowable subset of \( D_i \), and a binary constraint on \( X_i \times X_j \) is a relation \( R_{ij} \) determining the allowable subset of \( D_i \times D_j \).

A constraint satisfaction problem (CSP) is comprised of a set of \( n \) variables \( \{X_i\} \), and their domains \( \{D_i\} \), together with unary and binary constraints on the variables. A valid \( n \)-tuple in \( \prod D_i \) satisfies all constraints. Here, the solution to a CSP shall be the set of all valid \( n \)-tuples, rather than just one \( n \)-tuple.

Solutions to CSPs are generally found by backtrack search. The backtrack method imposes a linear ordering on the variables, and then builds the valid \( n \)-tuples by induction on \( n \). A partially valid \( k \)-tuple, \( k \leq n \), satisfies all those constraints which reference only the first \( k \) variables.

Backtracking operates by iteratively augmenting partially valid \( k \)-tuples to partially valid \( (k+1) \)-tuples.

**RETE** [3] is an incremental backtracking algorithm for the CSP arising in production system match. RETE may be viewed [20, 28] as the partial evaluation [5] of the requisite backtrack search control into a dataflow network that is used as template for finite differencing [14]. This RETE network contains the variables (often called \( \alpha \)-nodes) as leaves, and nonterminal join nodes (often called \( \beta \)-nodes) for tuple augmentation. An absence node is a specialized nonaugmenting join node which enforces the nonmatching of a negated variable. RETE is designed to incrementally recompute the CSP solution set from slowly changing variable domain sets; the finite differencing is the key to RETE's average case efficiency.

Arc Consistency (AC) [8] is a filtering step for CSPs, done prior to backtrack search. A value \( v \) in domain \( D_i \) is said to be unsupported if there is some \( R_{ij} \) for which \( v R_{ij} \) is empty. When a value lacks support, it cannot enter into any valid \( n \)-tuple. AC iteratively visits the domain sets, removing unsupported values. This reduces the effective size of the domain sets, hence the size of the largest CSP product set. (For example, a square root reduction for each domain set would halve the exponent of the full Cartesian product.) AC-4 [11] is the optimal algorithm for AC, having cost \( O(e d^2) \), where \( e \) is the number of binary relations, and \( d \) is the size of the largest domain.

The variables of a CSP and their relations may be organized into constraint graphs. The nodes of such graphs may include the variables, the arcs between variable pairs, or the joins which group the first \( k \) variables. Edges are used to connect the nodes. Nodes serve as templates for instantiation, as in the value instance of a variable node, or the \( k \)-tuple instance (often termed "partial instantiation") of a join node. In this paper, it will prove useful for the instances themselves be graph elements, having edges to other instances that mirror their node's topology. For example, the \( k \) leaves of a join instance (when viewed as a subtree) are precisely its \( k \)-tuple components. Also, the AC-4 algorithm used here [21] is a version that explicates the \( R_{ij} \) relations into concrete links between instances.

3. RETE's Constraint Organization

RETE has a dataflow network that provides a structural template for incremental backtracking. The backtrack organization imposes a linear ordering on the variables, which are processed in ascending sequence \( X_1, X_2, ..., X_n \), from left to right.

A partially valid \( k \)-tuple satisfies all constraints which reference only the first \( k \) variables. Backtracking generates a candidate \( k \)-tuple by augmenting a partially valid \((k-1)\)-tuple with an instance of variable \( X_k \) (that satisfies \( X_k \)'s unary constraints). Therefore, a join node tests just those binary constraints whose maximal reference is to the \( k \)th variable. This constraint testing strategy partitions the constraints into a linear organization. For the example constraint set shown in Figure 1, the corresponding linear RETE network is shown in Figure 2.

**Variables:** One, Two, Three, Four.

**Constraints:**
- \((= (One \ a) \ (Two \ a))\)
- \((= (One \ b) \ (Three \ b))\)
- \((= (One \ c) \ (Three \ c))\)
- \((= (One \ d) \ (Four \ d))\)
- \((= (Two \ e) \ (Four \ e))\)

**Figure 1.** Binary equality constraints on the variables named One, Two, Three, and Four. Each constraint is given by a predicate, and by a pair of variable/attribute accessor descriptors.

**Figure 2.** RETE's linear organization of the example binary constraints in Figure 1.
4. AC's Constraint Organization

Prior to backtrack search, arc consistency can explore the variable instances, removing those which are unsupported. This introduces lookahead into the linearly sequenced backtrack procedure, allowing variable $X_j$ to prune the domains of variables $X_i$, even when $i < j$. To do this, all constraints referencing both variables in the pair $<X_i, X_j>$ are grouped into a single relation $R_{ij}$. ($R_{ij}$ can either be known in advance, or dynamically generated by applying constraints to incrementally changing domain values.)

These $R_{ij}$ relations form arcs between the variable nodes. The potential for an arc between any pair of variables leads to a quadratic constraint partitioning. For the example constraint set given in Figure 1, the corresponding quadratic AC graph is shown in Figure 3.

4.1. Factorization

In standard AC, each arc in Figure 3 represents a single edge in the constraint graph. Using Mohr and Henderson's optimal AC-4 update algorithm [11] with $O(ed^2)$ complexity, the $d^2$ factor leads to quadratic complexity in the size of the variable domains. While appropriate for a single pass backtrack solution, this bound may be unacceptably high for incremental CSPs, where just one domain value may change, yet incur quadratic AC update cost in the domain (e.g., working memory) size.

Recently, a linear bound $O(ed)$ has been described for certain CSP problems [21]. This reduction in complexity can be obtained for certain CSPs in which the $R_{ij}$'s can be factored into smaller relations. For example, when the $R_{ij}$ are all equality relations, they can always be factored. The arcs in Figure 3 can then be reinterpreted as nodes (not edges) in the constraint graph, and AC-4 again used to obtain the (now optimal) linear complexity in $d$. Gupta has shown that 90% of production system constraints test equality [6]. Thus the equality-based factorization of production system constraint graphs by restricting the $R_{ij}$ to equality tests can yield significant speedups, despite the loss of some filtering power.

4.2. Directional AC

In performing AC update, as detailed below, there are situations in which the full $O(ed)$ bound can be reached. In the worst case, when cycles between variables are repeatedly traversed, the cost can be proportional to the size of WM. This expense may be unacceptable for RETE's small, incremental changes to WM.

One solution is to make AC's lookahead directional. Since RETE's constraints are applied in a strictly left-to-right direction, the requisite lookahead can be applied in a strictly right-to-left direction. This may sharply curtail the efficacy of constraint satisfaction's prefiltering. In fact, the case shown in Figure 4 demonstrates how directionality limits the filtering. However, since the updates now only affect the leftward transitive closure of the domain values, the update cost can be greatly reduced.

5. Building the RETE/AC Network

To build the combined RETE/AC network, a rule's linear join constraint graph is connected to its quadratic arc constraint graph via the common variable nodes. (The absence nodes' negated variables are not connected to any arcs.) Figure 5 shows the RETE/AC network for the example constraints in Figure 1, which combines the join constraint graph of Figure 2 with the arc constraint graph of Figure 3.
A rule's RETE/AC network can be recursively constructed from the rule's constraints (a detailed algorithm appears in the Tech Report [22]). Essentially, the preorder expansion linearly partitions the constraints for the join nodes, while the postorder return performs the quadratic partitioning, and constructs the constraint graph. The resulting graph structure may be used for either unfactored or factored AC topology, and for either undirected or directed AC update.

6. Updating the RETE/AC Network

As the changes to WM are made by the production system, they are intercepted by the AC system. Before an update to the RETE/AC join network is issued, the AC filtering process is first run. AC changes between active and inactive (and certain other events) are then passed on for the subsequent RETE update. A more detailed exposition follows.

6.1. AC Phase

When a WM element is added or deleted, the event is enqueued for subsequent processing. Changes to negated variables bypass the AC filter, and move directly to the RETE queue. When a nonnegated variable instance is inserted, it is spliced into the AC relation network, and put on the AC queue. This requires computation by the variable's neighboring arc nodes, to determine and construct the appropriate relation links. With factored AC, new arc instances may be created. When a nonnegated variable instance is deleted, it is spliced out of the AC relation network, and its neighboring instances placed on the AC queue.

When an update finally occurs, the AC filtering propagates from the AC queue. Changes between the active and inactive states are monitored. When a variable instance changes to active, it is put on the RETE queue as an insertion event. Conversely, a change to inactive enqueues an instance as a deletion event. An instance v is said to have no active support if there is some Rij for which vRij has no active members.

With undirected AC, the update is performed in three steps:
1. The connected components of the enqueued instances are initialized. The current activation state is recorded, and then set to ACTIVE. (This is necessary to detect and break cycles of unsupport.) Any instance which is unsupported is put on the AC control queue.
2. A modified AC-4 is then run from the AC control queue. Each cycle, an instance is dequeued and examined. If it is active but has no active support, it is deactivated, and all its neighbors are placed on the control queue; otherwise, it is left alone.
3. After quiescence, the components are again visited. All changes between active and inactive are enqueued for RETE, as described above.

With directed AC, the update is much simpler. The AC queue is maintained in topological order from right to left (i.e., decreasing variable order). As instances are dequeued, those which have changed their activation status are enqueued for RETE, and then propagate their leftward neighbors via the AC queue.

6.2. RETE Phase

The RETE update begins once the AC update has completed. Generally, there will be insertion and deletion events enqueued for variable instances. These propagate from left to right to form candidate joins in RETE's natural topological [17] or other [3] traversal. If hashed-equality RETE [23] is used, then memory is partitioned according to the value indices determined by the equality constraints at the successor join nodes. This eliminates explicit equality testing at joins, and can significantly reduce Cartesian product formation.

7. An Example Trace

As an illustrative example of how RETE/AC can reduce production system match computation, consider the simple constraints given in Figure 6. The corresponding constraint graph is shown in Figure 7A. For this example, we elect to use factored relations, directed AC, and hashed-equality RETE.

Variables: One, Two, Three.
Constraints:
(= (One x) (Two x))
(= (One y) (Three y))

Figure 6. Binary equality constraints for an illustrative example of RETE/AC's operation.

First, we add assertion p=(One x=a y=b) to WM, and update. In ordinary RETE, this would install p as an instance of variable One. In RETE/AC, however, we first determine p's AC relations, and explicate these relations by connecting p to the requisite instances of One's neighboring Arc nodes. Labelling the Arc node having the x attribute constraint X, and the other Arc node Y, this splicing in of p creates the instances X-a of Arc X and Y-b of Arc Y. As shown in Figure 7B, since these Arc instances do not have active support, assertion p remains inactive, and is not presented to the RETE network.
Next, we add assertion \(q=(\text{Two } x=u)\) to WM, and update. In ordinary RETE, this would install \(q\) as an instance of variable \(\text{Two}\), and join with \(p\) to form a partially valid 2-tuple. With RETE/AC, though, the AC phase first splices \(q\) into the AC relation graph. This connects \(q\) to Arc instance \(X=a\). Directed AC update now determines that \(q\) has active support, and gives active support to \(X=a\). However, \(p\) is still unsupported via inactive \(Y=b\). Therefore, \(q\), but not \(p\), is now presented to the RETE network, yielding the state shown in Figure 7C.

Finally, we add assertion \(r=(\text{Three } y=b)\) to WM, and update. The AC phase intercepts \(r\), and determines its AC relations, splicing it into the AC relation graph by connecting it to Arc instance \(Y=b\). \(r\) is inactive, but trivially has active support, since directed AC looks only at its (nonexistent) right neighbors. This change is sent to RETE, enqueuing \(r\), and propagated via the directed AC update to Arc instance \(Y=b\). \(Y=b\) similarly changes to the active state, and then propagates via the directed AC update to \(p\). Since \(p\) is inactive, but now has support from both neighboring Arc nodes, it is (finally) enqueued for insertion with RETE. The RETE update now occurs, with \(p\), \(q\), and \(r\) all active, and (via hashed-equality indexing) constructs the valid 3-tuple shown in Figure 7D.

The key observation is that \(p\) was not allowed to contribute to the RETE match until there was some evidence from rightward constraints that \(p\) might, in fact, lead to a valid \(n\)-tuple. When both of \(p\)'s equality constraints \((X \text{ and } Y)\) had active support, this possibility was no longer ruled out, and \(p\) was asserted to the RETE.

8. Empirical Observations

A multi-purpose RETE/AC was programmed in Common LISP [25] using the above algorithms for building and updating the RETE/AC network. Sufficient generality was implemented to enable comparisons of alternative operating modes, and to assure compatibility with all constraint sets specifiable in the OPS5 production system language [2]. The program was constructed primarily for algorithmic comparisons, and thus was not highly optimized.

The production rule sets from standard benchmark programs, including Monkey and Bananas, Tourney, and (some of) Rubik, were studied. Benchmark timings were conducted on a SUN3/260, and detailed node-level profilings were also obtained. For each rule, the six way experiments of (Standard RETE, and Hashed-Equality RETE) versus (No AC, Directed AC, and Undirected AC) were performed. More selective studies also compared factored versus unfactored AC relations.

Each line of Table 1 shows the benchmark timings for one rule in Tourney, using Standard RETE. The rules are ranked by descending time cost. The percentages are the ratios of Directed to No AC, and Undirected to No AC, respectively. Table 2 shows analogous timings for Hashed-Equality RETE.

The inexpensive rules, appearing in the lower portions of the tables, show the overhead of AC in RETE. What is far more significant here, however, is the efficacy of RETE/AC with the key bottleneck expensive rules. While the improvement is not dramatic, the data clearly show that

- Undirected AC can speed up Standard RETE (Table 1), and
- Directed AC can speed up Hashed-Equality RETE (Table 2)

for highly combinatorial rule matching.

To understand this behavior, more detailed profiling of unit computations was done. Study of Tourney rule \(\text{North}=\text{Pick-One-3}\), corresponding to the third row of Table 1, suggested why undirected AC was successful with Standard RETE. The profiling showed that undirected AC reduced the average variable domain size from 0.525 with no AC to 0.170, at a cost of 30 AC updates per RETE update. This reduced RETE's average Cartesian product formation per update from 164 to 82. (The comparable figures for Directed AC were 0.538 and 178, similar to No AC.) Thus undirected AC effectively applied rightward constraints to reduce the domain sizes, thereby reducing the overall cost of the CSP.
Similar detailed profiling suggested that the efficacy of directed AC in improving Hashed-Equality RETE was more likely due to reduced data flow. For example, with rule North=Pick-One-3 (corresponding to the fourth row of Table 2), directed AC gave a 20% decrease in domain data flow, yielding a 7% decrease in RETE Cartesian product data flow.

As expected from the quadratic vs. linear relationship, the profiling also showed that the unfactored arc topology can entail several orders of magnitude more AC updates than the factored equality relations. For expensive rules, unfactored AC proved unworkable.

### 9. Discussion

What are the inherent advantages and disadvantages of RETE/AC versus RETE? In what new directions can this research lead? This Section discusses these issues, and mentions other related work.

#### 9.1. Advantages and Disadvantages

There are two major advantages of using AC as a filter to RETE. First, the effective sizes of the variable domain sets can be reduced, which in turn reduce the effective size and time cost of the CSP. Second, by potentially reducing the number of RETE's partially valid instances, the time and space costs of intermediate join construction are reduced.

Similarly, there are two potential disadvantages with using RETE/AC. The first is the overhead cost of the preprocessing AC construction and update. This can be minimized, as shown above, by using factored AC relations and directed AC update. Second, is that the effective data flow might be increased if the same variable instance is repeatedly activated and inactivated. Our empirical studies, however, actually showed a decrease in variable data flow. Nonetheless, the use of reactivatable RETE memory, described below, would help reduce this cost.

As is clear from our empirical observations, RETE/AC is not an across-the-board optimization to be applied to an entire rule set. Rather, heuristically or empirically, RETE/AC can be used on a rule-by-rule basis to enhance RETE's performance for particular arrangements of rule constraints. Significantly, the studies showed RETE/AC as most effective for precisely the most expensive production rules.

### 9.2. Future Work

To decrease the cost of possibly increased effective data flow, a data organization employing reactivatable instance memory [16] can be used. With reactivatable instances, a deletion of a partially valid instance does not actually destroy it. Instead, it continues in an inactive state, connected to the variables which constructed it. When one of these variables is reinserted, the inactive instance may be reactivated (and initiate new join computation). The result is a space-for-time tradeoff, which substitutes TMS-like reactivations for redundant, time intensive, match computation.

Incorporating AC into production system match may find application with matching algorithms other than RETE. TREAT [9], for example, does not cache partially valid joins between match cycles. It may therefore be even more suitable than RETE for employing constraint satisfaction, benefiting from all its advantages, and few of its disadvantages. Specifically, increased effective data flow would not be a problem. RETE's dual algorithm, Linear MatchBox [19] can also use AC as a preprocessing step.

Constraints might be employed in other novel ways. For example, variable ordering heuristics [7, 15, 24] for
RETE's backtrack search could be modified to account for the influence of constraint satisfaction. Further, the methods presented here are not limited to binary constraints (though OPS5 is). Using Generalized Arc Consistency (GAC4) [12], multary constraints can be accommodated.

9.3. Related Work

Most of the related literature has already been discussed above. Directed AC methods have been used for updating constraint trees [1]. It might even be possible to approximate RETE's constraint graph by such a tree. The results of backtrack-free search [4] are not applicable here, since RETE's CSP requires all solutions. Dynamic AC methods for incrementally varying variables and relations have been described [10]. Analogous techniques are used here, though it is the variable domains, and not their topologies, that are modified. Tambe has restricted the expressivity of RETE match so that arc consistency essentially performs match [26]; here, however, no such restrictions are made, and fully general match expressivity is retained.

10. Conclusions

This paper integrated two classical AI algorithms for CSP, RETE and AC, to form the new RETE/AC algorithm, by adapting AC filtering methods for use with RETE. Interestingly, the implementation exploited striking commonalities [18] in these two data-driven algorithms. For example, hashed-equality partitioning was used in both AC's and RETE's memory organizations.

The key contribution was the novel use of rightward constraints in RETE to provide lookahead, and thus filter out provably invalid n-tuple computation. The mechanism for this constraint satisfaction elimination was arc consistency. This approach may prove effective in other incremental production system CSP algorithms, including TREAT and Linear MatchBox.

Empirical studies demonstrated the efficacy of the integrated approach. In fact, the more expensive the rule, the more efficacious was the RETE/AC technique. By effectively using constraint satisfaction in production system match, two complementary, though previously unrelated, lines of AI algorithm research have been brought together. Hopefully our results will stimulate further communication, research, and applications in these areas.

Acknowledgements

Dave Rager performed many of the benchmark experiments. Brian Milnes clarified some semantic issues regarding OPS5's RETE matcher. A careful reading by Mark Kantrowitz greatly improved the presentation.

References


