Incremental Binding-Space Match: The Linearized MatchBox Algorithm

Mark Perlin
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract*

We introduce Linearized MatchBox, a new incremental algorithm for conjunctive match. As with the parallel MatchBox algorithm [21], Linearized MatchBox operates in binding-space, rather than in the assertion-space of RETE-like [3] matchers. Linearized MatchBox is better-suited to serial computers than MatchBox, and is designed to reduce the number of broadcast operations to the binding-tuples.

Linearized MatchBox is built on a linear recursion that is dual to RETE's. The version presented here is restricted to equality tests, but does support negated conditions. In this paper, we:

- Motivate and introduce the Linearized MatchBox algorithm.
- Describe its relation to other matching algorithms.
- Compare it with MatchBox, and show why it is better for serial computers.
- Give quantitative criteria for determining when Linearized MatchBox is applicable to a particular rule matching situation.
- Describe our current implementation.
- Discuss its utility in alleviating the knowledge bottleneck for learning problem solvers such as PRODIGY.

Linearized MatchBox makes binding-space match a realistic alternative to assertion-space match on conventional computer architectures. It should find its greatest utility as one effective matching modality in a multi-strategy rule matching system.

1. Introduction

Forward chaining production systems (FC/PS) are an important inference mechanism in Artificial Intelligence (AI). They are used to build knowledge-based systems in such diverse domains as computer configuration [10], factory scheduling [5], and robot planning [8], and form the computational engine for hundreds of expert system applications. The FC/PS cycle repeats the following steps until no rule instances are selected:

1. match all the rules against all the assertions in the current working memory (WM) state, forming all possible rule instances;
2. select zero or more rule instances for execution;
3. execute the selected rule instances, modifying the current state;

Step (1), matching rules against assertions, consumes most of the computation; our paper presents a new matching algorithm for this task.

Matching a rule against all assertions entails considering the cartesian product of all possible tuples of assertions. Since this product formation is exponential in the number of rule conditions, completely recomputing the match each cycle can be very costly. Therefore, FC/PSs often employ incremental matchers [3, 21] to exploit product computations from prior cycles. To facilitate this incremental computation, match is commonly restricted [4, 11] to conjunctive match, which requires all constraints on a candidate assertion-tuple to be satisfied.

The most common approach to conjunctive match is assertion-space (called "tuple-space" in [21], and "k-space" in [24]) match. For a rule containing n conditions, this approach constructs n-tuples of assertions, where the kth tuple element is an assertion corresponding to the kth rule condition. The RETE [3] family of algorithms, reviewed in the next Section, are incremental conjunctive matchers which operate in assertion-space.

A dual approach, using binding-space, was introduced by us in [21]. Here, m-tuples of bindings are examined for assertion support, where the kth tuple element is a legal binding value for the kth variable. MatchBox [21] is an incremental conjunctive matcher working in binding-space. MatchBox, described below in Section 3, was originally designed for massively parallel architectures.

In this paper, we:

- Introduce Linearized MatchBox, a new binding-space algorithm for incremental conjunctive match suitable for serial processors.

* This work was supported in part by grant R29 LM 04707 from the National Library of Medicine, and by the Pittsburgh NMR Institute.
- Demonstrate the advantage of Linearized MatchBox over MatchBox on serial computers.
- Provide clear criteria for choosing between assertion-space and binding-space matchers in specific situations.
- Describe our implementations.
- Suggest why binding-space matchers may significantly speed up PRODIGY [11], a planning/learning problem solving architecture.
- Give useful extensions to Linearized MatchBox.

We focus on reducing cartesian product construction (i.e., "β-joins") for individual rules. Our conjunctive match semantics includes negated conditions. In the presentation, we restrict the constraints on assertion arguments to equality tests.

2. Assertion-Space Match: RETE

For clarity of exposition, we use these conjunctive rule conditions as a running example.

\[ 1 \text{ (f x * }); 2 \text{ (g x y); 3 \text{ (h y *)}. \]

The three conditions correspond to the requirement for three assertions, having predicates f, g, and h. The left-hand side (LHS) match uses variables x and y to join conditions. X, for example, is an equality constraint between the first condition's first argument, and the second condition's first argument. The arguments labelled "*" are unconstrained. One instance of this rule is the 3-tuple of assertions:

\[ <(f \ a \ c), (g \ a \ b), (h \ b \ d)> \]

corresponding to the 2-tuple of bindings \( x, y \leftarrow \langle a, b \rangle \). The conjunctive match constraints [15] are drawn in Figure 1A.

![Figure 1](image)

**Figure 1.** A. Bipartite graph depicting the constraints between the conditions f, g, and h, and the variables x and y. B. The similarly shaded x and y variables are grouped into a single test node.

The conjunctive match of our example rule against a set of assertions forms the set of valid assertion-tuples

\[ A = \{ a=<(f \ldots), (g \ldots), (h \ldots) | a f (1) = a g (1) \land a g (2) = a h (1) \} \],

where \( a(j) \) denotes the \( j \)-th argument of the \( i \)-th element of assertion-tuple \( a \). The computation of \( A \) may be implemented using a single product node, receiving input from each of its three assertion leaf nodes, as shown in Figure 1B. The valid assertion-tuples in \( A \) are formed by constructing all possible 3-tuples, and then testing them for constraint satisfaction.

Cartesian products can be incrementally constructed using finite differencing [13]. With finite differencing, changing one tuple \( t \) at some node \( N \) affects only those successor tuples that depend on \( t \); successor tuples containing a tuple from \( N \) other than \( t \) are not affected. The finite differencing propagates this dependency from a leaf node [18], recursively inserting or deleting tuples at succeeding nodes (if any) in a network of nodes.

Instead of forming complete n-tuples, tuples may be constructed by pairwise augmentation. A linear ordering on the conditions is used for a linearly recursive divide-and-conquer that enables intermediate filtering for potentially greater efficiency. As shown in Figure 2, variables are grouped into sets \( V_k = \{ \text{variables} v \mid v \text{ affects condition k, but no condition} \ k', \ k' > k \} \). When this linear recursion is partially evaluated [6] into a network, it is transformed [14, 19] into the RETE match algorithm [3]. At test node \( k \), only the variables in \( V_k \) are tested.

![Figure 2](image)

**Figure 2.** A. Linearly ordering the conditions, which partitions the variable set, as shaded. B. Variables can be arranged by the linearly ordered conditions they affect, producing the linear RETE network.

By incremental insertion and deletion, RETE maintains the assertion-space \( A \). Every k-tuple in \( A \) is valid up through the first k assertions, and, inductively, all of its j-tuple prefixes, \( j < k \), are also present. The assertion-space is therefore a network of valid partial k-tuples [17], with each k-tuple having support in smaller j-tuples, and, eventually, the assertions themselves.

Conjunctive match can also test for negated conditions. If a negated condition has an assertion that matches an k-tuple, the assertion blocks that k-tuple. Thus, only when no blocking assertions are matched, can an k-tuple form a rule instance. Negated conditions are readily incorporated into the incremental RETE framework by allowing blocked k-tuples to actively monitor for an unblocking event. This monitoring permits invalid k-tuples to be present on the fringe of the assertion-space network.

Equality constraints may be enforced by partitioning memory, rather than by performing explicit tests on arguments. This is done using hashed-equality [7, 22], which preindexes assertion-tuples by their binding values. Consider our example rule. At the final cartesian
product node that joins \(<(f \times \ast), (g \times y)\) 2-tuples with \(<(h \\ast y)\) assertions, the variable "y" is tested. Suppose that the memories preceding this node were partitioned by their y values, and that product construction respected this partition. Then, when inserting partial assertion-2-tuple \(<(f \times c), (g \times b)\>, having rule binding \(<y>\leftarrow <b>\), it would only form a product with those assertions such as \(<(h \times d)\> that similarly had binding \(<y>\leftarrow <b>\). This testless match is one case [21] of binding-space match, which we now explore.

3. Binding-Space Match: MatchBox

Binding-space B is the set of all possible m-tuples of binding values for a rule’s m variables. The template for a rule’s binding-space is a single node that specifies all of the conditions needed for a match, having each of the variables as a leaf input. This is shown in Figure 3, illustrating the dual relationship with the assertion-space template: the roles of condition and variable are reversed. The cartesian product of the value sets of each leaf variable constructs B.

![Figure 3. A. Reversing our perspective of the relations in the bipartite graph. Compare with Figure 1A. B. We group the conditions into a single test node. The leaves are variable classes.](image)

MatchBox [21] is an incremental binding-space algorithm for conjunctive match. It has each binding-tuple b in B monitor the insertions and deletions of those assertions that can affect it. An assertion affects all binding-tuples that match the partial assignment of assertion values to condition variables. When at least one assertion of each condition is detected, binding-tuple b is valid. (To satisfy a negated condition, conversely, no assertions must be present.)

Continuing with our example, consider the binding-tuple b = \(<x, y>\leftarrow <a, b>\). Substituting into the rule conditions, relevant assertions must satisfy the conditions 1 (\(f \times a\)); 2 (g a b); 3 (h b *).

Suppose assertions \((f \times a)\) and \((g \times a)\) have been presented. These match conditions 1 and 2, respectively. When assertion \((h \times b)\) is presented, it is broadcast to all matching-tuples having \(<y>\leftarrow <b>\), including binding-tuple b. The assertion matches condition 3, and the MatchBox then signals that all three conditions have support; thus, \(<x, y>\leftarrow <a, b>\) is a valid binding-tuple. Adding the assertion \((f \times d)\) would provide a second match for condition 1, but would not affect b’s validity.

More formally, we introduce \(A_{b,i}\) as the set of assertions \(a_i\) of condition i that can affect the validity of binding-tuple b. Clearly, \(a_i\in\mathcal{A}_i\), the set of assertions satisfying the unary constraints on condition i. The second requirement is that the argument projection of the assertion \(a_i\) equal the variable projection of the binding-tuple b. We now develop a useful notation for this relationship.

Let \(J_i\) be the set of arguments in condition i used in intercondition join testing. Let \(K_i\) similarly be the set of join variables used in condition i. There is an isomorphism \(\phi_i\) between \(J_i\) and \(K_i\) that relates the condition arguments to the variables used in join testing. We then have the two partial value binding spaces for condition i

- \(J_i\rightarrow Values\), is the cartesian product of projected argument values,
- \(K_i\rightarrow Values\), is the cartesian product of projected variable values,

where Values a set of possible binding values. \(\phi_i\) then induces the isomorphism \(\mu_i\) that relates these bindings as

- \(\mu_i(J_i\rightarrow Values)\rightarrow (K_i\rightarrow Values)\).

We develop a convenient notation for these projections. If \(a_i\) is a WM assertion of condition i, then we write

- \(\alpha_i(a_i)\) for the partial argument binding extracted by argument indices \(J_i\).

And, if b is a variable binding, then we write

- \(\beta_i(b)\) for the partial variable binding extracted by variables \(K_i\).

Immediately, then, we have the key relationship:

- \(\mu_i\circ\alpha_i(a_i) = \beta_i(b)\)

exactly when the partial argument bindings of assertion \(a_i\) are identical to the partial variable bindings of binding \(b\).

Therefore, we may define

- \(A_{b,i} = \{ a_i \in \mathcal{A}_i | \mu_i\circ\alpha_i(a_i) = \beta_i(b) \} \).

Since \(\mu_i\) is an isomorphism, we can use \(\alpha_i^{-1}\) to write

- \(A_{b,i} = \{ a_i \in \mathcal{A}_i | \alpha_i(a_i) = \mu_i^{-1}\circ\beta_i(b) \} \).

Now, the inverse function \(\alpha_i^{-1}\) maps a partial binding \(\alpha_i(a_i)\) into the set of all WM assertions \(a_i\in\alpha_i^{-1}\circ\alpha_i(a_i)\). That is, all those \(a_i\) satisfying the partial binding. Therefore, we may write

- \(A_{b,i} = \{ a_i \in \mathcal{A}_i | a_i\in\alpha_i^{-1}\circ\beta_i(b) \} \).

But this is just

- \(A_{b,i} = \mathcal{A}_i \cap \alpha_i^{-1}\circ\beta_i(b) \),

which provides an operational definition of \(A_{b,i}\) from the binding-space view.

The MatchBox Theorem (proved in the Appendix of [16]) holds that the set of assertion-tuples \(A_b\) for binding \(b\) is precisely the cartesian product of each condition axis set \(A_{b,i}\) in b’s MatchBox.
\[ A_{b} = \prod_{i} A_{b_{i}}, \]

where \( I \) is an index set of conditions. That is, the contents of \( b \)’s MatchBox is the product \( A_{b} \). In our example, the MatchBox \( A_{<a,b>} \) above would generate the product
\[ A_{b} = \prod_{i} A_{b_{i}}, \]
\[ = A_{b_{1}} \times A_{b_{2}} \times A_{b_{3}}, \]
\[ = \{ (f \ a \ c), (f \ a \ d) \} \times \{ (g \ a \ b) \} \times \{ (h \ b \ d) \}, \]
or, the two assertion-tuples
\[ = \{ <(f \ a \ c), (g \ a \ b), (h \ b \ d) > \} \text{ and} \]
\[ = (f \ a \ d), (g \ a \ b), (h \ b \ d) > \}]. \]

An assertion-tuple can appear in at most one MatchBox, say \( A_{b} \), since any valid assertion-tuple corresponds to exactly one binding-tuple [16]. This induces a partitioning of assertion-space by individual MatchBoxes \( A_{b} \):
\[ A = \Sigma_{B} A_{b}, \]
where \( \Sigma_{B} \) denotes the disjoint union over binding-space \( B \). This binding-space decomposition will prove an important analytic tool in Section 5.

If the right-hand side (RHS) variables are included in a rule’s binding-space as additional dimensions, it is unnecessary to maintain the actual assertions in the MatchBox. The MatchBox then becomes an equivalence class for all rule instances (i.e., assertion-n-tuples) having identical rule bindings. Instead, a nonnegative number suffices, specifying the number of assertions in each condition set \( A_{b_{i}} \). Each insert increments (and each delete decrements) this counter. A binding is valid exactly when all condition counts are \( \geq 0 \) (and all negated condition counts \( = 0 \)). This reduces space cost.

In fact, MatchBoxes need not locally record the state of each \( A_{b_{i}} \) assertion dimension at all. A global register can instead record the number \( |A_{b_{i}}| \) of partial assertions \( a_{i} \in A_{i} \) with \( \mu_{oc}(a_{i}) = b_{i}(b) \) (e.g., all \( a_{j} \) matching \( f \ a \ast \)) currently in the state. Only when this register changes between 0 and 1 need the insert/delete be broadcast to the monitoring MatchBoxes. Each MatchBox need only maintain a bit vector of binary flags, indicating that \( A_{b_{i}} \) is (0) empty, or (1) nonempty. When a change is detected, the vector is updated and logically equivalenced with a bit vector of negated-condition flags; the resulting vector is all 1’s if and only if the binding is valid. When the average number of assertions in \( A_{b_{i}} \) is much greater than 1, this filtering of broadcasts effectively reduces the time cost.

MatchBox was originally designed for a massively parallel computer (such as the Connection Machine) on which the broadcast operation was free. Each binding-tuple requires little more than a bit vector and selective connection to a global data bus. With such low processing demands, each binding-tuple can be allocated its own modest processor, and incremental match occurs in constant time. On a serial computer, however, broadcasts are no longer free, and may require a huge number of messages. In the next section, we adapt MatchBox for serial processing.

4. The Linearized MatchBox Algorithm

A binding-tuple is valid exactly when it has support for all of its conditions. We define the predicate \( \theta \) on the assertion set \( A_{b_{i}} \) as "\( A_{b_{i}} \) empty", for negated conditions \( i \), and "\( A_{b_{i}} \) not empty", otherwise. Then,
\[ \text{Valid}(b) \Leftrightarrow \forall \text{condition} \ i \in I. \ \theta(A_{b_{i}}). \]

Conditions may be grouped into the sets
\[ C_{k} = \{ c \ \mid \ \text{condition} \ c \ \text{contains: variable} \ k, \ \text{some} \ \text{variable} \ j, j \neq k, \ \text{but no variable} \ l, l-k \}, \text{and} \]
\[ D_{k} = \{ \ c \ \mid \ \text{condition} \ c \ \text{contains variable} \ k \ \text{only} \}, \]
as in Figure 4A. Note that conditions not containing any variables are not considered in binding-space match. Let \( b^{k} = (b_{1}, b_{2}, \ldots, b_{k}) \) be a partial binding-tuple, and \( C^{k} = \cup_{i=k}^{\infty}(C_{i} \cup D_{i}) \) a set of conditions. Inductively, we rewrite
\[ \text{Valid}(b^{k}) \Leftrightarrow \forall c \in C^{k}, \theta(A_{b^{k}}c) \]
\[ \Leftrightarrow \forall c \in C_{k}, \theta(A_{b^{k}}c) \land \forall c \in C^{k-1}, \theta(A_{b^{k}}c) \]
\[ \land \forall c \in D^{k}, \theta(A_{b^{k}}c) \]
\[ \Leftrightarrow \forall c \in C_{k}, \theta(A_{b^{k}}c) \land \text{Valid}(b^{k-1}) \]
\[ \land \text{Valid}(b). \]

Therefore \( \text{Valid} \) can be unravelled into a linear recursion, comprised of a local part that tests for conditions, and an inductive part that builds on valid chains of partial binding-tuples.

This linear recursion can be cast [14] into an incremental network algorithm, called Linearized MatchBox, dual to RETE match. The duality is shown in the contrasting Figures 2 and 4, where the roles of condition and variable are reversed. Further, RETE uses fixed node tests, and varying input data. Linearized MatchBox has varying node tests, and fixed input data. Finite differencing incrementally constructs cartesian products of partial binding-tuples. A binding-tuple \( b^{k} \) is present only when \( \text{Valid}(b^{k-1}) \land \text{Valid}(b^{k}) \), which we define as
\[ \text{Active}(b^{k}) \Leftrightarrow \text{Valid}(b^{k-1}) \land \text{Valid}(b^{k}). \]

Figure 4. A. Linearly ordering the variables, which groups the conditions. The lettering refers to example sets introduced below. B. A (small) linear call-graph. Each condition is placed into its appropriate call-node.
Linearized MatchBox introduces partial MatchBoxes $A_{b'}$, corresponding to partial binding-tuples $b'$. This is the key idea of Linearized MatchBox: reducing the number of broadcasts by sending messages only to those active partial (and complete) binding-tuples $b'$ that find support in the asserted data, rather than to all of binding-space. Finite-differencing assures that only binding-tuples with active support are constructed; inactive ones without support do not exist, hence do not receive update messages. An analysis of Linearized MatchBox versus MatchBox is given in Section 5.

4.1. Local MatchBox Update

Using finite differencing, new partial binding-tuples are continually created. For a newly created partial binding-tuple $b'$, its new MatchBox $A_{b'}$ must compute its current state. The partial MatchBox does this by examining $A_{b'j}$ for each $i$ in its condition subset. A very inefficient approach might be to review all assertions in WM to test $\theta(A_{b'j})$. Instead, we again use a global table of counters to reduce each test to constant time.

In Section 3, we introduced an efficiency mechanism for reducing local memory requirements (and broadcasts). Instead of redundantly maintaining private counters for each $A_{b'j}$ set, we associated with $A_{b'j}$ a public global counter for every partial binding-tuple $\beta_j(b)$. These global counters preserve the necessary WM state, and can be used for local MatchBox update. Such testing is also done when the network is initialized.

Each counter records the cardinality of the set

$$A_{b'j}=\{a_j \in A_j \mid \mu \delta_j(a_j) = \beta_j(b)\}.$$ 

These are updated as WM assertions $a_j \in A_j$ are inserted or deleted, based on the variable binding index value $\mu \delta_j(a_j)$. By our linearization construction, the partial binding-tuple $b'$ belongs to a node whose variables subsume the subset $K_i$; thus, $\beta_j(b') = \beta_j(b)$. Therefore, the projection $\beta_j(b')$ of partial binding-tuple $b'$ can be tested against $A_j$'s global counter, determining the size of

$$A_{b'j}=A_j \cap \alpha_i^{-1} \delta_j^{-1} \beta_j(b').$$

4.2. Global Updates

When an assertion enters WM, if it switches some $A_{b'j}$ between empty and nonempty, it is broadcast to all affected partial binding-tuples $b'$. Which MatchBoxes $A_{b'}$ are affected by an $a_j \in A_j$ presented to WM? Exactly those $b'$ in the set

$$\beta_j^{-1} \mu \delta_j(a_j)$$

that are active. With the parallel algorithm of the last Section, all MatchBox's were active, and this set was a geometric subvolume of the binding-space hypercube $B$. Linearized MatchBox, though, has a less symmetric topology: it is a bottom-up network of partial binding-tuples.

The purpose of Linearized MatchBox's more intricate partial binding-tuple network is precisely to reduce the broadcast expense. The only partial MatchBoxes $b'$ affected are those in the set

$$B' = \{b' \in \beta_j^{-1} \mu \delta_j(a_j) \mid Active(b')\},$$

which is a subset of $\beta_j^{-1} \mu \delta_j(a_j)$. If few $b'$ are active, then the broadcasts are effectively reduced. To complete this method, we need an efficient mechanism for determining $B'$.

When a binding-tuple $b'$ (and its partial MatchBox) is created from the partial variable node $X$, it knows the $\beta_j$ map for every condition $i$ tested at $X$. Then, for each $\beta_j$, $b'$ is registered in a global table indexed by its partial variable binding $\beta_j(b')$. (b' is unregistered on its deletion.) This provides an inverse index into $\beta_j^{-1} \mu \delta_j(b')$, the set of all active partial binding-tuples $b''$ having the same variable projection under $\beta_j$ as $b'$.

Subsequently, an assertion $a_j \in A_j$ may enter WM. $a_j$ has the partial variable binding-tuple $\mu \delta_j(a_j)$. But, $\beta_j^{-1} \mu \delta_j(a_j) = \beta_j^{-1} \lambda \delta_j(b')$, for every $b'$ in the inverse index table with binding projection $\beta_j(b')$. That is, by examining the assertion's $\mu \delta_j(a_j)$, we can immediately know the partial binding-tuples $b'$ satisfying:

1. $b' \in \beta_j^{-1} \mu \delta_j(a_j)$, and
2. $Active(b').$

These are precisely the $b' \in B'$ requiring an update to MatchBox $A_{b'}$ for the $i^{th}$ condition.

Once the partial MatchBoxes of $B'$ are known, they are locally updated, as above. If an $A_{b'}$ then changes its state between Valid and Active, it must propagate this change in the MatchBox network, creating or destroying MatchBox nodes. To coordinate this activity, a bottom-up topological traversal [18] is needed for correct, efficient update.

Therefore, the $B'$ broadcast enqueues the initial local updates. There may be multiple MatchBox changes scattered throughout a rule's MatchBox tree. As updates are performed, they may enqueue additional, interleaved events. These perform their finite differencing operations, and may also propagate, level-by-level. After the topological traversal quiesces, the update is completed and the MatchBoxes are all in their correct state.

Note that this mechanism may also be used to dynamically add new variable binding values, which extends the original MatchBox algorithm [21]. This can be done by using the projection $\mu \delta_j(a_j)$ to extract variable binding values from the assertions as they are
presented to WM, and install them as leaf nodes. More sophisticated arrangements, such as assuring that a binding value appears in Values(x) only when it has appeared in all of x's corresponding argument sets, are also possible.

4.3. An Example Trace

Continuing our example, the Linearized MatchBox is sketched in Figure 5A. Since the match variables are x and y, these form the leaf nodes. Condition 1's "(f x *)" of only affects variable x, and is therefore tested at the X variable node. Similarly, condition 3 "(h y *)" is tested at the Y node. Since condition 2 "(g x y)" tests both X and Y, it is tested at the XxY product node.

Prior to adding any assertions to the WM state, the known possible binding-tuples for X and Y are actively monitoring for support, as in Figure 5B. In Figure 5C, when 1 (f a b) is asserted, since \( \mu_{match}(1.2) = (\langle x \rangle, \langle a \rangle) \), it sends a message to the partial binding-tuple \( \langle x \rangle, \langle a \rangle \). This message supports condition 1's partial assertion "(f a *)", since the MatchBox for \( \langle x \rangle, \langle a \rangle \) does not examine the second argument of f, and therefore increments the global counter \( I_{XxY} \) to 1. Only one condition is needed at this MatchBox, and so the partial binding-tuple \( \langle x \rangle, \langle a \rangle \) is validated. However, there are not yet any valid y binding-tuples to form products with.

In Figure 5D, 2 (g a b) is asserted, and \( \mu_{match}(1.2) = (\langle x, y \rangle, \langle a, b \rangle) \). However, no MatchBox for \( \langle x, y \rangle, \langle a, b \rangle \) exists (yet) to receive the message. Nonetheless, the assertion's change to \( A_{X\times Y} \) is indirectly recorded by incrementing condition A2's global count \( I_{X\times Y} \) to 1.

When 3 (h b d) is asserted in Figure 5E, since \( \mu_{match}(1.2) = (\langle y \rangle, \langle b \rangle) \), it validates the partial binding-tuple \( \langle y \rangle, \langle b \rangle \). Then, \( \langle y \rangle, \langle b \rangle \) forms a product with the valid binding-tuple \( \langle x \rangle, \langle a \rangle \), producing the new binding-tuple \( \langle x, y \rangle, \langle a, b \rangle \). The new MatchBox checks the global count \( I_{X\times Y} \), which is nonzero, thus verifying that at least one partially matching assertion for condition 2 has been observed. The binding-tuple network therefore recursively propagates, and ultimately completes a rule instance with the full binding \( \langle x, y \rangle, \langle a, b \rangle \).

5. Analysis of the Algorithms

In this Section, we show

* why Linearized MatchBox is better for serial computers than MatchBox, and
* when to use binding-space match instead of assertion-space match.

5.1. Linearized MatchBox vs. MatchBox

Although MatchBox is designed for a massively parallel architecture, it does find use on serial computers. Linearized MatchBox, which is tailored for serial computation, should show some improvement over MatchBox. Here, we provide a cost comparison for the serial model.

Linearized MatchBox constructs just those MatchBoxes \( A^k \) whose partial binding-tuple \( b^k \) has inductively valid support. These \( b^k \) are our active binding-tuples, defined above as

\[ Active(b^k) = Valid(b^{k-1}) \land Valid(b_k) \]

Using the definition of \( \theta \) from above, since

\[ Valid(b^k) \iff \forall c \in C_k. \theta(A^k_c) \land Valid(b^{k-1}) \land Valid(b_k) \]

whenever there exist partial binding-tuples \( b^{k-1} \) and \( b_k \).

Figure 5. Circled binding-tuples have valid MatchBoxes. A. The Linearized MatchBox class network. B. Initial binding-tuples. C. Add (f a b), validating partial binding-tuple \( \langle x \rangle, \langle a \rangle \). D. Add (g a b), affecting no binding-tuple, but recording its partial assertion(s). E. Add (h b d), validating \( \langle y \rangle, \langle b \rangle \), and recursively constructing more binding-tuples by incremental cartesian product formation.
that are both valid, our bottom-up inductive incremental construction will ensure a MatchBox partial binding-tuple for all active \( b^k \). Therefore, the binding-space tuple network is exactly the set of active partial binding-tuples. Each partial binding-tuple in this network actively monitors the broadcasts of relevant assertions.

On a serial computer, the broadcast to each MatchBox has unit cost. Therefore, the broadcast cost is proportional to the number of monitoring MatchBoxes. For Standard MatchBox, this is the full exponential complement of binding-tuples. For Linearized MatchBox, though, only the active binding-tuples monitor WM. Further, the number of active partial binding-tuples is asymptotically equal to the number of active complete binding-tuples. The efficacy of Linearized MatchBox, relative to MatchBox, can therefore be estimated by the fraction of active complete binding-tuples, relative to all possible complete binding-tuples.

We derive the fraction of active binding-tuples probabilistically as:

\[
\text{Prob}(\text{active}) = \prod_{i=1}^{\text{num conditions}} \text{Prob}(\text{condition } i \text{ has support}),
\]

since we assume the condition assertions to be independent, and this

\[
= \prod_{i=1}^{\text{num conditions}} \text{Prob}(\theta(A_{bi}^i, j)).
\]

We estimate the number of assertions in condition \( i \)'s set \( A_{bi}^j \) by a Poisson distribution [2],

\[
\text{Prob}(k; m) = e^{-m} \left( \frac{m^k}{k!} \right).
\]

We do this because, in our model, we assume

- that \( I_{Ab^i, j} \), the number of assertion events \( a \) of condition \( i \) that have \( \mu_{a} \alpha_{a} = \beta_i(b^j) \), is small, and
- that these assertions are independently occurring events, hence the statistics \( m=I_{Ab^i, j} \) are independent random variables.

Using the definition of \( \theta(A_{bi}^i, j) \), the Poisson distribution determines the probability

\[
\text{Prob}(\theta(A_{bi}^i, j)) = (1-e^{-m}) \text{ for observing at least one asserted positive condition, and}
\]

\[
e^{-m} \text{ for observing no asserted negated conditions.}
\]

Therefore, decomposing the condition set \( I \) into positive conditions \( I^p \) and negated conditions \( I^\neg \), we have:

\[
\text{Prob}(\text{active}) = \prod_{i=1}^{I^p} (1 - \exp(-m(A_{bi}^i, j))) \prod_{i=1}^{I^\neg} \exp(-m(A_{bi}^i, j)).
\]

For rule instances, index set \( I \) is the set of all conditions in the LHS, and \( b^p = b^\neg = b \). Thus, Linearized MatchBox reduces the actively monitoring MatchBox set according to the number of distinct partial assertions. When there are a small number of partial assertions for each positive condition (i.e., small \( I_{Ab^i, j} \)), and a large number for each negated condition, the fraction of active binding-tuples is significantly reduced. In this situation, Linearized MatchBox is very effective. When, conversely, there is greater data flow for positive assertions, and fewer blocking negated assertions, a greater number of MatchBoxes are constructed, and Linearized MatchBox may be less useful.

### 5.2. Match Costs: Binding-Space vs. Assertion-Space

To compare various binding-space and assertion-space matching algorithms, we need estimates of the MatchBox axis set cardinalities \( I_{Ab^i, j} \). We introduce a probabilistic model for assertions to provide these estimates. The idea is to count up the different ways that distinct WM assertions contribute to equivalent matches.

The number of condition \( i \) assertions in the state depends both on (1) the selective unary filtering of assertions that often precedes tuple formation, and (2) the data flow of partial assertions. We model each unary assertion set \( A_i \), corresponding to the rule's \( i \)'th condition, with two parameters:

- \( p_i \) the probability of any given argument assignment for \( A_i \) being present in the state, and
- \( N_{ij} \) the number of possible values for the \( j \)th argument of \( A_i \). When a uniform joint distribution for the value assignments is assumed, \( 1/N_{ij} \) is the probability that a randomly selected assertion has some given value appearing in the \( j \)th argument.

Under these assumptions, the estimated mean size of \( A_i \) is

\[
m(A_i) = p_i \sum j N_{ij}.
\]

We now estimate the cost of cartesian product formation using straightforward counting arguments that exploit our independence and uniformity assumptions. To eliminate the complexities of optimal condition ordering [15, 23], we estimate the cost for single node, rather than linear node, topologies. This also restricts our discussion to equality relations, since nonequality tests occur subsequent to \( n \)-tuple formation.

We derive estimates for hashed-equality RETE, and for MatchBox. The product construction for these algorithms is simpler to estimate in binding-space than in assertion-space. The idea is to compute the MatchBox update cost for each affected binding, and then sum over all these binding-tuples.

We therefore write the expected value as

\[
\mathbb{E}(\Sigma \text{ affected } b \in B | I_{Ab^i})
\]
which, by additivity of expected values,
\[ = \sum_{\text{affected \ } b \in B} E(\Delta_i \ \Delta_b), \]
Since \( E(\Delta_b) \) and \( E(\Delta_i \ \Delta_b) \) are assertion subspaces, under our assumption of independent assertions, these MatchBox sizes are independent of \( b \). Therefore, the sum becomes the product
\[ = I_{\text{affected \ } b \in B} E(\Delta_i \ \Delta_b), \]
which, for an assertion \( a_i \), is just
\[ = (\beta_i^{-1} \sum_{\text{outside}(a_i)} x) \times E(\Delta_i \ \Delta_b). \]
(When a single assertion \( a \) affects multiple rule conditions \( i \), we consider its affect as \( a_i \) separately for each condition \( i \).)

Starting from this binding-space framework, in the Appendix we derive estimates for forming an incremental cartesian product with respect to an assertion of the \( i \)th positive condition. The estimates are:

**Incremental hashed RETE match.** (Single node.)
\[ (\Pi_{\text{classes}} \ \text{EC outside } i \ \text{N}^{\text{EC}}) \times (\Pi_i \ \text{Pi} \ \text{Pi}_{\text{nonvars}} \ j \ \text{Ni}_j), \]

**Incremental binding-space match.** (MatchBox.)
\[ (\Pi_{\text{vars \ outside \ } i} \ \text{N}^{\text{k}}) \times \exp(-p_i \ \Pi_{\text{nonvars \ inside \ } i} \ \text{Ni}_j), \]

Analogous results can be derived that include negated conditions.

Given a rule and a model of the predicate data, the incremental MatchBox cost can be compared with the hashed-equality cost. The more appropriate matching strategy can then be selected for the rule, as follows. Note that the number of variables is generally greater for MatchBox, since RHS variables must be included in the binding-space construction. Accounting for this, the ratio of the update cost of hashed-equality RETE
\[ (\Pi_{\text{vars \ outside \ } i} \ \text{N}^{\text{k}}) \times \Pi_i \ \text{Pi}_{\text{nonvars \ outside \ } i} \ \text{Ni}_j \]
to the update cost of MatchBox
\[ (\Pi_{\text{vars \ outside \ } i} \ \text{N}^{\text{k}}) \times \exp(-p_i \ \Pi_{\text{nonvars \ outside \ } i} \ \text{Ni}_j), \]
is, after eliminating the common factor \((\Pi_{\text{vars \ outside \ } i} \ \text{N}^{\text{k}})
\[ \times \exp(p_i \ \Pi_{\text{nonvars \ inside \ } i} \ \text{Ni}_j). \]
This ratio shows that the binding-space MatchBox algorithm has advantages over hashed-equality RETE when there is redundancy in the assertions. That is, when many WM assertions map to the same \( A_{b;i} \) set, the MatchBox equivalence classes can effectively reduce product computation.

6. Implementations and Utility

We have implemented Linearized MatchBox in CACHE™ [20], a Call-Graphing [14] development environment written in Common LISP for the MACINTOSH/II computer. CACHE™ provides color animations of the algorithm's operation, showing incremental insertion and deletion of binding-tuples (MatchBoxes) as the assertions are entered and withdrawn. Using comparable data structures, we also implemented a hashed-equality matcher that supports negated conditions. This will facilitate the construction and evaluation of mixed rule systems that use the criteria of Section 5 to automatically select appropriate matching strategies.

One candidate is the matcher of the PRODIGY [11] rule system. PRODIGY is a backward chaining production system (BC/PS) planner that learns control rules [12]. These control rules are used in a FC/PS as knowledge that directs the BC/PS search. In the limit, as more search control is learned, matching control rules in the FC/PS becomes PRODIGY's dominant (> 95%) bottleneck computation [1]. As with SOAR [9], the expense of applying knowledge eventually outweighs the benefits of learning [24].

Interestingly, PRODIGY (and SOAR) uses a variable-based matcher restricted to equality tests, and is therefore an easy case for Linearized MatchBox. Unlike SOAR, though, PRODIGY's conjunctive matcher is nonincremental; recently, incremental RETE matchers have been grafted on. Our analysis shows that a combination of Linearized MatchBox and hashed-equality RETE should prove far more effective, and may help overcome PRODIGY's knowledge application bottleneck.

7. Conclusions

We have introduced Linearized MatchBox, a new binding-space algorithm for conjunctive match. It improves on the parallel MatchBox algorithm by reducing broadcast cost, better adapting it to serial computers. We presented cost estimates that help determine the applicability of Linearized MatchBox for particular rule problems. Linearized MatchBox has been implemented, and its utility in overcoming PRODIGY's control knowledge bottleneck was discussed.

Several extensions to Linearized MatchBox are readily implementable:
- Using OPS-5's semantics to have predicate tests return TRUE when binding values are missing. This can be done by adding a wild-card value.
- Dynamically adding new binding values. This is easier with Linearized MatchBox than with the original MatchBox algorithm. New binding values can be checked for and added to leaf variable nodes. The network's finite differing automates recursive MatchBox construction.
- Testing nonequality relations. This can be done at a partial binding-tuple once all the requisite variable bindings are present.
- Reactivating, rather than deleting, inactive binding-tuples. This follows the TMS-like strategy of Scaffolding [17], where no tuples are ever discarded. Inserting new tuples incrementally
builds up the binding-space, and, in the limit, constructs all MatchBoxes. This approach is useful with nonequality tests, since tests on binding-tuples need never be repeated.

The ultimate assessment of Linearized MatchBox will be by the users of production system (PS) applications. If PS implementers can provide Linearized MatchBox to PS programmers, then potential speedups will be passed on to users. This paper presents a new AI tool for the PS implementer, describing the mechanisms and utility of our new algorithm.

Appendix

This Section provides additional detail for the match cost estimates given in Section 5.2. Specifically, we compute the cost of incremental update for both the hashed-equality RETE assertion-space and MatchBox binding-space conjunctive matching algorithms.

What is \( I_{pi}^{-1} \mu_{o, ai}(ai) \)? Condition i's variable set is \( K_i \), hence assertion \( ai \) fixes values for \( x_k \in K_i \) and leaves the other variables in \( K-K_i \) free. Therefore,
\[
\beta_i^{-1} \mu_{o, ai}(ai) = \prod_k W_k,
\]
where, as above, \( W_k \) is the set \( \text{Values}_k \) of variable \( k \), for \( k \in K-K_i \), and a singleton set otherwise. But this is just
\[
= \prod_k |W_k| = \Pi_{k \in K-K_i} N_k = |\text{Vars} k \text{ outside } i| N_k.
\]
Recall that \( \text{Values}_k \) is formed by intersecting all the values in each LHS condition i's argument \( j \), which ensures that \( N_k \leq N_{i,j} \), and may significantly reduce the product size.

For hashed-equality RETE, what is \( E(\Delta_i \text{Ab}) \)? This is computed by taking the assertion-subspace product
\[
\prod_{i \in J} \prod_{j} \mu^{1-1_0}_{o, ai}(ai) \times E(\Delta_i \text{Ab}),
\]
over those conditions partially matching binding-tuple \( b \). For each assertion \( a \in \mu^{1-1_0}_{o, ai}(ai) \), the arguments in \( J_i \) are fixed by their projection \( \mu_{o, ai}(ai) = B_i(b) \). Therefore, the free arguments of condition i are those not in \( J_i \). Thus, the product is
\[
\prod_{i \in J} \left( \prod_{j} \Pi_{\text{nonvars} j N_{i,j}} \right).
\]

The update cost for hashed-equality RETE when asserting condition i is then
\[
\beta_i^{-1} \mu_{o, ai}(ai) \times E(\Delta_i \text{Ab}) = \left( \prod_{\text{vars } k \text{ outside } i} N_k \right) \times \Pi_{\text{nonvars} j N_{i,j}}.
\]

References


M. W. Perlin, "Call-Graph Caching: Transforming Programs into Networks," in Proc. of the Eleventh Int. Joint Conf. on Artificial Intelligence, Detroit, Michigan, Morgan Kaufmann, August, 1989, pp. 122-128.


