The Match Box Algorithm for Parallel Production System Match

Mark W. Perlin

May 29, 1989
CMU-CS-89-163

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

We introduce Match Box, a new increment... matching algorithm for determining the tuple instantiations of forward-chaining production rules. Match Box is rooted in the mathematical interconnections between tuple and binding spaces, a framework also applicable to other pattern matching algorithms. The idea is to precompute a rule's binding space, and then have each binding independently monitor working memory for the incremental formation of tuple instantiations. A key feature of Match Box is that on a massively parallel architecture, it can perform a rule's computationally intensive incremental join testing in constant time. It also finds application on conventional serial processors.

This work was supported in part by grant R29 LM 04707 from the National Library of Medicine.

Introduction

Production systems (Waterman & Hayes-Roth, 1978) are a problem solving architecture based on the matching of rule patterns against working memory objects. On each iterative cycle, a production system recognizes which rules are relevant, selects one or more for execution, and then acts by executing the selected rules. Recognition entails matching all the rules against the entire object set D, an NP-hard problem which dominates the computational cost of the system. (A tuple of objects matches, or instantiates, a rule's pattern when the tuple satisfies all the pattern's conditions; such a tuple is termed an instantiation of the rule.)

There are conjunctive matching algorithms (Forgy, 1982; Miranker, 1987) that have improved average case behavior. They exploit the empirical observations that (1) the rule set is virtually constant and that (2) there are few changes to the object set each cycle. This situation allows intercycle caching of the subtuples (or partial instantiations) that have satisfied some of the rules' conditions, thereby avoiding redundant predicate recomputation. For each rule, this amounts to a data driven search (Tambe & Newell, 1988) of the rule's tuple space T, i.e., its set of all candidate tuples.

A different matching strategy for a rule is to explore B, the space of all possible variable bindings for a rule. In this paper, we shall motivate, formulate, and present Match Box, a new algorithm for production system matching based on incremental search of the binding space. On certain parallel architectures, Match Box can use a processor-for-time tradeoff to reduce incremental matching to constant time. Under some conditions, it can also be more efficient on a serial processor than standard RETE-like tuple space search. Unlike the limitations inherent in token-level parallelism (Gupta, 1986), Match Box demonstrates effective fine-grained parallelism at the level of match operations.

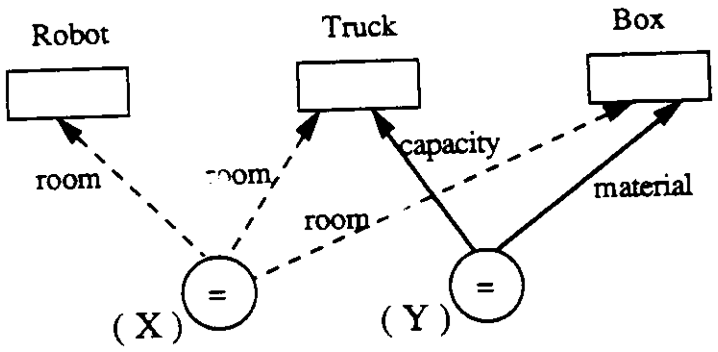
Example

We begin with an extended example to motivate the Match Box. Starting from an informal, English language description of a rule, we show how its binding space B is constructed and then used for determining instantiations.

Assume that in a simplistic two dimensional world we have robots, trucks, and boxes. Each object has a location, which may be either Room left or Room right. Suppose that boxes are made of only three materials: paper, lead, and glass. Further suppose that there are only trucks corresponding to the first two materials, light-weight for paper, and heavy-weight for lead, but no sort of truck for the fragile glass. With this background, we could then develop rules for the robots to move boxes using the trucks. The pattern of one such rule might be:

> If there is a Robot in some room,
> and there is a Truck of some weight class in the same room,
> and there is a Box of the same weight class in that room, ...

These conditions may be graphically depicted by the constraints (Perlin, 1989):



To program these three conditions in the (approximate) syntax of the OPS-5 production system language, one could write:

```
(p robot-moves-box
  (robot ^room <X>)
  (truck ^room <X> ^capacity <Y>)
  (box   ^room <X> ^material <Y>)
 --> ...)
```

Regardless of the specification method, we end up with two variables, the common room X, and the material Y. Note that X is restricted to the values left and right. Similarly, Y must take values from the set {paper, lead}, which is the intersection of the value set for box materials {paper, lead, glass}, and the truck capacity value set {paper, lead}. The set of possible bindings for our rule is then the set product Values(X) x Values(Y) = {(left, paper), (left, lead), (right, paper), (right, lead)}. This is the binding space B, formed from the values of XxY, and shown below.

	X variable		
Left	Right		
X = Left, Y = Paper	X = Right, Y = Paper	Paper	Y variable
X = Left, Y = Lead	X = Right, Y = Lead	Lead	

The Binding Space B

When does a binding b in B have support? That is, how can we determine whether there exists one or more instantiations (i.e., a 3-tuple containing a robot, a truck, and a box satisfying the rule's constraints) that actually has binding b? One approach is to attach a subspace of tuple space T, the set product Robots x Trucks x Boxes, to each binding b in B, and observe when this subspace assumes full dimensionality (n=3, for the three objects). We illustrate this by tracing through a sample sequence of changes to working memory D.

Before we begin adding and deleting objects to working memory, the rule's binding space B is initialized to an empty state, where each binding's tuple subspace is initialized to the 0-dimensional set product <{},{},{}>.

X = Left	X = Right	
< {},{},{}>	< {},{},{}>	Y = Paper
< {},{},{}>	< {},{},{}>	Y = Lead

We first assert that Robot Fred is located outside the rooms. This has no effect on B, since only locations Left or Right can lead to a binding of a robot with the other two objects.

We next assert that Robot Fred moves inside to the Left Room. This information can influence all bindings b in B such that the value of X is bound to "Left". These bindings form a subspace B' of B, with the X coordinate set to "Left", and the Y coordinate free. We therefore broadcast the occurrence of Robot Fred to all the bindings in B' = {(X,Y) | X = "Left"}. When Fred arrives at some binding point in B', he represents a Robot object contributing to the first coordinate of T', the local subspace of tuple space T = Robots x Trucks x Boxes. Fred is therefore deposited into the first slot of T', augmenting b's local tuple subspace T', and increasing its dimensionality from 0 (empty) to 1. This has the effect of updating the affected 3-tuple instantiations. The result of this broadcast and updating is shown below.

X = Left	X = Right	
< {Fred},{ },{ }>	< { },{ },{ }>	Y = Paper
< {Fred},{ },{ }>	< { },{ },{ }>	Y = Lead

Suppose that we now assert that there is a light-weight Paper-carrying Truck in Room Left, called Truck1. This affects the subspace B' of binding space, where the first coordinate is constrained by X="Left", and the second coordinate Y must equal "Paper". Therefore, B' consists of the single point b = ("Left", "Paper"). When this Truck object arrives at b, it affects the second coordinate of tuple space subspace T' ⊆ Robots x Trucks x Boxes. The result of Truck1's broadcast is therefore that the single binding space point b has its second coordinate (Trucks) updated with the new Truck1 object, as shown below.

X = Left	X = Right	
< {Fred},{Truck1},{ }>	< { }. .{ }>	Y = Paper
< {Fred},{ },{ }>	< { },{ },{ }>	Y = Lead

Similarly, asserting the presence of a heavy-weight Lead-carrying Truck in the Right Room would update the second coordinate of local T' tuple subspace at the point ("Right", "Lead") in binding space. The current state is shown below. Since no local tuple subspace T' has full dimensionality, thus far no 3-tuple instantiations have been formed.

X = Left	X = Right	
< {Fred},{Truck1},{ }>	< { },{ },{ }>	Y = Paper
< {Fred},{ },{ }>	< { },{Truck2},{ }>	Y = Lead

Suppose we now assert that there are two Paper Boxes in the Left Room, a Red one and a Blue one. These two objects affect the points {(X,Y) in B | X="Left", Y="Paper"}, i.e., the single point b = ("Left", "Paper"). Adding Red and Blue to the third coordinate the local tuple subspace T' of Robots x Trucks x Boxes, results in the state shown below. The dimensionality of b's tuple subspace T' is now the requisite 3, and the two points of T' (<Fred, Truck-1, Red> and <Fred, Truck-1, Blue>) are the 3-tuple instantiations of binding b. Since no other point in binding space has a T' with full dimensionality, these are the only two instantiations of the rule.

X = Left	X = Right	
<{Fred},{Truck1},{R,B}>	< {},{},{}>	Y = Paper
< {Fred},{},{}>	< {},{Truck2},{}>	Y = Lead

Should Robot Fred now move to the Right Room without taking any boxes, we would have the state shown below. Here, no local T' tuple subspace has full dimensionality 3 at any point in binding space; hence the rule has no instantiations.

X = Left	X = Right	
<{},{Truck1},{R,B}>	< {Fred},{},{}>	Y = Paper
< {},{},{}>	< {Fred},{Truck2},{}>	Y = Lead

Mathematics

Conjunctive matchers impose two kinds of conditions on the objects in candidate tuples: predicates on a single object (constant tests and comparison tests), and predicates on more than one object (join tests). The former (called alpha tests in RETE) serve primarily as a filter for the computationally more expensive join (or, beta) tests, and we introduce the map α to effect the alpha filtering. (For detailed definitions and proofs, see the Appendix "Mathematical Development".)

An instantiation of a rule R is an n-tuple of data objects in working memory D satisfying the rule's n conditions (and possibly additional absence tests). Let $D_k = \alpha_k(D)$ be the subset of objects in D satisfying the single object predicates of the k^{th} condition, $k \in K = \{1,...,n\}$. The k^{th} object in any instantiation must belong to the rule's D_k, since R's k^{th} condition includes both the single and multiple object predicates. Therefore, any instantiation of R belongs to the set $\prod_K(D_k)$; this set product is T, the <u>tuple space</u> of the rule R.

When R has join tests, a set of variables $\{x_j \mid j \in J\}$ is implicitly introduced into the match. Let μ be the mapping from the accessors of a tuple's object attributes onto J. The <u>binding</u> of an instantiation is the assignment of each variable to its corresponding value, extracted by applying μ to the n-tuple (comprised of objects with attributed values). This correspondence is denoted by the function f: T -> B, which assigns a tuple its binding. Let V be the set of possible values. Since there may be restrictions on the legal values for each variable x_j, we introduce $V_j = \{v \in V \mid v \text{ is a permissible value for variable } x_j \text{ in rule } R\}$. Then any binding for rule R must be an element in $\prod_J V_j$. This set product is B, the <u>binding space</u> of the rule R.

Consider the commutative diagram

$$\begin{tikzcd}
& P(D) \arrow[dl, "\alpha"'] \arrow[dr, "\sigma"] & \\
T \arrow[rr, "f"] \arrow[dr, "\beta"'] & & B \arrow[dl, "\tau"] \\
& 2 &
\end{tikzcd}$$

where 2 is the two point set {TRUE, FALSE}. α maps data objects from the power set P(D) into tuples in T, performing all requisite unary predicate filtering. In standard matching, the map β then tests candidate tuples by using f to map tuples of T into bindings in B, and then having τ perform all remaining (non-unary) predicates between objects. The goal of any match algorithm is to search T for the instantiation subset $I = \beta^{-1}(\text{TRUE})$.

The function f can be modified to test all the equality relations by constraint checking. (τ, then, tests only the nonequality relations.) The composition $\sigma = f \circ \alpha$ can then be introduced. σ routes the data objects into those bindings they can possibly influence by tuple formation. Each component map $\sigma_k = f_k \circ \alpha_k$ routes objects passing α_k (the k^{th} tuple coordinate filter) into the k^{th} dimension of affected match boxes.

An alternative description of β^{-1} is $f^{-1} \circ \tau^{-1}$. Therefore the instantiation set $I = \beta^{-1}(\mathrm{TRUE}) = f^{-1} \circ \tau^{-1}(\mathrm{TRUE})$. If the set of valid bindings $B_0 = \tau^{-1}(\mathrm{TRUE})$ were computed in advance, then the set I could be found by computing $f^{-1}(B_0)$.

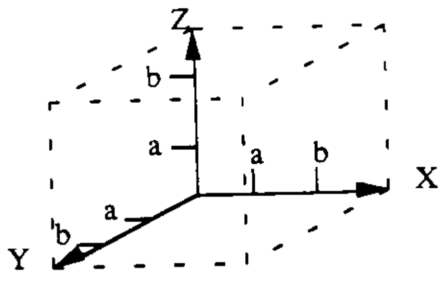
The key mathematical fact is that for any binding $b \in B$, $f^{-1}(b)$ is the cartesian product $\prod_K f_k^{-1}(b)$ of the inverse functions $\{f_k^{-1}: B \rightarrow D_k\}$ for individual tuple components. We call this inverse image set product the match box of b. Thus, if σ routes objects into the valid binding subset B_0, and each $b \in B_0$ monitors its match box for full dimensionality, we can then observe the formation of instantiating tuples. Therefore, the instantiations can be formed without further testing or other expensive computation: all the necessary matching has already been done by precomputation of the (valid) binding set B_0.

The Match Box Algorithm

The match box algorithm explicitly constructs the binding space B. Each axis of B is formed from the (prespecified and fixed[1]) values of a match variable. Working memory objects are routed from conditions to those coordinates in B that satisfy the partial binding constraints of the object and condition. Broadcasts to just the valid bindings in B can be effected by suitably restricting the broadcast geometry, even with general relations between the variables. For simpler exposition, however, we consider only equality relations, which leads to the limiting case with each variable encoding one equality constraint, independently of the other variables. (That is, the function f does all the consistency checking work, and τ trivially maps into TRUE.)

The match box algorithm first constructs a binding space B for each rule. Each axis of B uniquely corresponds to a variable, and is formed from its variable's set of possible values. With equality, this value set is the intersection of the value sets of each attribute constrained by the equality. For example, suppose that attribute FOO of one object, with possible values {a, b, c}, and attribute BAR of another object, with values {a, b, d}, are constrained by variable x; then x's axis is the vector <a, b>. If there are three such variables x, y, z, each with value axes <a, b>, then B is the three dimensional binding space shown below.

[1] The binding space axes can be dynamically augmented during incremental match. However, as with RETE's Build operation, such actions may be expensive, since they entail iterating over the current working memory set.



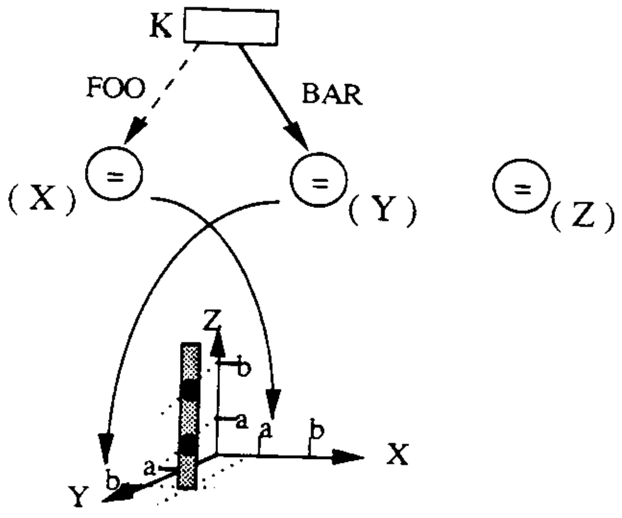
Every cycle, a new object d is incrementally added (or deleted) to the match boxes in B.

1. The object is broadcast to the bindings $\sigma(d)$ in B.
2. At each binding b in $\sigma(d)$,
   - 2a- Augment (subtract) the object set of each affected tuple coordinate with d.
   - 2b- If the binding's match box $f^{-1}(b)$ (i.e., tuple subspace) achieves full dimensionality, signal the match.
   - 2c- With full dimensionality, b's instantiations are exactly the tuples in the match box.

We further detail these two steps.

In step 1, the broadcast $\sigma(d)$ is done by executing $\sigma_k(d)$ for every k in the set of tuple indices K. Recall, $\sigma_k = f_k \circ \alpha_k$. α_k performs the unary predicates on d, either returning the object d or terminating with $\perp$. Let K' be that subset of K for which α_k succeeds. For every k in K', $f_k(d)$ then broadcasts d into the full m-dimensional subspace of B whose coordinates are consistent with the values of d's kth coordinate's attributes. (The broadcast need only be done when each attributed value of d is actually a member of its corresponding axes' value set.)

For example, suppose (as above) that a rule has three variables, x, y, and z, each taking values from the set {a, b}, and that condition k has attributes tested by variables x and y. If an object d successfully passes α_k, then μ_k will map d to variables x and y. During this mapping, the attributed values of d can be extracted and bound to their variables. So if FOO(d)=a, and BAZ(d)=b, with x and y constraining FOO and BAZ, respectively, then we reach the variables with x bound to a, y bound to b, and z unconstrained. Since $f_k(d) = \{a\} \times \{b\} \times V_z$, this is the subspace of B to which object d is broadcast. The mapping and the broadcast subspace are shown below.



In step 2, object d arrives at some binding b in B, and updates the match box. The update entails adding (deleting) d from the kth coordinate set, and then performing some bookkeeping operations. For example, a counter recording the size of the kth coordinate set may be incremented (decremented). If the counter changes from (to) 0, then a counter recording the dimensionality of the match box may concommitantly be incremented (decremented). When the dimensionality counter equals the full n = dim(T), a flag can then signal the presence of instantiations. (To accommodate absence tests, the counters and flags can be reorganized to signal empty, instead of nonempty, sets.)

There are other efficiencies in our algorithm. For example, if a tuple coordinate k is not constrained by any relation, then there is no point in broadcasting objects from k into B, since they will always go to every valid binding. In this case, a separate match box for the unconstrained tuple coordinates can be maintained. This null kernel can be conjoined with any (smaller dimensionality) flagged match box in B to form complete instantiations, thereby reducing B's broadcasts and memory requirements.

Applications

A rule's matching computation usually entails testing the relations of all candidate tuples. While incremental state-saving algorithms such as RETE can exploit the fixed rule set and slow variation in WM to improve average case behavior, worst case behavior requires formation and testing of all tuples. This corresponds to the objective of match: exploration of tuple space. Thus the complexity is proportional to the size of tuple space $|T|$, or $|D^n|$; i.e., the standard match procedure is exponential in the tuple size.

The match box algorithm, on the other hand, does not explore T. Rather, tuples are constructed indirectly, by specifying a cross product of object sets for each binding having support (i.e., flagged as having tuples) in its match box. The worst case broadcast must communicate with $|B|$ bindings. At each binding in B, at most n (the tuple size) memory updates are done, each update taking constant time. (Of course, average case behavior costs much less than this.) These operations are done for each object inserted or deleted from WM, so the total cost is roughly $n|D||B|$. That is, the incremental complexity is proportional to $|B|$, or $|V_1| x |V_2| x ... x |V_m|$, exponential in the size of the variable's value sets.

On a serial processor, we have a clear criterion for using the Match Box method over standard matching: when a rule's binding values are known in advance, with a product set B much smaller than the tuple space T. This situation can occur, for example, when a rule has no optimally filtering join ordering, with large working memories. Maintaining B and its inverse partial tuples also necessitates a large space-for-time tradeoff.

This tradeoff is translated into a processor-for-time tradeoff in a massively parallel architecture. Consider the situation where for every rule, each binding b in B is allocated its own processor. This processor maintains the $f^{-1}(b)$ tuple subspace, and all associated bookkeeping (counters and flags) information. The number of such processors is the sum of $|B_R|$, for every R in the rule set. Since, for any given R, n is fixed and the cost of incrementally updating B_R is distributed across $|B_R|$ noncommunicating processors, the incremental join match is accomplished in constant time.

We are examining possible designs for a hardware realization of Match Box. In some production systems (e.g., OPS-2, SOAR), the set semantics restricts actions to refer to variable bindings, and not directly to working memory objects. Therefore counters and flags alone suffice: no memory for objects is needed for the match. The requisite hardware thus amounts to an associative memory for self-recognition during broadcasts, coupled with a dozen or so registers and gates. The speed and VLSI area requirements should therefore be within an order of magnitude of current memory circuits.

(Historically, unary predicate testing and conflict resolution cost little in the overall rule matching scheme. With constant time massively parallel join match, these anxillary actions increase in proportionate cost. Since these actions are not inherently expensive, they can be easily parallelized into less costly, or even constant time, operations.)

The following preliminary simulation suggests the efficiency of our algorithm. On a simple rule set for the toy problem of the farmer crossing the river with his fox, goat, and cabbage, we recorded the join tests performed for classical RETE and the Match Box. This problem can be encoded with three rules, only one of which has a join test; this rule's binding space has just the two points {LEFT, RIGHT}. Complete problem solution over the 8 cycles required 21 join tests for RETE, but only 12 routing operations Match Box, a 50% reduction. In the parallel architecture, the cost would just be the number of broadcasts (i.e., the number of working memory changes). We are currently profiling more practical domains.

Related Work

Matching based on bindings, rather than on tuple testing, has been used as part of standard matching algorithms. For example, hashed RETE (Scales, 1986) performs efficient equality testing at beta join nodes in the RETE network. Restated in our framework, RETE partitions tuple space testing into n successive stages, each joining $T_{k-1} x D_k$ into T_k, where T_k is $D_1 x D_2 x \ldots x D_k$, $k \leq n$. This induces a restricted family $\{f_k\}$, which perform equality tests for those variables J'_k that appear in tuple coordinate k and at least one additional coordinate $k' < k$. Each f_k maps $T_{k-1} x D_k$ into B_k, where B_k is a restricted binding space formed from the variables of J'_k. The b's in B_k are created dynamically from partial matches, and are not precomputed. Each binding b, in essence, maintains a two dimensional match box comprised of the tuples in T_{k-1} and the objects in D_k corresponding to the partial binding b. When $f_{k-1}(b)$ achieves full dimensionality (i.e., an incoming tuple detects a nonempty object set, or vice versa), the match is signaled.

Copy and constrain (Pasik, 1987) is another standard match technique employing binding values. Here, when a rule's variable has a finite value set, the rule is split into a corresponding set of rules, each having a fixed value in place of the variable. This is the same as analyzing the m dimensional binding space B, finding a suitable variable x_j, and splitting B into the family $\{B_v \mid v \text{ a value for variable } x_j\}$, where each B_v is an m-1 dimensional subspace of B. Each new B_v corresponds to the binding space of a new rule.

Interestingly, this is a binding space transformation done to improve the efficiency of standard tuple space match.

Standard constraint propagation methods use relations to prune constant sets and arrive at consistent solutions. This can be exploited in rule matching, as in (Bibel, 1988), where the possible constant values of relations help in efficiently determining instantiations. In fact, with our binding space formalism, many other constraint propagation techniques may help improve match.

Conclusion

Production system matching is a well-known computationally expensive problem that requires exhaustive search of the tuples that can be formed from available data, which satisfy a set of relations. The resulting subset of tuple space (or k-space) T is the set of instantiations. Standard matching amounts to efficient generation and testing of (partial) tuples to effectively explore this space.

However, this problem has a dual character: it is the bindings of tuples that actually undergo testing, not the tuples themselves. If the data objects can be routed to the valid bindings they can affect, then tuples need not be explicitly constructed. Instead, the partial testing of tuples can be replaced by the partial constraint satisfaction of objects, with respect to their affected bindings.

It is often the case that the admissible values of variables are known in advance. The Match Box algorithm exploits this situation by explicitly constructing a cross product space B of possible variable bindings. (When the relations are restricted to equality constraints, B is identically the space of valid bindings.) Broadcasts through subsets (with equality alone, subspaces) of B correctly route objects to affected bindings. By proper storage of these data objects, each binding can track the tuple instantiations formable from the data that would correspond to the binding. This is accomplished without actually incurring the cost of tuple construction. More importantly, the precomputation of B eliminates any need for actual testing of objects or tuples.

If |B| is very large, Match Box is not practical on a conventional serial processor. However, with smaller |B|, Match Box can be useful for many rules. On a massively parallel architecture, with one processor allocated to each binding, the processor for time tradeoff allows matching to be executed incrementally in constant time. Further empirical results are needed to assess the utility of Match Box for the NP-hard production system match problem.

Acknowledgements

Jean-Marc Debaud of the Philosophy Department at Carnegie Mellon University implemented the algorithms, ran the simulations, and was an excellent sounding board. Jaime Carbonell provided a number of useful insights into the method. The ideas were improved by discussions with Milind Tambe and Brian Milnes.

References

Bibel, W. (1988). Constraint Satisfaction from a Deductive Viewpoint. Artificial Intelligence, 35, 401-413.

Forgy, C. L. (1982). Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. Artificial Intelligence, 19(1), 17-37.

Gupta, A. (1986). Parallelism in Production Systems. Ph.D. dissertation, Department of Computer Science, Carnegie Mellon University.

Miranker, D. P. (1987). TREAT. A New and Efficient Match Algorithm. Ph.D. dissertation, Columbia University.

Pasik, A. (1987). Improving Production System Performance on Parallel Architectures by Creating Constrained Copies of Culprit Rules. Department of Computer Science, Columbia University.

Perlin, M. W. (1989). Constraint-Based Specification of Production Rules. School of Computer Science, Carnegie Mellon University.

Scales, D. J. (1986). Efficient Matching Algorithms for the Soar/Ops5 Production System. Master's thesis, Stanford University.

Tambe, M., & Newell, A. (1988). Some Chunks Are Expensive. Proceedings of the Fifth International Conference on Machine Learning (pp. 451-458).

Waterman, D. A., & Hayes-Roth, F. (1978). Pattern Directed Inference Systems. New York: Academic Press.

Appendix: Mathematical Development

We begin with a number of definitions. After setting up a commutative diagram, we shall prove some facts about binding and tuple spaces. The key result needed for the Match Box algorithm is that an inverse image of a binding is a subspace (i.e., a box) in tuple space.

Suppose we have a set $\{x_j\}$ of variables, $j \in J = \{1, 2, ..., m\}$. Each variable x_j draws its values from a corresponding (possibly restricted) value set V_j. A binding of the variables is therefore an element of the product set $V_1 x V_2 x ... x V_m$. The set of possible bindings, or the binding space, is $B \equiv \Pi_J V_j$.

A rule imposes certain relations on the variable; a binding is valid exactly when it satisfies these relations. A single relation is an ordered pair (ρ_i, J_i), where $\rho_i \in \{=, <, ...\}$ has arity ≥ 2, and $J_i \subseteq J$. For example, the relation $(=, \{1,3,4\})$ imposes the ternary equality relation $x_1 = x_3 = x_4$. Let $\rho \equiv \{(\rho_i, J_i)\}$ be the set of relations for a rule. Each relation may be operationalized into a test

$$\tau_i : B \rightarrow 2$$

$$b \rightarrow \text{TRUE, if the projection } \Pi_{J_i}(b) \text{ satisfies } \rho_i;$$
$$\text{FALSE, otherwise,}$$

where 2 is the two point set {TRUE, FALSE}. The conjunction of these tests $\{\tau_i\}$ determines the test

(1) $$\tau : B \rightarrow 2$$

$$b \rightarrow \text{TRUE, if } \forall i, \tau_i(b) \text{ is TRUE;}$$
$$\text{FALSE, otherwise.}$$

A different perspective arises from examining the tuple space T formed from the data objects D. Let D be the set of (working memory) data objects, and P(D) its power set. Our ultimate objective is to find n-tuples of objects satisfying the binary (or greater arity) relations ρ, as well as the unary relations on each component. Let $K=\{1,2,...,n\}$ be the tuple's index set, and α_k: $P(D) \rightarrow P(D)$ the filter which performs the unary tests and maps, e.g., D into its filtered subset D_k. Then tuple space T is formed as the set product $D_1 x D_2 x \ldots x D_n$. and $\alpha \equiv \Pi_K \alpha_k$ is the product map.

(2) $\alpha(D) = T$,

forms all possible tuples (satisfying the unary predicates α_k) from working memory D.

We now connect tuple space T with binding space B. Each tuple coordinate $k \in K$ has an associated set of labels L_k, used for accessing attributes of the kth object in the tuple. A particular rule is, in general, only interested in a subset (or abstraction) of the accessors

$$A \subseteq K x L_k.$$

Each of these accessors is mapped into a variable via the one-one onto map

$$\mu: A \rightarrow J.$$

When each attribute in A is assigned a value, we have the rule's abstraction of a tuple. Assigning values to j in J forms a binding. Therefore, considering sets of values tranforms $\mu:A \rightarrow J$ into a new function $f:T \rightarrow B$. Then

(3) $f: T \rightarrow B$

$t \rightarrow b$, the binding corresponding to the accessors of tuple t.

Combining (1), (2), and (3), we have the diagram:

$$\begin{tikzcd} & P(D) \arrow[dl, "\alpha"'] & \\ T \arrow[rr, "f"] & & B \arrow[dl, "\tau"] \\ & 2 & \end{tikzcd}$$

This diagram is usually completed in production systems by introducing the map $\beta = \tau \circ f$:

$$\begin{tikzcd} & P(D) \arrow[dl, "\alpha"'] & \\ T \arrow[rr, "f"] \arrow[dr, dashed, "\beta"'] & & B \arrow[dl, "\tau"] \\ & 2 & \end{tikzcd}$$

where β represents the join tests performed on two or more object in a tuple t by executing the composition $\tau \circ f$.

The objective of rule-based pattern matching is to find the set I of instantiations, where $I \equiv \beta^{-1}(\mathrm{TRUE})$. This search for $I \subseteq T$ is sometimes termed k-space (i.e., tuple space) search (Tambe & Newell, 1988). Our commutative diagram can be used to describe various tuple space search strategies, such as RETE, TREAT, and variant network topologies. We next use the diagram to introduce a new search strategy.

Let $B_0 \equiv \tau^{-1}(\mathrm{TRUE})$, the set of <u>valid</u> <u>bindings</u> satisfying the relation ρ. Since $\beta^{-1} = f^{-1} \circ \tau^{-1}$,

$$\begin{aligned} f^{-1}(B_0) &= f^{-1} \circ \tau^{-1}\ (\mathrm{TRUE}), \\ &= \beta^{-1}(\mathrm{TRUE}), \\ &= I. \end{aligned}$$

Thus, another method for computing the instantiation set I is to compute $f^{-1}(B_0)$.

To increase search efficiency, we first decrease the size of B. We do this by merging the variables of each equality relation into a single variable, and replacing the equality test with a consistency check. For example, $(=, \{1,3,4\})$ is executed by checking that the three attributes $\mu^{-1}(1)$, $\mu^{-1}(3)$, and $\mu^{-1}(4)$ are in fact equal, and then storing the consistent attribute (if it exists) into the new variable x_1'; variables x_1, x_3, and x_4 are discarded.

This leads to a new (not necessarily one-one) map μ from A onto (a possibly smaller) J. Transforming $\mu{:}A \rightarrow J$ to sets of values, we have the new function

$f: T \to B_\perp$

$t \to b$, if the consistency checks hold,

$\perp$, otherwise,

as defined in the following paragraph. B has been lifted to $B_\perp = B \cup \{\perp\}$, where $\perp$ denotes inconsistency. We have augmented $f{:}T \to B_\perp$ to perform all the equality tests, while $\tau{:}B_\perp \to 2$ now evaluates only the nonequality relations.

We more precisely define f as

$f: T \to B_\perp$

tuple $t \to$ binding b,

where b_j is defined by the operations:

1. Let $A' = \mu^{-1}(j)$

   $= \{a \in A \mid \mu(a) = j\}$.

   (A' is nonempty, since m is onto;

   hence $\exists$ tuple coordinate k having an accessor $a_{k,j}$ constrained by j.)

2. Let $S = \cap_{A'}$ value[a, t],

   where value[a,t] evaluates the accessor a on tuple t,
   retrieving its value as a singleton set.

3. If $S \neq \varnothing$, then all a in A' agree, and b_j equals the single value of S.

   If $S = \varnothing$, though, then an inconsistency has been detected, so $b_j = \perp$.

If any $b_j = \perp$, then $b = \perp$.
Combining the jth coordinate values, we have

$f(t) = \Pi_J \cap_K$ value[a,t].

Consider two limiting examples in the diagram:

$$\begin{tikzcd}
 & P(D) \arrow[dl, "\alpha"'] & \\
T \arrow[rr, "f"] & & B_{\_} \\
I \arrow[u, phantom, "\cup"] & & B_0 \arrow[ll, "f^{-1}"'] \arrow[u, phantom, "\cup"] \\
 & \text{true} \arrow[ul, "\beta^{-1}"] \arrow[ur, "\tau^{-1}"'] &
\end{tikzcd}$$

1. If ρ is comprised entirely of equality relations, then f does all the work, τ is trivial, and $B=B_0$.

2. If ρ has no equality relations, then μ is one-one, f does no consistency checking, and τ performs all the tests.

We now explore the finer grained structure of f needed to work with objects in D, rather than tuples. The attribute set A restricted to the coordinate k is $A_k = \{(k,i) \mid i \in L_k\}$. Clearly, A is partitioned into its disjoint A_k, $k \in K$. These inclusions induce the maps μ_k,

$$\begin{tikzcd}
A_k \arrow[r, hook, "i_k"] \arrow[dr, "\mu_k"'] & A \arrow[d, "\mu"] \\
 & J
\end{tikzcd}$$

Transforming to sets of values, we induce the family of functions f_k,

$$\begin{tikzcd}
D_k \arrow[r, hook, "i_k"] \arrow[dr, "f_k"'] & T = \prod_K D_k \arrow[d, "f"] \\
 & B = \prod_J V_j
\end{tikzcd}$$

More precisely,

f_k: $D_k \to \mathrm{Sub}(B)$

object $d_k \to S$, a subspace of B, Dim(S) = Dim(B),

where the jth coordinate projection of S

$\pi_j(S)$ is defined by the operations:

1. Let $A'_k = \mu_k^{-1}(j)$.

    By the unary α filtering, the jth variable refers to the kth object at most once. So the cardinality $\#A'_k = 1$, if f constrains the kth tuple coordinate by checking the jth variable; 0, otherwise.

2. If $\#A'_k = 1$, $A'_k = \{a\}$,

    $\pi_j(S)$ = singleton set comprised of value[a,t], i.e., the constrained value a.

    If $\#A'_k = 0$, then x_j is unconstrained by f_k, so

    $\pi_j(S) = V_j$.

So $f_k(d_k) = \Pi_J$ value$[a_{k,j},t]$.

The coordinate restrictions $\{f_k\}$ can be combined to form the function f.

<u>FACT</u>. $f = \cap_K f_k \circ \pi_k$, where π_k is the projection of the kth coordinate.

<u>Proof</u>. Let $t = <d_1, ..., d_n> \in T = \Pi_K D_k$.

$\cap_K f_k \circ \pi_k(t) = \cap_K f_k(d_k)$

$= \cap_K \Pi_J$ value$[a_{k,j},t]$,

i.e., the value or the V_j identity set,

$= \Pi_J \cap_K$ value$[a_{k,j},t]$,

$= f(t)$.

We now demonstrate the

<u>LEMMA</u>. Let $h: P \rightarrow Q$, with $P = \Pi_K(P_k)$ a product space, and $h = \cap_K (h_k \circ \pi_k)$ a well formed function, and

$$\begin{tikzcd} P_k \arrow[r, hook] \arrow[dr, "h_k"'] & P \arrow[d, "h"] \\ & Q \end{tikzcd}$$

Then $h^{-1} = \Pi_K(h_k^{-1})$.

Proof. Let $q \in Q$.

$$\begin{aligned} h^{-1}(q) &= \{ p \mid q = h(p) \}, \\ &= \{ p \mid q = \cap_K h_k \circ \pi_k(p) \}, \\ &\qquad \text{by the well-formedness of the function,} \\ &= \{ p \mid q \in h_k \circ \pi_k(p), \forall k \in K \}, \\ &= \{ p = <p_1, p_2, ..., p_n> \mid q \in h_k(p_k), \forall k \in K \}, \\ &\qquad \text{by the independence of the coordinates,} \\ &= \Pi_K \{ p_k \mid q \in h_k(p_k) \}, \\ &= \Pi_K h_k^{-1}(q). \end{aligned}$$

Since $f = \cap_K f_k \circ \pi_k$, by the Lemma we have

THEOREM. $f^{-1} = \Pi_K \ f_k^{-1}$.

COROLLARY. For any binding $b \in B$, the set of tuples $f^{-1}(b)$ mapping into b forms a subspace of T. We call $f^{-1}(b)$ the match box of b.

When $f^{-1}(b)$ achieves full dimensionality n, binding b then has support in tuple space. For example, for n=3, if

$$\begin{aligned} f_1^{-1}(b) &= \{\text{the object } d_1\}, \\ f_2^{-1}(b) &= \{d_2, d_3\}, \\ f_3^{-1}(b) &= \{\}, \end{aligned}$$

then b has no support. Should adding object d_4 augment $f_3^{-1}(b)$ to $\{d_4\}$, then

$$f^{-1}(b) = f_1^{-1}(b) \times f_2^{-1}(b) \times f_3^{-1}(b),$$

$$= \{d_1\} \times \{d_2, d_3\} \times \{d_4\},$$

$$= \{< d_1, d_2, d_4>, < d_1, d_3, d_4>\}.$$

Thus, tuple formation has been reduced to the independent routing of objects.

To correctly route the data objects through B_0, we introduce the map $\sigma_k = f_k \circ \alpha_k$

$$\begin{tikzcd}
& P(D) \arrow[dl, "\alpha_k"'] \arrow[dr, "\sigma_k"] & \\
D_k \arrow[rr, "f_k"] & & B
\end{tikzcd}$$

which places objects d in D_k into their proper bindings in binding space B. Performing σ_k for every $k \in K$ gives $\sigma = \cup_K \sigma_k$:

$$\begin{tikzcd}
& P(D) \arrow[dl, "\alpha"'] \arrow[dr, "\sigma"] & \\
T \arrow[rr, "f"] & & B
\end{tikzcd}$$

The match box algorithm performs σ on all objects d, and monitors $f^{-1}(B_0)$ for tuple formation. That is, the diagram traversal for computing $I \subseteq T$ is reversed from $\tau \circ f \circ \alpha$ object testing, to $f^{-1} \circ \sigma$ object routing:

$$\begin{tikzcd}
& P(D) \arrow[dr, "\sigma"] & \\
I & & B_0 \arrow[ll, "f^{-1}"'] \\
& \text{true} \arrow[ul, dashed, "\beta^{-1}"] \arrow[ur, dashed, "\tau^{-1}"'] &
\end{tikzcd}$$