FAST MANY-TO-ONE MATCHING ALGORITHMS

by

Paul W. Purdom
Indiana University
Bloomington, IN 47405

and

Cynthia A. Brown
Northeastern University
Boston, Massachusetts 02115

TECHNICAL REPORT NO. 169

FAST MANY-TO-ONE MATCHING ALGORITHMS

by

Paul W. Purdom, Jr., Indiana University
and Cynthia A. Brown, Northeastern University

May, 1985

This material is based on work supported by the National Science Foundation under Grant No. MCS 83-01742.

This report to appear in the Proceedings of the 1st Int'l. Conference on Rewriting Techniques & Applications, Dijon, France, May 20–22, 1985.

FAST MANY-TO-ONE MATCHING ALGORITHMS

Paul Walton Purdom, Jr.
Indiana University
Bloomington, IN 47405

and

Cynthia A. Brown
Northeastern University
Boston, Massachusetts 02115

Matching is a fundamental operation in any system that uses rewriting. In most applications it is necessary to match a single subject against many patterns, in an attempt to find subexpressions of the subject that are matched by some pattern. In this paper we describe a many-to-one, bottom-up matching algorithm. The algorithm takes advantage of common subexpressions in the patterns to reduce the amount of work needed to match the entire set of patterns against the subject.

The algorithm uses a compact data structure called a *compressed dag* to represent the set of patterns. All the variables are represented by a single pattern node. Variable usages are disambiguated by a *variable map* within the pattern nodes. Two expressions that differ only in the names of their variables are represented by the same node in the compressed dag. A single compressed dag contains all the patterns in the system.

The matching proceeds bottom-up from the leaves of the subject and set of patterns to the roots. The subject keeps track of the variable bindings needed to perform the match. A hashing method is used to speedily retrieve specific nodes of the compressed dag.

The algorithm has been implemented as a part of a Knuth-Bendix completion procedure. In comparison with the standard matching algorithm previously used, the new algorithm reduced the number of calls to the basic match routine to about 1/5 of their former number. It promises to be an effective tool in producing more efficient rewriting systems.

Introduction

A *matching* of a term p to a term s is an assignment of values to the variables of p that makes it equal to s. Matching is a necessary preliminary step in applying a rewrite rule to an expression, and as such it is a fundamental operation in any system that uses rewriting.

In a typical rewrite system, an expression (the *subject*) must be simplified by applying any rule from a set of rewrite rules whose left side (the *pattern*) matches any of the subject's subexpressions. Thus, each subexpression of the subject must be tested against the entire set of patterns until a match is found. Using the traditional dag (directed acyclic graph) algorithm, the time for this process is at best proportional to the number of patterns times the size of the subject; at worst, it is proportional to the size of the subject times the size of the set of patterns. (The size of an expression is the number of nodes in its dag representation.) The idea of a many-to-one matching algorithm is to represent the patterns in such a way that much of the work involved in a repetitious application of ordinary matching can be avoided.

A thorough study of linear pattern matching on trees was done by Hoffmann and O'Donnell [HOD82], who developed fast bottom-up algorithms for many-to-one matching using linear patterns. (A linear expression is one in which each variable occurs only once.) In this paper we build on their ideas to develop a fast algorithm for arbitrary many-to-one matching. The algorithm makes use of a highly compact representation of the set of patterns which we call a *compressed dag*. It was designed to be used in the context of the Knuth-Bendix completion procedure [KB70], and therefore allows for dynamic updating of the pattern representation. Matching consumes a major portion of the time in a typical run of the Knuth-Bendix procedure (in our measurements it was applied ten times as much as unification and was by far the most frequently used basic operation), so an improvement in the efficiency of matching should lead to a major speed-up of the Knuth-Bendix procedure. Similar gains can be expected for other systems that make heavy use of many-to-one matching.

The Compressed Dag

We begin by describing the data structures used by the algorithm. The fundamental structure is the *compressed dag* used to represent the set of patterns. A dag is typically used to represent an expression that may contain common subexpressions. Each unique subexpression is represented by exactly one node in the dag; if two expressions use the same subexpression, they both point to that node. An expression in which all subexpressions are unique would be represented by a dag which was a tree. Figure 1 shows the dag for the expression $X * Y + X * Z$ (where X, Y, and Z are variables).

An ordinary dag contains a distinct leaf node for each *distinct* variable in the expression it represents. The compressed dag contains a *single* node representing any variable. The difference between a dag and a compressed dag is therefore in the representation of subexpressions that differ only in the names of their variables. In a dag, these subexpressions have distinct nodes; in a compressed dag, they are represented by the same node. One problem remains: keeping track of the use of variables.

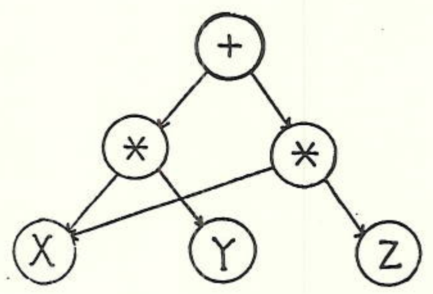

Figure 1. A dag for the expression $X * Y + X * Z$

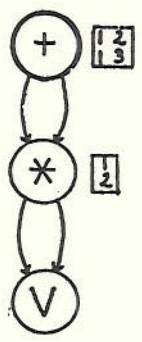

Figure 2. A compressed dag for the expression of Figure 1.

Variable names are not of any particular significance in themselves; they are simply indicators for where the same quantity must be substituted uniformly in the expression. Therefore, the information that must be given when the variable nodes are combined has to do with where and how a variable is used repeatedly. In the compressed dag, the variables of each node (i.e., the variables used in the expression represented by the node) are renamed using unique names of the form $V_1, V_2, \ldots, V_n$ from left to right in order of first occurrence. For each node, the *variable map* records how the variables of the subnodes are identified with variables of the node.

Conceptually, the variable map may be thought of as an uneven array, with a row for each child of the current node. Each row has a number of positions equal to the number of distinct variables in the corresponding child. The entries in the row tell which variable in the current node is associated with each variable in the child.

For example, consider Figure 2, which shows the compressed dag for the expression $X * Y + X * Z$. There are three distinct variables in this expression, so renaming gives $V_1 = X, V_2 = Y$, and $V_3 = Z$. The two main subexpressions, $X * Y$ and $X * Z$, are the same except for variable names, as they are represented by the same node, which stands for the expression $V_1' * V_2'$. The variable map for the main expression is shown as a small rectangle to the right of the node. The first row corresponds to $V_1 * V_2$ (i.e., to $X * Y$). Its first position contains a 1, showing V_1' is mapped to V_1; its second position contains a 2, showing V_2' is mapped to V_2. The second row of the variable map corresponds to $V_1 * V_3$ $(X * Z)$. Its first position contains a 1, showing that V_1' maps to V_1; its second position is a 3, showing that V_2' maps to V_3.

The variable map for the node $V_1' * V_2'$ is very simple. Each child is just a variable, so the rows contain only one position. The child in each case is the unique variable node. The 1 in the first row shows that the first variable is V_1', while the 2 in the second row shows that the second variable is V_2'. (The expression $X * X$ would have a separate node in this compressed dag.

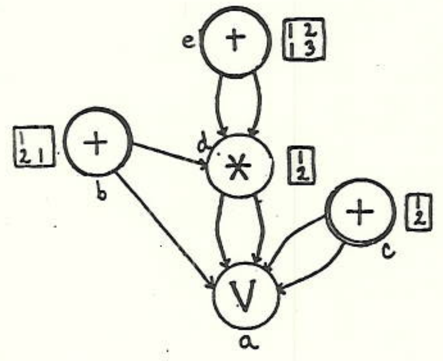

Figure 3. A compressed dag for the patterns $X*Y+X*Z$, $X+Y*X$, and $X+Y$.

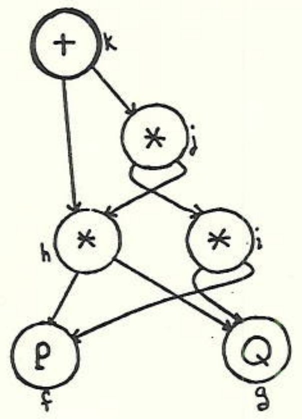

Figure 4. A dag for the expression $P*Q+(Q*P)*(P*Q)$.

Its variable map would have two rows, each containing a 1.) Using the compressed dag and the variable maps, it is easy to construct an expression identical to the original expression except for variable names.

So far we have described the compressed dag for a single expression. In the many-to-one matching algorithm, *all* the patterns are represented using a single compressed dag with multiple roots, one for each pattern. Figure 3 shows the compressed dag for the set of patterns $X*Y+X*Z$, $X+Y*X$, and $X+Y$. Each node in the compressed dag has been given a letter name, for ease of reference later.

The Matching Algorithm

The purpose of the matching algorithm is to find subexpressions of the subject that are matched by a pattern, and to keep track of the bindings of variables in the pattern necessary to obtain the match. The basic idea, which we adopted from [HOD82], is to use a bottom-up approach. In this section we give a high-level description of the matching algorithm; some implementation details are left for the following section.

Unlike the set of patterns, the subject is represented as an ordinary dag. Matching begins at the leaves of the subject. If any pattern contains a variable, then the set of patterns has a variable node, which matches any node in the subject. In addition, any constant node in the subject is matched by the corresponding constant node in the set of patterns, if it exists. Each node in the subject keeps a list of the pattern nodes that match it, together with the variable bindings needed to perform the match. These bindings map each variable in the pattern node to a node in the subject. For example, if there is a variable node in the set of patterns, then each node in the subject records that it is matched by the variable node, and that a binding of the variable is the subject node itself.

After all the children of a subject node have been matched, we attempt to match the node itself. Taking the operator of the node and the first child, we check to see whether there is a match for them in the set of patterns: a pattern node that matches the first child and has a parent with the correct operator as its value. The following steps have to be repeated for each such match. The first step is to record the variable bindings for the parent pattern node that are implied by the match of the first child, using the variable map. We then check for a match of the second child of the pattern node against the second child of the subject that is consistent with the variable bindings already established. That is, we use the variable map to ensure that variables of the pattern node that occur in both children have been bound to the same subject node in both children during the bottom-up match. Subsequent children are checked in the same way as the second one. If a node and all its children are matched, then the complete match is successful and is recorded in the subject node. Matching a subject node with the root of a pattern signals an opportunity to apply a rewriting rule.

As an example, consider the patterns of Figure 3 and the subject $P*Q+(Q*P)*(P*Q)$, whose dag is shown in Figure 4. Proceeding from the leaves upward, both node f (which represents expression P) and node g (representing Q) are matched by the variable node (node a) in the set of patterns. Nodes h (representing $P*Q$) and i (representing $Q*P$) are matched by the variable node and by node d (representing $V_{d1} * V_{d2}$) in the set of patterns. Node h matches d with V_{d1} set to f and V_{d2} set to g; i matches d with V_{d1} set to g and V_{d2} set to f.

Now consider node j (representing $(Q*P)*(P*Q)$). Each of its children has two matches, one to the variable and one to d. However, node d has no parent whose operator is a $*$, so the only match (besides the match to the variable) is to d again, with V_{d1} set to i and V_{d2} set to h.

Now we are ready to match k, the main node of the subject. Its operator is $+$, so we must look for $+$ nodes in the set of patterns whose first child has been matched against the first child of k. There are three such nodes, b, e, and c. For each of these nodes in turn, we consider whether the match of its first child against h is compatible with a match of the second child against j.

The first child of b (b represents the expression $V_{b1}+V_{b2}*V_{b1}$) is the variable node. It matches h using V_a set to h. Thus we must have V_{b1} set to h, by the variable map. To extend this match of k and its first child to include the second child (j), we must find a match to j by the second child of b that is compatible with setting V_{b1} to h. This requires matching d to j with V_{d2} set to h, since the variable map for b tells us that the first variable of the node (i.e., V_{b1}) is identified with the second variable of its second child (i.e., V_{d2}). Looking at the matches for j, the match to the variable can be disregarded because it matches j with a rather than with d, but the match

Subject Node	Pattern Node	Variable Assignments
f	a	$V_a = f$.
g	a	$V_a = g$.
h	a	$V_a = h$.
	d	$V_{d1} = f,\ V_{d2} = g$.
i	a	$V_a = i$.
	d	$V_{d1} = g,\ V_{d2} = f$.
j	a	$V_a = j$.
	d	$V_{d1} = i,\ V_{d2} = h$.
k	a	$V_a = k$.
	b	$V_{b1} = h,\ V_{b2} = i$.
	c	$V_{c1} = h,\ V_{c2} = j$.

TABLE 1. Summary of matches of Figure 4 against Figure 3.

to d matches the correct node and also agrees on the value for V_{b1}. This match also sets V_{d1} to i, which implies that V_{b2} is set to i. We thus have a complete match of b to k, with V_{b1} set to h and V_{b2} set to i.

The second pattern node to consider is e. We have a match for its first child, d, to h, with V_{d1} set to f and V_{d2} set to g. By the variable map, this requires setting V_{e1} to f and V_{e2} to g. To extend the match, we must find a match to j by the second child of e that agrees with these settings of the variables. One of the matches of j is to a; the second one is to d, but it requires setting V_{e1} to h, so our partial match of e to k cannot be extended.

The third + node in the pattern is c (representing $V_{c1} + V_{c2}$). We match the first child of c (the variable) with V_a set to h, requiring that V_{c1} be set to h, and the second child (also the variable) with V_a set to j, requiring that V_{c2} be j. This gives another complete match.

Table 1 summarizes the results of this example, showing all the matches for each node in Figure 4 against the set of patterns in Figure 3.

Implementation Considerations

In the previous section we described the basic concept of the matching algorithm; in this section we explain some modifications that lead to a more efficient implementation. In doing the bottom up match it is important to be able to quickly locate parent nodes on the basis of their children and their operator. There are several good approaches to this problem; the one we have investigated uses hashing and a modified version of the compressed dag data structure.

We first describe the modified data structure. Each parent node in the original compressed dag is broken up into n *partial parent* nodes, where n is the number of children of the original

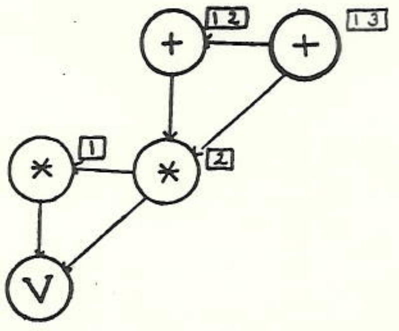

Figure 5. The curried version of the dag of Figure 2.

node. Each of the nodes contains the operator and two pointers: one to its child and one to the previous node in the series (the previous node pointer is **nil** in the partial parent of the first child). As in our earlier dags, there is a unique node for a given set of pointers and variable map, so expressions with the same operator and first few operands share their first few nodes. We call this data structure a *curried dag*, since the transformation is reminiscent of the process of currying a lambda expression. Figure 5 shows the curried dag for the dag of Figure 2.

The curried dag is used in combination with a hash table. The hash table is built bottom-up, just as the set of patterns is. There is an entry in the hash table for the variable node and for each constant. There is also an entry for each set of non-leaf nodes that differ only in their variable maps; it is hashed to using the value of its operator, its current child pointer, and its previous node pointer. These sets nearly always have one element.

To match a nonleaf node, we check for matches for its first child. For each such match, we look in the hash table for the list of nodes having the correct operator, the pattern that matched the first child of the subject node as its child, and a **nil** previous node pointer. For each node on the list, its variable map is used to establish an appropriate set of variable bindings; we then continue by looking at the matches for the second child of the subject. For each such match, we attempt to find a list of nodes in the hash table that has the matching pattern node as its current child, the main operator of the subject node as its operator, and the node that matched the first child and the main operator as its previous node. If such a list is found, we check each element's variable bindings to see whether it can be used to extend a previous partial match. We continue in the same way until a complete match (or matches) is found. The list of matches for a node usually contains just one or two elements; it contains at least one if there is a variable node, since the variable node matches everything.

The use of curried dags helps to reduce the number of inquiries to the hash table. If nodes were stored in the hash table according to the main operator and all the children, then for a subject with an n-ary operator, each of whose operands had been matched in k ways, it would require n^k inquiries to determine which combinations were present. With a curried dag, some possible values for low-numbered children will usually be eliminated. If a total of k possible matches survive at each stage, then $k + (n-1)k^2$ inquiries will be needed altogether. If all the operators are unary or binary, then there is no advantage to using a curried dag.

An alternate approach is to have each pattern node contain pointers to those nodes for which it is the first child. To make access more rapid, the pointers can be stored in lists according to the main operator of the parent, and the lists can be indexed by the operators. Partial parents of n children would require a list of pointers to the appropriate partial parents of $n+1$ children. The trade-off between the two approaches depends on the nature of the patterns and on the average number of matches. If there tend to be a small number of successors (nodes with one more child) for each partial parent, the alternate method works well, while if there tend to be a small number of matches for each child, then the main method works well.

Because of the way the variable maps are constructed, each variable of the first child of a node is mapped to the variable in the parent with the same number. This means that a parent node need not include a variable map for its first child. Significant time can also be saved by giving special treatment to matches with the variable node.

Performance

The goal in developing the many-to-one bottom-up matching algorithm was to make matching time proportional to the size of the subject, and independent of the number or size of the patterns. We know of no way to achieve such good performance in the worst case, so we need to consider our algorithm's typical behavior.

In Hoffman and O'Donnell's bottom-up method, [HOD82], extensive preprocessing of the patterns is used to produce a table that drives the matching phase. Given the sets of patterns matched by the left child and right child of a node, the table precomputes the set of patterns that can be matched by the node itself. Provided the table fits in memory, the matching is guaranteed to run in a time that is linear in the size of the subject. In the worst case, the size of the table can be exponential in the size of the patterns. Hoffmann and O'Donnell identified interesting sets of patterns on which the table size is guaranteed to be reasonably small.

We have taken a somewhat different approach to this problem. Our preprocessing phase (which builds the curried dag and the hash table) is linear in the size of the patterns. For our intended applications it is also important to be able to change the set of patterns dynamically at a moderate cost, and our method achieves that goal. Instead of precomputing the set of possible matches for a given set of matches on the children, we keep lists of matches on the children and

Example	Standard	New	Other
group	4055	885	887
central groupoid	87145	4828	11737

TABLE 2. Matches on newly proposed rules by standard and new matching algorithms while processing examples from [KB70]. The Other column shows the matches of old rules against new rules; both programs used the standard algorithm for these.

explore each list as it arises. As long as the lists stay reasonably small, this does not waste a significant amount of time, and it uses much less space than the other approach.

We implemented the matching algorithm as part of a Knuth-Bendix completion program. The program is written in WEB, the Pascal programming system created by Knuth [Kn84]. The new algorithm replaces a straightforward top-down one-rule-at-a-time matching algorithm. We ran a number of examples using both the new and the old algorithms. Table 2 summarizes the results of these tests on two important examples from [KB70]: the first group theory example (Example 1) and the second central groupoid example (Example 16 with four axioms). The Standard column of the Table gives the number of matches done on newly-proposed rules by the standard top-down matching algorithm. The New column gives the corresponding number for the new algorithm. Both programs used the standard method when testing old rules using a new rule as a pattern; the number of such matches is shown in the Other column. The programs were almost identical except for the matching algorithm that was used.

Table 2 shows the promise of the new matching algorithm. It greatly reduces the number of of matching operations required in the Knuth-Bendix procedure, and should also lead to a smaller running time. In the present implementation, both methods run at about the same speed (1.5 seconds for the standard method versus 1.8 seconds for the new on the group theory example; 15.0 seconds for the standard versus 12.8 seconds for the new on the central groupoid.) This implementation does not take full advantage of the new method, since it must keep a representation of each rule in both compressed and uncompressed form, leading to considerable extra overhead.

The Knuth-Bendix procedure does three kinds of processing that involve pattern-matching and its variants: matching a subject against many patterns, in an attempt to simplify it; matching a single pattern against many subjects, when trying to see if a new rule can be used to simplify any of the existing rules; and unification. Moreover, rules are used both as patterns and as subjects. At present the patterns in our algorithm are represented as a compressed dag, while the subject is an ordinary dag; this requires that the rules be stored in both forms by the system. It is desirable to find algorithms for the two types of pattern matching and for unification which use compressed dags as the only structure for storing terms. There appears to be no real difficulty in doing this. The current approach was selected because it is a little simpler.

The troubling aspect of our measurements is that the running time of our new algorithm did not improve rapidly with problem size. There are two reasons for this. First, with the new algorithm, matching was no longer taking much of the time, while the overhead of maintaining two data structures was. This overhead will disappear once algorithms that need a single data structure are programmed. Second, the average number of matches per node is large enough that algorithms that depend quadratically on the average number of matches are slower than one would like. (For the central groupoid example, 5 matches per node was common.) We plan, therefore, to also carefully investigate the alternate algorithm of the previous section, since its time increases linearly with the average number of matches per node.

Extensions

Extensions of the Knuth-Bendix procedure that use associative and commutative (ac) unification and matching [St81] promise to extend its usefulness to a wide class of equational systems. The matching algorithm was designed so that it would be straightforward to extend it to do ac matching.

While matching takes up the major portion of the Knuth-Bendix procedure's time, unification is also a significant factor in the running time. There is a need for many-to-one unification in the Knuth-Bendix procedure. It is possible to adapt existing unification algorithms to run on compressed dags. However, these existing algorithms proceed in a top-down manner and so would get no advantage from the special properties of the compressed dag. It is an interesting question as to whether an efficient bottom-up unification algorithm can be developed.

References

[HOD82] Christoph M. Hoffmann and Michael J. O'Donnell, *Pattern Matching in Trees*, JACM **29** (1982) pp. 68-95.

[KB70] Donald E. Knuth and Peter B. Bendix, *Simple Word Problems in Universal Algebras*, in *Computational Problems in Universal Algebra* (edited by John Leech), Pergamon (1970) pp. 263–297.

[Kn84] Donald E. Knuth, *Literate Programming*, The Computer Journal **27** (1984) pp 97–111.

[St81] Mark E. Stickel, *A Unification Algorithm for Associative-Commutative Functions*, J. ACM **28** (1981) pp 423–434.