Pattern Matching in Trees

CHRISTOPH M. HOFFMANN AND MICHAEL J. O'DONNELL

Purdue University, West Lafayette, Indiana

ABSTRACT. Tree pattern matching is an interesting special problem which occurs as a crucial step in a number of programming tasks, for instance, design of interpreters for nonprocedural programming languages, automatic implementations of abstract data types, code optimization in compilers, symbolic computation, context searching in structure editors, and automatic theorem proving. As with the sorting problem, the variations in requirements and resources for each application seem to preclude a uniform, universal solution to the tree-pattern-matching problem. Instead, a collection of well-analyzed techniques, from which specific applications may be selected and adapted, should be sought. Five new techniques for tree pattern matching are presented, analyzed for time and space complexity, and compared with previously known methods. Particularly important are applications where the same patterns are matched against many subjects and where a subject may be modified incrementally Therefore, methods which spend some time preprocessing patterns in order to improve the actual matching time are included

Categories and Subject Descriptors: F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*pattern matching*, G 2 2 [**Discrete Mathematics**]: Graph Theory—*trees*

General Terms Algorithms, Theory

Additional Key Words and Phrases. incremental pattern matching, bottom-up matching, top-down matching, subtree replacement systems, interpreter generation, theorem proving

1. *Introduction*

Many computing techniques involve simplifying expressions (trees) by repeatedly replacing special types of subexpressions (subtrees) according to a set of replacement rules. For example,

(1) Hoffmann and O'Donnell [14] show how tree replacements may be used in automatically generated interpreters for nonprocedural programming languages. The defining equations for the programming language are taken as the replacement rules. An interpreter may then process an input expression by replacing subexpressions according to the given rules until no more replacements are possible. Interpreters may be generated which are absolutely faithful to the semantics of the language as given by the defining equations. The tree-replacement approach is very convenient for producing interpreters for existing languages such as LISP and LUCID or for implementing experimental languages. Elsewhere, the merits of the language of equations as a programming language in its own right are examined [15].

(2) Guttag et al. [12] and Wand [34] suggest that defining equations may be treated as tree replacement rules to yield direct implementations of abstract data types. Guttag et al. [13] describe a working system based on this idea, as does Goguen [11].

This work was supported in part by the National Science Foundation under Grant MCS 78-01812.

Authors' address: Department of Computer Science, Purdue University, West Lafayette, IN 47907

Such a system does not differ in essence from the interpreters or equational programs in (1) but in this case would be embedded into a procedural language as subroutine.

(3) Intermediate code produced by a compiler may be represented by trees. Certain types of code optimizations, for example, the elimination of redundant operations and constant propagation, may be viewed as replacement rules [10, 16, 33].

(4) In [7] Collins represents algebraic terms as trees and formulates symbolic computation as tree replacements. The replacement rules formalize operations such as differentiation and certain algebraic simplifications.

(5) One approach to the automatic proving of equational theorems is to treat a set of equational axioms as replacement rules and transform one side of the equation to be proved into the other by a sequence of tree replacements. Knuth and Bendix [20] discuss some of the cases in which tree replacements yield efficient theorem provers. Most studies of equational theorem proving, such as [9, 22, 25, 31], have not used the replacement system approach. Chew [6] has recently developed an algorithm combining replacement systems with the methods of Nelson and Oppen [25].

Many of the theoretical properties of tree replacement systems have been studied in [3a, 11, 23, 26, 30]. In this paper we develop theoretically and practically efficient algorithms for one of the key technical issues in implementing replacement systems.

An implementation of a tree replacement system requires practical solutions for the following:

(a) a method for finding subtrees which may be replaced;
(b) a way of choosing the next replacement to be performed;
(c) a way of actually replacing the subtree.

Part (c) is an easy programming problem; (b) is a question which is quite complicated in its theoretical effects. It has been treated abstractly in [26] and algorithmically in [14]. Part (a) is the subject of this paper.

A large part of the overhead in implementing tree replacements comes from the repeated searching for the next subtree to be replaced. This is essentially a tree-pattern-matching problem. We believe that good solutions to the problem of tree pattern matching are a prerequisite for making implementations based on tree replacements competitive in efficiency with ad hoc methods, especially in the realm of interpreters for nonprocedural languages.

Tree pattern matching is analogous to the problem of pattern matching in strings studied in [1, 4, 21]. We consider two essentially different ways of extending the Knuth–Morris–Pratt string-matching algorithm to tree patterns, each with several variations.

One may view first-order unification as a tree-pattern-matching problem [3, 28, 29]. However, first-order unification differs from the tree pattern matching considered here in that a pattern is matched against the entire subject tree and not against proper subtrees as well. Pattern matching in our sense has been studied in [18, 23, 24, 27]. With the exception of [23], these papers examine the problem without considering the specific requirements of subtree replacement systems. Karp et al. [18] give an algorithm which finds all matches of a pattern tree to subtrees of a subject. By preprocessing the pattern(s) involved we get more efficient methods. Recently, Overmars and van Leeuwen [27] have studied tree pattern matching, but with a different class of trees. They discovered independently many of the techniques we develop in Section 8, and their fastest algorithm has a performance equal to our

Algorithm D. We discuss their results and the relationship to our work in Section 9. Kron's work [23] is related to the bottom-up techniques of Sections 3 and 4. We discuss the details at the end of Section 4.

In applications of tree replacements the same set of rules is typically used many times. Preprocessing of the rules is advantageous if it speeds up their application. Each replacement causes a local change in the subject tree. So our pattern-matching techniques should be able to respond incrementally to local changes in the subject to avoid repeated rescanning of the entire tree. For the sake of a simple presentation we discuss each algorithm in terms of a static subject first and then introduce adaptations to handle changing subjects.

In Section 2 we precisely define the matching problem and our criteria for a good solution. The remainder of the paper divides into two parts, corresponding to the two basic approaches we give. Sections 3–7 develop the bottom-up approach to pattern matching. Here we match in a subject tree by traversing it from the leaves to the root. This method is a significant generalization of the Knuth–Morris–Pratt string-matching algorithm. In Sections 8 and 9 we give our second approach, matching top down by traversing the subject root to leaves. While the bottom-up method generalizes string matching, the top-down method reduces tree matching to a string-matching problem.

The bottom-up method is characterized by more expensive preprocessing but faster matching and a better response to local changes. It is developed from the notion of *match sets*—sets of subpatterns which match at a particular tree node. The basic matching algorithm is introduced in Section 3. Properties of match sets are studied in Section 4. Since it turns out that certain tree patterns have exponentially many different match sets, which would lead to an exponential preprocessing algorithm, we introduce in Section 5 a restriction on tree patterns which allows efficient preprocessing algorithms. Section 6 gives the preprocessing algorithm and discusses its relationship with the preprocessing algorithms in [1, 21]. In Section 7 we sketch a better preprocessing algorithm for binary tree patterns.

Sections 8 and 9 give our top-down algorithm and discuss possible improvements. These algorithms have better preprocessing times than the bottom-up method, but the matching times and update behavior are inferior to the bottom-up method. Tree patterns are reduced to strings which are matched along paths in the subject, as in [18]. The preprocessing for this technique is little more than the preprocessing algorithm for string matching [1]. The basic idea of the top-down method lies in the use of counters for coordinating the matches of different path strings. This counting also turns out to be the limiting factor of the algorithm and is responsible for the worst-case bound. We can improve this bound on machines with bit-string operations, as indicated in Section 9.

For the restricted class of tree patterns introduced in Section 5 we have preprocessing algorithms which require

$$O(patsize^2 + patsize^{rank} \times ht)$$

steps. Here *patsize* is the sum of the pattern sizes, *ht* the height of a specific tree which has to be constructed as part of preprocessing, and *rank* the highest rank in the alphabet. In the worst case *ht* may be as big as *patsize*. The actual match, bottom up, requires $O(subsize + match)$ time, where *subsize* is the size of the subject tree and *match* is the number of matches found. For binary alphabets we have a preprocessing algorithm which requires only $O(patsize \times ht^2)$ steps when coupled with a modified bottom up matching algorithm requiring

$$O(subsize \times ht + match).$$

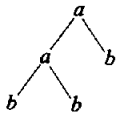


Figure 1

For top-down matching we have an $O(patsize)$ preprocessing algorithm. Here we need no restrictions on the tree patterns. The matching requires

$$O(subsize \times suf \times patno)$$

steps, where *suf* is a quantity depending on the structure of the pattern suffixes (at most equal to the maximum height of a pattern) and *patno* is the number of tree patterns to be matched. For machines with bit-string operations we can, within the same time bound for preprocessing, match using a different technique in only $O(subsize \times patno)$ steps. If each pattern has a height not exceeding the number of bits in a machine word, then this algorithm is of practical importance.

In Section 10 we discuss other possibilities of bottom-up tree pattern matching on machines with bit-string operations, and a trade-off principle for matching time versus preprocessing time and space.

2. *The Tree-Matching Problem*

We are given a finite ranked alphabet Σ of function symbols, including constants as nullary functions. S denotes the set of Σ-terms, formally defined as follows.

Definition 2.1

(i) For all b in Σ of rank 0, b is a Σ-term.
(ii) If a is a symbol of rank q in Σ, then $a(t_1, \ldots, t_q)$ is a Σ-term provided each of the t_i is.
(iii) Nothing else is a Σ-term.

We view Σ-terms as labeled ordered trees. Thus the term $a(a(b, b), b)$ is the tree of Figure 1. Note that the trees $a(a(b, b), b)$ and $a(b, a(b, b))$ are considered to be different. In the following we use "Σ-tree" and "Σ-term" interchangeably.

We are also given a special nullary symbol v, not in Σ, to serve as placeholder for any Σ-tree. We define the set of $\Sigma \cup \{v\}$-terms just as Σ-terms but add to (i) that v is a $\Sigma \cup \{v\}$-term. S_v denotes the set of $\Sigma \cup \{v\}$-terms.

Definition 2.2. A *tree pattern* is any term in S_v. If $b(t_1, \ldots, t_q)$ is a term, then define $\mathrm{son}_i(b(t_1, \ldots, t_q))$ to be t_i for $1 \leq i \leq q$.

We now explain how tree patterns are to be matched in Σ-trees.

Definition 2.3. A pattern p in S_v with k occurrences of the symbol v *matches* a subject tree t in S *at node* n if there exist Σ-trees $t_1, \ldots, t_k$ in S (not necessarily the same) such that the Σ-tree p', obtained from p by substituting t_i for the ith occurrence of v in p, is equal to the subtree of t rooted at n.

Example 2.1. Consider the pattern $p = a(a(b, v), v)$, with two occurrences of the symbol v, and the Σ-tree $t = a(a(b, c), a(a(b, b), b))$. Then p matches t at the two nodes marked in Figure 2. For the match at the root, the trees t_1 and t_2 to be substituted in p are $t_1 = c$ and $t_2 = a(a(b, b), b)$. For the match at the marked interior node we have $t_1 = b$ and $t_2 = b$. □

We wish to solve a matching problem in which we are given a finite set of patterns $p_1, \ldots, p_k$ from S_v and a subject tree t from S and are asked to identify in t every node at which any of the p_i match.

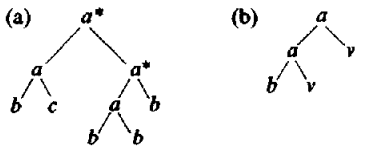


FIG 2. (a) Subject tree. (b) Pattern.

Definition 2.4 (*The Matching Problem*). A *matching problem* consists of a finite set of patterns $p_1, \ldots, p_k$ in S_v and a subject tree t in S. A *solution* to a matching problem is a list of all the pairs (n, i), where n is a node in t and p_i matches at n.

Our definition is motivated principally by algorithmic problems arising in the implementation of subtree replacement systems. Allowing different substitutions for different occurrences of v is equivalent to using a different variable symbol at each occurrence. This restriction is motivated by theoretical problems which arise when repeated variables are permitted in the specification of the replacement axioms [26, Sec. VII].

Note that S_v contains S as subset. Thus every Σ-tree is also a pattern. We develop our results assuming patterns contain at least one occurrence of v, since patterns without variable occurrences are uninteresting from a practical viewpoint. This assumption does not limit our results.

Our matching problem is in some ways more specific, and in some ways more general, than first-order unification. Our use of v corresponds to allowing terms with nonrepeated variables as patterns, while in first-order unification repeated variables are allowed and variables may also appear in the subject. On the other hand, in unification only two trees are matched against each other, and only at the root, whereas we match any number of patterns anywhere in the subject tree.

Definition 2.5. The *size* of a *tree* is the total number of subtrees (equivalently, nodes) in it. The *size* of a *forest* is the sum of the sizes of all trees in it. The *height* of a tree is the number of edges in a longest path from the root to a leaf of the tree.

We are especially interested in applications in which the set of patterns remains fixed and is to be matched against a sequence of subject trees. We therefore consider preprocessing the tree patterns and distinguish *preprocessing time*, involving operations on the patterns independent of any subject tree, and *matching time*, involving all subject dependent operations. Minimizing matching time is the first priority. Preprocessing time is then minimized with respect to a fixed process for matching. Trade-offs between preprocessing time and matching time are considered if the improvement in preprocessing is dramatic and the degradation in matching is small. We also consider the space requirements in preprocessing and matching.

We are especially interested in algorithms which may clearly be adapted to assimilate local changes to the subject without rescanning the entire tree. For bottom-up matching we achieve linear matching times, but preprocessing time may be exponential. To keep bottom-up preprocessing time polynomial, we need some additional constraints on patterns. For top-down matching we lower the preprocessing time to linear, with no restrictions on patterns, at the cost of a slight increase in matching time. The bottom-up method adapts more easily to changes in the subject.

For the remainder of this paper, complexities will be expressed in terms of

patno: the number of different patterns involved
patsize: the size of the pattern forest
subsize: the size of the subject tree
sym: the number of symbols in the alphabet Σ

rank: the highest rank (arity) of any symbol in Σ
match: the number of matches which are found

All suggested methods for tree matching should be compared to the naive algorithm (based on a simple form of unification), which merely tries every pattern at every position in the subject tree. The naive algorithm does no preprocessing but takes $O(patsize \times subsize)$ matching time.

3. *The Bottom-Up Matching Algorithm*

The key idea of the bottom-up matching algorithm is to find, at each point in the subject tree, all patterns and all parts of patterns which match at this point. Let n be a node in the subject labeled with the q-ary symbol b, and suppose we wish to compute the set M of all those pattern subtrees other than v which match at n in the sense of Definition 2.3. (Since v matches anywhere, we always have a match of v.) Suppose we have already computed such sets for each of the sons of n, and call these sets, from left to right, $M_1, \ldots, M_q$. Then M contains v plus exactly those pattern subtrees $b(t_1, \ldots, t_q)$ such that t_i is in M_i, for $1 \leq i \leq q$. Therefore we could compute M by forming trees $b(t_1, \ldots, t_q)$ for all combinations $(t_1, \ldots, t_q)$, where the t_i are chosen from M_i, and then asking whether each candidate for membership in M is a subpattern. Once we have assigned these sets to each node in the subject tree, we have essentially solved the matching problem, since each match is signaled by the presence of a complete pattern in some set.

Note that there can be only finitely many such sets M, because both Σ and the set of subpatterns are finite. Thus we could *precompute* these sets, code them by some enumeration, and then construct tables. Given a node symbol b and the codes of the M_i, these tables give the code for M. In the case of a q-ary symbol b, we would have a q-dimensional matrix for that symbol.

Given such tables, the matching algorithm becomes trivial: Traverse the subject tree in postorder and assign to each node n the code c representing the set of partial matches at n as discussed. The tables consist of arrays, one for each alphabet symbol. If node n is labeled with the q-ary symbol b, then the q-dimensional array for b is used. The code c at n is the value indexed by the tuple $(c_1, \ldots, c_q)$ where c_i is the code assigned to the ith son of n (from the left). If the set represented by c contains the pattern p_i, then the pair (n, i) is added to the solution.

The matching time of this algorithm is clearly $O(subsize)$ for computing all codes plus $O(match)$ for listing the solution. The constant of linearity involves one array reference for computing the codes, a single test to determine whether a complete pattern match is present, plus the overhead for the postorder traversal. Note that the codes may be assigned so that all codes indicating matches are contiguous. The space requirements depend on the table size and are discussed in Section 4.

Example 3.1. Consider a matching problem in which the patterns

$$p_1 = a(a(v, v), b) \qquad \text{and} \qquad p_2 = a(b, v)$$

are to be matched. Assume the alphabet Σ is $\{a, b, c\}$, where a is binary and b and c are nullary symbols. For reasons to be explained later, of the thirty-two possible sets of pattern subtrees only the following five can arise as result of matching:

$$\begin{aligned}
&\text{Set 1} = \{v\},\\
&\text{Set 2} = \{b, v\},\\
&\text{Set 3} = \{a(v, v), v\},\\
&\text{Set 4} = \{a(b, v), a(v, v), v\},\\
&\text{Set 5} = \{a(a(v, v), b), a(v, v), v\}.
\end{aligned}$$

Figure 3

Table for node label a.

	Right son				
Left son	1	2	3	4	5
1	3	3	3	3	3
2	4	4	4	4	4
3	3	5	3	3	3
4	3	5	3	3	3
5	3	5	3	3	3

Table for node label b: 2

Table for node label c: 1

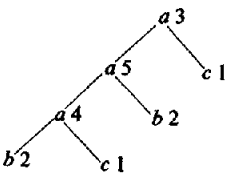


Figure 4

Thus, assigning a 4 to some node n of a subject would indicate that each of the members of Set 4 matches at n. In particular, p_2 matches. Assigning 5 implies a match of p_1.

Figure 3 shows the tables for a, b, and c. For instance, the entry at (3, 2) in the table for a is 5, because at the left son we have a match of both $a(v, v)$ and v, and at the right son of b and of v. For the nullary symbols b and c the tables are 0-dimensional, consisting of one entry each.

Figure 4 shows the complete assignment of codes when using the bottom-up algorithm with these tables. Note that p_1 matches at the node with code 5 and p_2 at the node with code 4. □

There is some similarity between bottom-up matching and formal parsing methods such as LR(k) parsing. In both cases a finite number of possible configurations are precomputed, and tables are formed to drive the parsing/matching process. As with LR(k) parsing, our tables will sometimes be very large, but we isolate a significant class of problems in which the table size is kept small.

When a local change is made to a subject tree, matching codes must be recomputed for the changed portion and some ancestors of the changed portion. In Section 4 we see that the number of ancestors whose codes must be recomputed is bounded by the largest height of a pattern. Note that in these ancestors new matches could appear or old matches disappear. Thus it seems intuitively unlikely that any method could update with less recomputation.

4. *Pattern Relations and Match Sets*

We now turn to studying the sets of partial matches used in the bottom-up matching algorithm of Section 3. We begin by precisely defining these sets and deriving properties which we will later exploit in designing good preprocessing algorithms.

Definition 4.1. Let $F = \{p_1, \ldots, p_k\}$ be a set of patterns in S_v and PF the set of all subtrees of the p_i. A subset M of PF is a *match set* for F if there exists a tree t in S such that every pattern in M matches t at the root and every pattern in PF $- M$ does not match t at the root.

Note that if v is in PF, then v is in every match set. Observe also that the concept of match sets depends on the pattern forest F.

Example 4.1. Consider the pattern forest $F = \{p_1, p_2\}$, where p_1 and p_2 are as in Example 3.1. Then the set $M = \{a(b, v), a(v, v), v\}$ is a match set because of the tree $a(b, c)$. However, $M' = \{a(b, v), v\}$ is not a match set, because a match of $a(b, v)$ implies a match of $a(v, v)$ at the same node. □

Observe that the set of all possible match sets contains all sets which the bottom-up matching algorithm could assign (in encoded form) in any subject tree, given the pattern forest F.

Given F, let Match(t) denote the match set which must be assigned at the root of the subject tree t. PF is the set of all pattern subtrees from F. We can now formally state the two properties on which the bottom-up matching algorithm is based.

Definition 4.2

(1) If a is a nullary symbol, then

$$\text{Match}(a) = \begin{cases} \{a, v\} & \text{if } a \text{ is in PF,} \\ \{v\} & \text{otherwise.} \end{cases}$$

(2) If a is q-ary, $a > 0$, then

$$\text{Match}(a(t_1, \ldots, t_q)) = \{v\} \cup \{p' \mid p' \text{ has root } a \text{ and is in PF, and for } 1 \le j \le q, \text{son}_j(p) \text{ is in Match}(t_j)\}.$$

Note that because of (2), Match(t) does not depend on any node in t whose distance from the root exceeds the maximum height of a pattern. Because of this and the manner in which codes are assigned, the bottom-up matching algorithm responds well to local changes in a subject tree. See [15] for details.

In principle, the required enumeration of sets and tables may be generated by a simple closure strategy which starts with Match(a) for all nullary symbols a and repeatedly closes under the operation (2) of Definition 4.2. Such an algorithm would require

$$O(set^{(rank+1)} \times sym \times patsize)$$

time, where *set* is the number of distinct match sets generated. The table size would be $O(set^{rank} \times sym)$. In order to improve this time limit and to bound the size of *set*, which could be as bad as $O(2^{patsize})$, we need to understand certain relations between patterns and members of match sets. We define the following relations on tree patterns.

Definition 4.3. Let p and p' be patterns in S_v. Then p is *inconsistent* with p' (written $p \| p'$) if there is no subject tree t in S with both p and p' in Match(t). p and p' are *independent* (written $p \sim p$) if there are trees t_1, t_2, t_3 in S such that p is in Match(t_1), p' is not in Match(t_1), p is not in Match(t_2), p' is in Match(t_2), and p and p' are in Match(t_3). p *subsumes* p' ($p \ge p'$) if, for all t in S, p in Match(t) implies that p' is in Match(t). p *strictly* subsumes p' ($p > p'$) if $p \ge p'$ and $p \ne p'$. $p < p'$ iff $p' > p$.

Example 4.2. $a(b, v) \| a(c, v)$, since b and c cannot both be matched in the same position. $a(b, v) \sim a(v, c)$, since $a(b, v)$ in Match($a(b, b)$), $a(v, c)$ not in Match($a(b, b)$); $a(b, v)$ not in Match($a(c, c)$), $a(v, c)$ in Match($a(c, c)$); $a(b, v)$ in Match($a(b, c)$), $a(v, c)$ in Match($a(b, c)$). Finally, $a(b, v) > a(v, v)$. □

Figure 5

Given distinct patterns p and p', exactly one of the relations $\|$, $\sim$, $>$, and $<$ must hold between p and p'. The elementary properties of the three relations are summarized below. Note that in the absence of variables distinct patterns must be inconsistent.

PROPOSITION 4.1. *For trees p_1, p_2, p_3 in S_v:*

(a) $p_1 > p_2$ *and* $p_2 > p_3$ *implies* $p_1 > p_3$;
(b) $p_1 \| p_2$ *iff* $p_2 \| p_1$;
(c) $p_1 \sim p_2$ *iff* $p_2 \sim p_1$;
(d) $p_1 \| p_2$ *and* $p_3 > p_2$ *implies* $p_1 \| p_3$;
(e) $p_1 \sim p_2$ *and* $p_2 > p_3$ *implies* $p_1 \sim p_3$ *or* $p_1 > p_3$.

Recall that M' of Example 4.1 is not a match set because $a(b, v)$ subsumes $a(v, v)$. The inclusion of one pattern (e.g., $a(v, v)$) in M may be the consequence of the presence of another pattern which subsumes it (e.g., $a(b, v)$). Therefore, there may be a subset of patterns in M which completely determines M. We partition each match set M into a set M_0 of pairwise independent trees and a set M_1 of trees subsumed by some tree in M_0. M_0 is called the *base* of M.

PROPOSITION 4.2. *Given a pattern forest F and match set M for F, there is a unique partition of M into sets M_0 and M_1 such that for distinct p_1, p_2 in M_0, $p_1 \sim p_2$ holds, and for each p' in M_1 there is a p in M_0 such that $p > p'$.*

Observe that different match sets must have different base sets, owing to Proposition 4.1a. Thus we may represent match sets by their base sets.

Definition 4.4. Given a pattern forest F, the *independence graph* G_I of F is as follows: The vertices of G_I are distinct trees in PF. There is an undirected edge between p and p' iff $p \sim p'$.

Example 4.3. Consider the pattern forest $F = \{p_1, p_2, p_3\}$, where $p_1 = a(b(b(v)), v)$, $p_2 = a(b(v), b(v))$, and $p_3 = a(v, b(b(v)))$. There are three additional trees in PF: $p_4 = b(b(v))$, $p_5 = b(v)$, and $p_6 = v$. Since the trees p_1, p_2, p_3 are pairwise independent, whereas no other tree pairs are, the independence graph G_I of F is as shown in Figure 5, with a connected component p_1, p_2, p_3 and three isolated points. □

From the independence graph we can derive an upper bound on the number of possible match sets of a given pattern forest.

THEOREM 4.3. *The number of possible match sets of a pattern forest F is at most the number of cliques in the independence graph G_I of F, counting all subcliques, including the trivial ones.*

This theorem follows easily from Proposition 4.2. To illustrate it, consider F of Example 4.3. The theorem would limit the number of match sets of F to ten, for G_I has six trivial cliques, three cliques of size 2, and one clique of size 3. We would thus expect six match sets with a base set of a singleton, three match sets with base sets consisting of two trees each, and one match set with a base set of three elements. However, in this example there is no match set with the base $\{p_1, p_3\}$, since matching

both p_1 and p_3 at the root implies that p_2 matches at the root as well. Thus Theorem 4.3 gives an upper bound only. For deriving exact limits we would need to introduce other structural properties and analyze relations between more than two patterns.

For certain pattern forests the graphs G_1 could be such that the number of cliques grows exponentially with the number of subtrees in F and hence exponentially with the size of F. In such cases the number of distinct match sets may also grow exponentially.

THEOREM 4.4. *There are classes of pattern forests for which the number of distinct match sets grows exponentially with the size of the forest.*

PROOF. We define a class of balanced binary trees p_j^i, $0 \le i$, $0 \le j \le 2^i$, of height i, with all interior nodes labeled a. In p_j^i, all leaves are labeled v except the jth leaf from the left, which is labeled b. For $j = 0$, all leaves are labeled v.

$$\begin{aligned} p_0^0 &= v, \\ p_1^0 &= b, \\ p_j^{i+1} &= a(p_j^i, p_0^i), \qquad 0 \le j \le 2^i, \\ p_j^{i+1} &= a(p_0^i, p_{j-2^i}^i), \qquad 2^i < j \le 2^{i+1}. \end{aligned}$$

Define the pattern forest $F_n = \{p_i^n \mid 1 \le i \le 2^n\}$. The *size* of F_n is $O(2^n)$. Furthermore, $p_i^n \sim p_j^n$ for distinct nonzero values of i and j. Now consider sets Q of integers between 1 and 2^n, and define for each such set Q a balanced binary tree P_Q of height n with all interior nodes labeled a and such that the ith leaf from the left is labeled b if i is in Q, c otherwise. Then p_i^n matches P_Q at the root iff i is in Q. There are 2^{2^n} such sets Q; thus there must be at least as many different match sets. □

As a consequence of Theorem 4.4, a preprocessing algorithm based on computing tables indexed by match sets to drive the bottom-up matching algorithm must be impractical in certain cases. Since independence among subpatterns in a forest is responsible for a possible exponential growth of the number of match sets, we conclude the section with a necessary condition for independence based on the structure of patterns.

PROPOSITION 4.5. *Let p, p' be independent patterns. Then p contains disjoint subtrees t_1 and t_2 and p' contains disjoint subtrees t_1' and t_2', in corresponding positions, such that $t_1 > t_1'$ and $t_2' < t_2$.*

PROOF. Since v and nullary symbols in corresponding positions cannot be independent of other patterns, we may assume that

$$\begin{aligned} p &= a(p_1, \ldots, p_q), \\ p' &= a(p_1', \ldots, p_q'). \end{aligned}$$

The proof is by induction on the height of p.

Basis. If p has height 1, then the p_i have height 0, thus are nullary symbols or v, and thus, for $1 \le i \le q$, $p_i \ge p_i'$ or $p_i' \ge p_i$. If, for all i, $p_i \ge p_i'$ ($p_i' \ge p_i$), then $p \ge p'$ ($p' \ge p$). But $p \sim p'$ by assumption, and thus we can find the required trees among p_i and p_i'.

Induction Steps. Assume that the proposition holds for all p of height less than h, and assume that p has height h. Surely $p_i \| p_i'$ cannot hold; otherwise p and p' would be inconsistent. If there is some i such that $p_i \sim p_i'$, then apply the induction hypothesis to p_i and p_i'. Otherwise, for all i, $p_i \ge p_i'$ or $p_i' \ge p_i$, and the argument of the induction basis completes the proof. □

Note that mutual subsumption, in opposite directions, of disjoint subtrees is necessary but not sufficient for independence, since it does not rule out the possibility that other subtrees are inconsistent. For example, $a(b, v, c)$ and $a(v, b, d)$ are inconsistent, yet there are disjoint subtree pairs satisfying the "only if" condition of Proposition 4.5.

Proposition 4.5 is used when testing the restrictions imposed on tree patterns in the next section.

We have recently learned that the idea of bottom-up tree pattern matching was discovered independently by Kron [23]. He calls match sets "batches" and defines the relations $>$, $\|$, $\sim$ (which he calls "more specific than," "not overlapping," and "intersecting," respectively) equivalently by containment and intersection properties of the sets of Σ-terms which two patterns match at the root.

He matches patterns in a subject tree using an automaton as well. Instead of using matrices as tables, however, he computes the match set to be assigned to node n with q sons by a subautomaton which, in q transition steps reading the match set codes of the sons, determines the code for the new match set. There is one subautomaton per alphabet symbol. As a result, his match time is $O(subsize)$. One can visualize each subautomaton as a trie encoding of one of our matrices. Depending on the pattern structure, this leads to smaller space requirements in certain cases.

The preprocessing of Kron is essentially the method sketched in the paragraphs following Definition 4.2. Because of Theorem 4.4, this preprocessing takes time exponential in the pattern size in the worst case. As Kron tells us, he was aware of this, but it was not a concern of his research in [23]. We are going further and analyzing match sets seeking a definition of a subclass of tree patterns with polynomial preprocessing time. We give such a definition in the following section.

Preprocessing in Kron's sense has been used in practical situations by Wilhelm [10]. Since this work seems to accomplish practically viable preprocessing times, we conclude that the exponential worst case of bottom-up matching does not arise frequently in these applications.

5. *Simple Pattern Forests*

Because of the exponential growth of the number of match sets for certain pattern forests (Theorem 4.4), we wish to restrict patterns when generating tables to drive the bottom-up matching algorithm of Section 3. Theorem 4.3 suggests disallowing independence among pattern subtrees. This restriction is not as drastic as it might seem and has not seriously hindered us when generating interpreters for LISP, LUCID, and the Combinator Calculus using these techniques [14].

Definition 5.1. A pattern forest F is *simple* if it contains no independent subtrees.

For simple forests, the independence graph has no edges; hence, by Theorem 4.3, the number of distinct match sets is at most the size of the forest. Furthermore, simple forests have a number of useful properties which can be exploited in the design of efficient matching algorithms.

Definition 5.2. If F is a pattern forest, and p, p' are subpatterns in PF, then p *immediately* subsumes p', $p >_i p'$, iff $p > p'$ and there is no other subpattern p'' in PF such that $p > p''$ and $p'' > p'$. Immediate subsumption is the transitive reduction of subsumption on the set of all subpatterns of F.

Definition 5.3. The *immediate subsumption graph* G_S of the forest F has as vertices all distinct subpatterns in F. There is a directed edge from p to p' iff $p >_i p'$. In general, G_S is a directed acyclic graph with v as the only leaf.

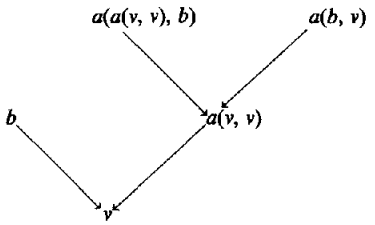


FIG 6 The immediate subsumption graph of F.

LEMMA 5.1. *The immediate subsumption graph G_S of a simple forest F is an inverted tree with v as root.*

PROOF. Let p, p', and p'' be distinct subtrees in F, and assume that p subsumes both p' and p'', but neither $p > p''$ nor $p'' > p'$. Since p subsumes both trees, $p' \| p''$ is impossible (Proposition 4.1d); hence p' and p'' must be independent. But then F cannot be simple. Hence either $p' > p''$ or $p'' > p'$. □

Observe that for simple forests, the base set M_0 of any match set must be a singleton. Using Lemma 5.1 and Proposition 4.2, we thus easily obtain

THEOREM 5.2. *Let F be a simple forest and M any match set for F with base set $\{p\}$. Then M consists precisely of the trees encountered on the path from p to v in G_S.*

This theorem is the central result for simple forests. It frees us from having to construct explicitly the individual match sets, for G_S provides them at once along with their structure and interrelation. We conclude the section with an example illustrating Theorem 5.2, and a discussion of the relationship between G_S and the failure function f constructed in the algorithm for string pattern matching in [1, 21].

Example 5.1. The pattern forest $F = \{a(a(v, v), b), a(b, v)\}$ is simple, since there are no independent trees or subtrees. Its immediate subsumption relation is

$$b >_i v, \qquad a(v, v) >_i v,$$
$$a(b, v) >_i a(v, v), \qquad a(a(v, v), b) >_i a(v, v),$$

which has the graph G_S shown in Figure 6. From this graph we then obtain as possible match sets the five sets of Example 3.1:

$$\{v\},$$
$$\{b, v\},$$
$$\{a(v, v), v\},$$
$$\{a(b, v), a(v, v), v\},$$
$$\{a(a(v, v), b), a(v, v), v\}.$$

Note the correspondence of these sets to the paths in G_S. □

There is a connection between the immediate subsumption graph G_S and the failure function f used in string-pattern-matching algorithms in [1, 21]. This connection is observed by visualizing a string pattern $a_1a_2 \cdots a_m$ as the nonbranching tree $a_m(\ldots a_2(a_1(v))\ldots)$. Note the reversal of the character sequence. The addition of v as a leaf permits us to conceptualize the a_i as symbols of arity 1 and permits sliding the nonbranching tree in the subject. Matching this pattern in the subject $b_1b_2 \cdots b_n$ is now equivalent to matching the nonbranching tree pattern in the tree $b_n(\ldots b_2(b_1(c))\ldots)$, where c is a new nullary symbol. Having translated the string-matching problem into a tree-matching problem in this way, we now observe that G_S is just the graph of the failure function f constructed for the original string problem by the algorithms in [1, 21]. To observe this, note that a subtree corresponds to a

pattern prefix, and that $p > p'$ iff p' is a pattern prefix which matches, as suffix, in the pattern prefix p. Hence $p >_1 p'$ iff p' is the longest proper prefix of p which matches, as suffix, in the prefix p, which is just the definition of the failure function.

Note also that because of Proposition 4.5, pattern forests derived from string patterns must be simple, because nonbranching trees cannot have disjoint subtrees. Hence there is no counterpart in string matching to the exponential explosion of match sets, which can occur for nonsimple forests in tree matching.

6. *Table Construction for Simple Forests*

For a simple pattern forest F, the tables to drive the bottom-up algorithm of Section 3 may be constructed in two steps. First, construct the subsumption graph $\bar{G}_S$ whose vertices are the trees in PF. $\bar{G}_S$ has a directed edge from p to p' iff $p \geq p'$. Observe that this is equivalent to finding all match sets which can occur when matching in any subject. Then, for each alphabet symbol a of arity m, we use $\bar{G}_S$ to construct a table T_a such that $T_a[p_1, \ldots, p_m]$ is the match-set code which should be assigned to any node labeled a at whose sons we have assigned the match-set codes p_1 to p_m from left to right, respectively.

We find it convenient to represent a match set M by its base set tree, that is, by the largest (in the sense of $>$) tree in M. This is a reasonable choice since, by Proposition 4.2 and Theorem 5.2, the largest tree in M completely determines M. The advantages of this coding is that we can now define the entry $T_a[p_1, \ldots, p_m]$ as the largest tree in PF subsumed by $a(p_1, \ldots, p_m)$, because of observation (2) below. Note that the tree $a(p_1, \ldots, p_m)$ need not occur in PF.

To construct $\bar{G}_S$, observe that for distinct patterns p, p',

(1) If $p > p'$, then height$(p) \geq$ height(p').
(2) Let $p = a(p_1, \ldots, p_m)$. Then $p > p'$ iff either $p' = v$ or $p' = a(p'_1, \ldots, p'_m)$, where $p_j \geq p'_j$ for $1 \leq j \leq m$.

So we may process patterns in order of increasing height and compare each pattern to all patterns of no greater height using observation (2). Since the subpatterns p_i and p'_i in (2) above are of strictly smaller height than p and p', respectively, $p_j \geq p'_j$ has already been checked by the time p is compared to p'.

Algorithm A

Input: Simple pattern forest F.

Output: Subsumption graph $\bar{G}_S$ for F.

Method:

1. List the trees in PF by increasing height
2. Initialize $\bar{G}_S$ to the graph with vertices PF and no edges.
3. **For each** $p = a(p_1, \ldots, p_m)$, $m \geq 0$, of height h, by increasing order of height, **do**
4. **for each** p' in PF of height $\leq h$ **do**
5. **If** $p' = v$ **or**
   $p' = a(p'_1, \ldots, p'_m)$ where, for $1 \leq i \leq m$, $p_i \to p'_i$ is in $\bar{G}_S$,
   then
6. Add $p \to p'$ to $\bar{G}_S$.

For the analysis of Algorithm A, observe that step 1 requires $O(patsize)$ time using bucketsort. Steps 3–6 require $O(patsize^2 \times rank)$ steps, assuming that $\bar{G}_S$ is stored as an adjacency matrix, so that checking whether $p_j \to p'_j$ requires constant time. The space complexity is dominated by the $O(patsize^2)$ adjacency matrix. Thus Algorithm A requires $O(patsize^2 \times rank)$ steps and $O(patsize^2)$ space.

To generate the table T_a, recall that for the m-ary symbol a and trees $p_1, \ldots, p_m$ in PF, $T_a[p_1, \ldots, p_m] = p$, where p is the largest (in the sense of $>$) tree in PF such that $a(p_1, \ldots, p_m) \geq p$. This can be seen as follows. If $a(p_1, \ldots, p_m) \geq t$, then either $t = v$ or $t = a(p'_1, \ldots, p'_m)$ and, for $1 \leq i \leq m$, $p_i \geq p'_i$. Then the set

$$M = \{t \text{ in PF} \mid a(p_1, \ldots, p_m) \geq t\}$$

is precisely the match set which should be coded by the entry $T_a[p_1, \ldots, p_m]$, assuming p_i codes the match set with base set tree p_i. Recall that by Lemma 5.1 subsumption induces a total order on the elements of M; hence the largest tree p in PF subsumed by $a(p_1, \ldots, p_m)$ is precisely the base set tree of M and thus the code which should be assigned to $T_a[p_1, \ldots, p_m]$.

Now observe that by (2), $a(p_1, \ldots, p_m) > p$ is easily testable from $\bar{G}_S$. Furthermore, if we process the patterns in PF in increasing order of subsumption and for each p in PF assign p to all of the entries $T_a[p_1, \ldots, p_m]$ such that $a(p_1, \ldots, p_m) \geq p$, then the last assignment made to the entry will be the maximal subsumed p in PF. Thus, if we write each p into the appropriate table positions when p is processed, the final values in the table are the correct ones.

Algorithm B

Input. $\bar{G}_S$ for a simple pattern forest F

Output. Tables to drive the bottom-up matching algorithm $T_a[p_1, \ldots, p_m]$ will contain the largest (under subsumption) tree in PF which is subsumed by $a(p_1, \ldots, p_m)$

Method.

1 List PF in increasing order of subsumption by performing a topological sort on $\bar{G}_S$
2 Initialize all entries in all tables T_a to v
3. **For each** pattern $p = a(p_1, \ldots, p_m)$ by increasing order of subsumption **do**
4 **For each** m-tuple $\langle p'_1, \ldots, p'_m \rangle$ such that, for $1 \leq j \leq m$, $p'_j \geq p_j$ **do**
5 $T_a[p'_1, \ldots, p'_m] := p$.

The table for the symbol a of arity q has $patsize^q$ entries. Thus Algorithm B constructs no more than $patsize^{rank} \times sym$ entries. When a tree p is assigned to an entry in T_a, then p belongs to the match set which should be coded by this entry. Thus the number of repeated assignments to each entry cannot exceed the size of the largest match set, that is, the height of $\bar{G}_S$. Thus at most $patsize^{rank} \times sym \times ht$ assignments are done in step 5.

Note that p'_i ranges over those trees in PF such that $p'_i \to p_i$. Hence we can find the necessary tuples easily from the adjacency matrix of $\bar{G}_S$. In an implementation of this algorithm the patterns in PF are numbered, and the tables T_a are indexed by these numbers. We summarize the complexity of preprocessing patterns in simple forests by the following.

THEOREM 6.1. *We can construct tables to drive the bottom-up matching algorithm in the case of simple pattern forest in*

$$O(patsize^2 \times rank + patsize^{rank} \times ht \times sym)$$

time and

$$O(patsize^2 + sym + patsize^{rank})$$

space.

Note that it is easy to test whether a pattern forest is simple. Using Proposition 4.6, it suffices to test, in step 5 of Algorithm A, whether p and p' contain two (immediate)

TABLE I TABLE T_a GENERATED FOR THE SYMBOL a

Left subtree match	Right subtree match				
	v	b	$a(v, v)$	$a(b, v)$	$a(a(v, v), b)$
v	$a(v, v)$	$a(v, v)$	$a(v, v)$	$a(v, v)$	$a(v, v)$
b	$a(b, v)$	$a(b, v)$	$a(b, v)$	$a(b, v)$	$a(b, v)$
$a(v, v)$	$a(v, v)$	$a(a(v, v), b)$	$a(v, v)$	$a(v, v)$	$a(v, v)$
$a(b, v)$	$a(v, v)$	$a(a(v, v), b)$	$a(v, v)$	$a(v, v)$	$a(v, v)$
$a(a(v, v), b)$	$a(v, v)$	$a(a(v, v), b)$	$a(v, v)$	$a(v, v)$	$a(v, v)$

subtrees in corresponding positions which subsume each other in opposite directions. If such a pair exists, then the pattern forest is not simple.

Example 6.1. We illustrate Algorithm B with the table T_a generated for the symbol a, given the pattern forest of Example 5.1. The table is essentially that of Example 3.1; however, for readability we represent entries and index values by trees, rather than enumerating them.

In this example, all table entries are assigned by step 5, so none of them is v. Consider $p = a(a(v, v), b)$ in the traversal of step 3. The m-tuples of steps 4 and 5 now range over the sets p'_1 in $\{a(v, v), a(a(v, v), b), a(b, v)\}$, since $a(a(v, v), b)$ and $a(b, v)$ are the two trees subsuming $a(v, v)$, and p'_2 in $\{b\}$, since there is no other tree subsuming b. So $a(a(v, v), b)$ is entered in $T_a[a(v, v), b]$, $T_a[a(a(v, v), b), b]$, and $T_a[a(b, v), b]$. The entry $T_a[a(v, v), b]$ had already been assigned the smaller pattern $a(v, v)$, since $a(v, v) > v$ and $b > v$, but this entry is wiped out by $a(a(v, v), b)$ at this time. Table I shows the table T_a. □

Clearly Algorithm B constitutes the bottleneck of preprocessing, both in space and in time requirements. Often the situation can be improved by introducing one or more pairing functions, thereby reducing *rank* to 2. Although pairing is always possible, it need not preserve simplicity of the forest and is thus of limited value.

Example 6.2. Consider the pattern forest $\{a(b, v, c), a(v, b, d), a(e, c, v)\}$. All subtrees other than v are pairwise inconsistent, and thus the forest is simple. Introducing a pairing function, no matter which subtrees are paired, will introduce independence. For example, pairing the first and second subtree results in a new forest $\{a'(\text{pair}(b, v), c), a'(\text{pair}(v, b), d), a'(\text{pair}(e, c), v)\}$ in which $\text{pair}(b, v)$ and $\text{pair}(v, b)$ are independent subtrees. □

There is a different approach to speeding up preprocessing. Recall that G_S generalizes the failure function of string matching. We suspect that there is an efficient bottom-up matching algorithm using G_S directly, without any tables. So far we have only achieved a running time of

$$O(subsize \times patsize \times ht)$$

by this approach, which is inferior to the naive method.

7. *Faster Preprocessing for Binary Simple Forests*

Algorithm A is quadratic in *patsize* since it constructs $\bar{G}_S$, the transitive closure of G_S, rather than G_S. It seems there should be an algorithm for computing G_S for simple pattern forests which requires $O(patsize)$ steps only. So far, we have not found an algorithm this efficient, but in the special case of binary simple pattern forests we can construct G_S in $O(patsize \times ht^2)$ steps. Here ht may be as large as *patsize*, but it is usually much smaller. Given the algorithm for computing G_S, it is then possible to

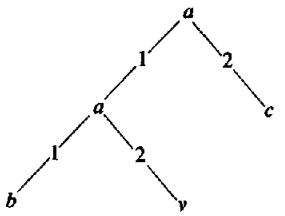


Figure 7

adapt it to do the pattern matching as well, bypassing the expensive step of table generation. We sketch the idea of this algorithm next.

Recall that in a simple forest F, for each subpattern p in PF there is exactly one largest subsumed subpattern p' in PF, except when $p = v$. Let $f(p)$ denote this tree p', that is, the tree immediately subsumed by p. Denote the *ith iterate* of f by $f^i(p)$, $0 \leq i$, where

$$f^0(p) = p,$$
$$f^{i+1}(p) = f(f^i(p)).$$

Note that G_S is the graph of the function f.

Consider computing $f(p)$, where the root of p is a binary symbol, that is, $p = a(p_1, p_2)$. We should examine trees of the form $a(f^i(p_1), f^j(p_2))$, $i + j > 0$, as possible candidates for $f(p)$. For this purpose we will maintain sets $S(a, p_1)$, where a is in Σ and p_1 is a pattern subtree. Each set contains pairs $\langle p_2, p \rangle$ of subpatterns. The pair $\langle p_2, p \rangle$ is in $S(a, p_1)$ iff $p = a(p_1, p_2)$ is in PF. In computing $f(p)$ we now probe in the sets $S(a, p_1)$, $S(a, f(p_1))$, $S(a, f^2(p_1))$, ... for pairs whose first component is p_2, $f(p_2)$, etc. The first such pair found (other than the pair $\langle p_2, p \rangle$ in $S(a, p_1)$) must be $f(p)$, since F is a simple forest. We make at most $O(ht^2)$ probes, since $f^{ht}(t) = v$, for any subpattern.

We can make a single probe efficiently by representing the set $S(a, p_1)$ by an array in which the second component of a pair is stored as the element indexed by the first component. In order to avoid an $O(patsize^2)$ overhead for initializing all vectors, we use the constant time array initialization of [2, Ex. 2.12]. The running time of the algorithm is thus $O(patsize \times ht^2)$.

Observe that the algorithm can be adapted to do the matching using the sets $S(a, p_1)$ without using the table generation (Algorithm B). This leads to a matching algorithm which requires at most $O(subsize \times ht^2)$ steps.

8. *Top-Down Matching Algorithm*

Like the bottom-up matching algorithm, our top-down matching algorithm is related to the Knuth–Morris–Pratt string-matching algorithm. Instead of generalizing string matching, however, the top-down approach reduces tree matching to string matching. The top-down method has slower matching time than the bottom-up, but better preprocessing time.

The key idea of reducing tree pattern matching to string matching is to regard each path from root to leaf in a tree as a string in which symbols in the alphabet are interleaved with numbers indicating which branch from father to son has been followed. Since variables always match, we do not include them in these strings.

Example 8.1. The tree pattern $a(a(b, v), c)$ is associated with the set of strings $\{a1a1b, a1a2, a2c\}$. Note that we have omitted the symbol v from the end of the second string. Figure 7 shows how the set of strings appears in the given tree. □

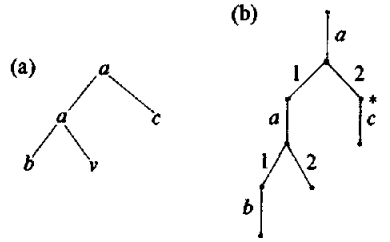


FIG. 8 (a) Tree pattern (b) Associated trie

This idea was first noticed by Karp et al. [18] and used in a tree-matching algorithm with no preprocessing. Their algorithm achieved a matching time of

$$O((patsize + subsize) \times \log(patsize))$$

for one pattern, which must be a full binary tree. For several patterns their algorithm would require

$$O((patsize + subsize) \times \log(patsize) \times patno).$$

Our contribution is to show how, using the Knuth–Morris–Pratt algorithm for string matching, we can improve the bounds to $O(patsize)$ preprocessing, plus $O(subsize \times patno)$ for matching, in the case of patterns which are full trees. If the patterns are not full trees, more time for matching is needed. We thus improve the bound of Karp et al. by a factor of $\log(patsize)$.

For simplicity of presentation we develop our results for the case of a single tree pattern first. Given the pattern p, it is easy to generate all path strings for the root-to-leaf paths. We could then use the algorithm of Aho and Corasick [1] to produce an automaton which recognizes every instance of a path string within a subject tree. Since the combined length of all strings could be $O(patsize^2)$, we need to modify this construction so as to avoid generating the strings explicitly. In this way we can lower the preprocessing to $O(patsize)$.

The first step in the Aho–Corasick algorithm is to build a trie for the path strings of the tree pattern p. This trie is called the "goto function" in [1]. A trie is a tree whose nodes represent the distinct prefixes of the path strings. If node n represents x and n' represents xa, a in $\Sigma \cup N$, then n is father of n', and the edge from n to n' is labeled a. We illustrate the construction with an example. Since it amounts to a simple tree transformation, we do not formally give an algorithm.

Example 8.2. The pattern tree $a(a(b, v), c)$ has the associated trie shown in Figure 8. For example, the marked node represents the prefix $a2$. □

Informally, the trie is constructed by first enumerating the outedges of every pattern node and then splitting every node labeled with a symbol other than v into two nodes connected by an edge which is labeled with the original node label.

The subsequent steps in constructing a matching automaton are exactly as in [1], for we are now dealing with a string problem. Thus the entire construction requires $O(patsize)$ steps if we use a failure-function representation of the automaton and $O(patsize \times sym)$ if we use a transition-matrix representation.

We need to include in this construction a simple modification which records, with each accepting state of the automaton, the length(s) of the accepted string(s). The length of a path string is the number of alphabet symbols in it (numbers are ignored). Thus the length for $a2c$ and $a1a2$ is 2 in both cases.

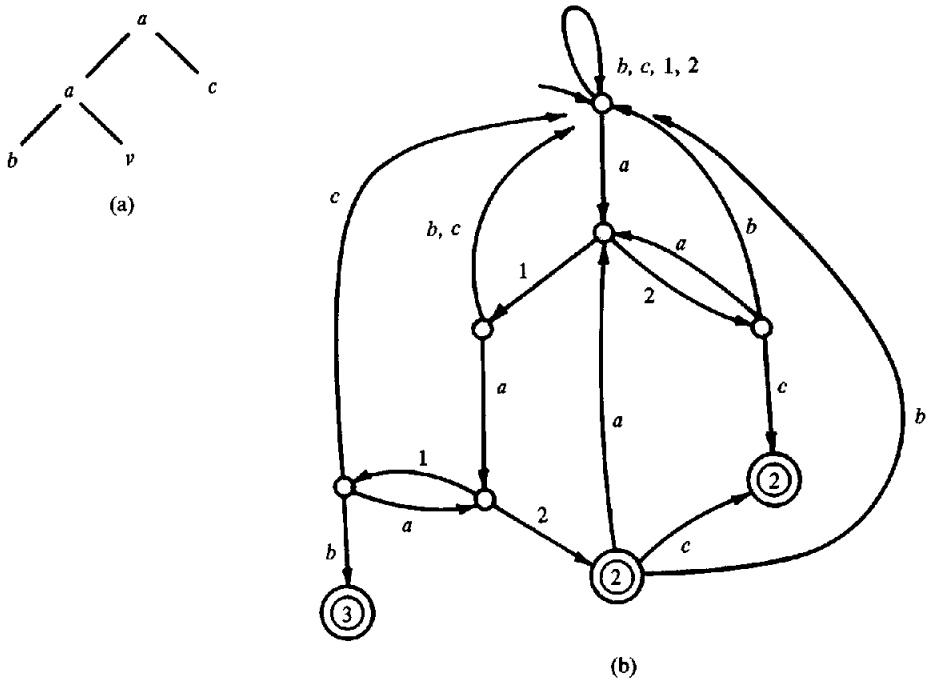


FIG. 9 (a) Pattern (b) Matching automaton.

Example 8.3. In Figure 9 we give the automaton associated with the pattern of the previous example. Accepting states are circled twice and are labeled with the length of the accepted path string. □

We now have to solve the problem of how the matching algorithm can decide whether two different path strings begin at the same node and thus contribute to a pattern match at that node. For this purpose we associate with each node a counter, initialized to zero. Each counter will record the number of distinct root-to-leaf paths which match beginning at that node.

Let us traverse the subject tree t in preorder, computing the automaton states as we visit nodes and traverse edges. For recovering former states when returning from a completely traversed subtree we can use the traversal stack. Every time the matching automaton enters a final state, we have matched one or more path strings, and we should indicate this fact at the points at which the matched paths begin. So we increment the counters of those nodes by 1. The traversal stack for the preorder traversal is kept in an array. Thus we can find the beginning node of a matched path string in the traversal stack and can access it in constant time once we know the length of the matched string.

At the end of the traversal the pattern matches at each node whose counter equals the number of leaves in the pattern (i.e., the number of path strings). We can now give the matching algorithm.

We will use an array of triples $\langle n, s, j \rangle$ as traversal stack, where n is a node in the subject tree, s the state the automaton has entered when the traversal visits n, and j a number indicating how many sons of n have been visited. Additionally, we have an array *Count*, indexed by nodes n of the subject tree, which contains the associated counters.

We assume that the algorithm uses a transition-table representation of the automaton and indicate by $A[s, c]$ the state the automaton enters when in state s reading symbol c.

We use a procedure *Tabulate*, which maintains the counters and updates the list of matches found. This procedure can access the stack of triples.

Algorithm D (Top Down Matching)

Input A string matching automaton for tree pattern p in transition matrix representation, and a subject tree t.

Output. A list, Match, of all nodes in t at which p matches.

Comment. $A[s, c]$ is the state entered from s under input c in the matching automaton.
Stack[t].i denotes the ith component of the triple stacked at position t in the array Stack
son$_i(n)$ denotes the ith son of tree node n

Method:

```
 1  Match := empty,
 2  For all nodes n in t do Count[n] = 0,
 3  Nextstate = A[start state, label(root of t)];
 4. Top := 1,
 5  Stack[Top] = ⟨root of t, Nextstate, 0⟩,
 6. Tabulate(Nextstate),
 7. While Top > 0 do begin
 8       ⟨Thisnode, Thisstate, Nsons⟩ := Stack[Top],
 9.      If Nsons = arity(Thisnode) then Top = Top − 1,
10.          else begin
11               Nsons := Nsons + 1,
12.              Stack[Top].3 := Nsons;
13.              Intstate = A[Thisstate, Nsons],
14.              Tabulate(Intstate);
15.              Nextnode = son_Nsons(Thisnode),
16.              Nextstate := A[Intstate, label(Nextnode)],
17.              Top := Top + 1,
18.              Stack[Top] = ⟨Nextnode, Nextstate, 0⟩,
19.              Tabulate(Nextstate),
20.          end (if)
21       end (while)
```

```
Procedure Tabulate (State)
1. For all s such that State has a match of length s
2.      do begin
3           n = Stack[Top − s + 1].1;
3           Count[n] := Count[n] + 1,
4           If Count[n] = number of leaves in pattern then
5               Add n to Match,
6       end (for)
```

Except for the work of procedure Tabulate, the complexity of Algorithm D is $O(subsize)$, since each edge is traversed at most twice. This is also true for the failure-function representation of the matching automaton (see [1]). The total work of procedure Tabulate is proportional to the number of times any counter has been incremented, or equivalently, to the sum of all counter values upon completion of the traversal. We can estimate this sum by deriving a bound on the number of different counters which can be incremented in an accepting state, for this will also bound the work done for each call of the procedure.

Definition 8.1. Given a tree pattern p and a path string s of p, the *suffix number* of s is the number of path strings s of p which are suffixes of s, including p itself. The *suffix index* of p is the maximum suffix number of the path strings of p.

Equivalently, the suffix index is the largest number of counters which could be incremented in any accept state of the automaton.

Example 8.4. For the pattern $p = a(a(a(v, b), c), b)$ we have the path strings $a1a1a1$, $a1a1a2b$, $a1a2c$, $a2b$. The suffix number of $a1a1a1$ is 1, whereas the suffix number of $a1a1a2b$ is 2, since $a2b$ is a suffix which occurs as root to leaf path in p. The suffix index of p is also 2. □

THEOREM 8.1. *Algorithm D requires $O(subsize \times suf)$ steps, where suf is the suffix index of the pattern to be matched.*

For patterns which are full trees, that is, all path strings are of equal length, *suf* must be 1, since a distinct path string s_1 can be a proper suffix of a distinct path string s_2 only if s_1 is shorter than s_2. This gives us

COROLLARY 8.2. *If Algorithm D matches a pattern which is a full tree, then only $O(subsize)$ steps are needed.*

In the worst case, *suf* could be $O(patsize)$.

Example 8.5. Consider the pattern,

$$p_k = \underbrace{a(a(\ldots a}_{k \text{ times}}(v, b) \ldots b), b).$$

Its suffix index is k, owing to the path string $(a1)^{k-1}a2b$, which has every shorter path string as suffix. Note that *patsize* is $2k + 1$. □

COROLLARY 8.3. *The bound of $O(subsize \times patsize)$ for Algorithm D is attained for certain patterns.*

PROOF. Consider matching the pattern p_k of Example 8.5 in the subject,

$$t_n = \underbrace{a(a(\ldots a}_{n \text{ times}}(c, b) \ldots b), b),$$

where $n = k + m$. Then the sum of the counter values in t_n after Algorithm D has finished exceeds $m \times k$. Note that *patsize* is $2k + 1$ and *subsize* is $2n + 1$. □

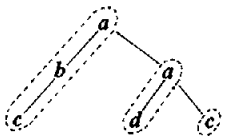
We thus have in Algorithm D a performance range anywhere between that of the bottom-up algorithm and that of the naive matching algorithm, depending on the structure of the pattern.

Without going into details we note that Algorithm D may be adapted to assimilate local changes in the subject tree. As in the case of the bottom-up algorithm, we need to reprocess only a small area surrounding the part which has changed. However, the algorithmic details are far more complicated than in the case of the bottom-up algorithm, although in principle quite straightforward.

We conclude this section with a brief discussion of how to match more than one tree pattern, using the approach of Algorithm D.

Recall that we represent a tree pattern by its root-to-leaf path strings. We can do this for several patterns as well, but we should keep track of which pattern(s) each path string comes from. The preprocessing algorithm can be adapted to process several patterns by building separately for each pattern the associated trie and then merging these tries, keeping track of which pattern(s) each path string at a leaf of the trie belongs to. This can be done in $O(patsize)$ steps resulting in a trie of $O(patsize)$ nodes. Now apply the methods of [1] to complete the trie to a matching automaton.

Figure 10



In the case of a single pattern we associated with each of the final states a list of the lengths of the matched path strings. For multiple patterns we now associate with final states lists of pairs. Each pair gives the length of the matched path string and the pattern to which it belongs.

It remains to explain how we can correlate matches of individual path strings. We do this simply by associating *patno* counters with each node in the subject tree and dedicating the ith counter to counting how many path strings of the ith pattern have been matched, beginning at that node. If the ith counter reaches a value equal to the number of leaves of the ith pattern, then we have just matched the ith pattern.

As before, the work is proportional to the subject size plus the sum of all counter values and can be estimated as

$$O(\mathit{subsize} \times \max(\mathit{suf}) \times \mathit{patno}),$$

where the maximum is taken over all tree patterns in the forest. This bound is easily shown to be the best possible, generalizing Corollary 8.2. Furthermore, if no path string is a suffix of another, then we have only $O(\mathit{subsize})$ steps for matching such a pattern forest.

9. *Improvements to Top-Down Matching and Related Work*

Recently, Lang et al. [24] improved Algorithm D by basing the matching of path strings on the Boyer–Moore algorithm [4]. Since the Boyer–Moore algorithm requires the ability to skip portions of the subject string, a different representation of trees is used: Trees are represented by ordered lists of *left paths.*

Example 9.1. For the tree $t = a(b(c), a(d, c))$ the list of left paths is $\langle abc, ad, c\rangle$, as shown in Figure 10. □

We can obtain left paths by first deleting from each path string the longest prefix ending with a branch number greater than 1 and then deleting the remaining branch numbers. Thus, from $a2a1d$ we obtain ad, and from $a2a2c$ we get c. The list of these left paths uniquely determines a binary tree. For alphabet symbols of arity higher than 2, additional information has to be given for each left path string.

The algorithm first preprocesses the list of left paths of the pattern, constructing a Boyer–Moore-type automaton for recognizing the first left path, combined with an Aho–Corasick-type automaton for recognizing the remaining left paths. A match of the remaining left paths is attempted only at places at which the first left path has been completely matched. Note that the advantages of the Boyer–Moore machine diminish as the number of different strings to be matched increases. See [8] for a discussion of this phenomenon. The subject tree is also represented as an ordered list of (linked) left paths, so that we can skip ahead for the Boyer–Moore matching technique.

A subtlety of the algorithm, when it is applied to trees of arity exceeding 2, arises from the fact that a match of the jth left path implies an update of the appropriate counter only if the counter has a specific value, because the left path may be descending from a node with more than two sons. For details see [24].

Lang et al. [24] implemented both their algorithm and our Algorithm D. First experiments seem to indicate a sublinear average matching time for their algorithm. The worst-case performance of their algorithm is the same as that of Algorithm D.

Overmars and van Leeuwen [27] have given algorithms to match *lexicographic* trees, that is, trees in which the branches rather than the nodes are labeled with symbols from an alphabet. They assume that the branches emanating from each node are ordered left to right by their labels and that no label occurs more than once. Lexicographic trees arise as tries.

Overmars and van Leeuwen consider matching a given lexicographic tree (the pattern) in a larger lexicographic tree (the subject). A match is an alignment of the pattern nodes with certain subject nodes. The alignment must respect the father–son relation in such a way that the branches emanating from a subject node are labeled with the same symbol as the corresponding pattern branches. Note that not all branches of a subject node need to be covered by corresponding pattern branches.

Their algorithms were discovered independently from our work. Their technique, like our Algorithm D, is based on Karp et al.'s idea of matching path strings. In the case of lexicographic trees, however, no branch numbers need to be interleaved in path strings. Overmars and van Leeuwen also use counters to coordinate the matches of path strings.

Their best algorithm does preprocessing of the pattern similar to ours, identifying for each path string the suffixes which are also path strings. They give the preprocessing in their own terminology, but it amounts essentially to the algorithms of [1]. Their best matching algorithm has the same worst-case time bound as our Algorithm D. Other algorithms given in [27] do little or no preprocessing of the pattern and have inferior bounds on the matching time.

We wish to stress that the approaches of Algorithm D, Overmars and van Leeuwen, and Lang et al. are inherently limited by using counters for deciding whether there is a match. As long as counters are used and incremented in steps of one up to the number of leaves of a pattern, a simple counting argument shows that the bound of Theorem 8.1 cannot be improved except by a constant factor. We see only two ways for improving this situation. Either means are found to increment counters in larger steps (or, equivalently, to smaller values) or a new method for coordinating path strings is used. The former would imply that recording of matches is delayed in some way. For the latter approach we can offer a solution which reduces the worst case bound to $O(subsize + match)$.

Assuming a machine model in which, in constant time, we can perform bit-string operations of union, intersection, and right shift by one position, we can improve Algorithm D as follows. We associate with each node n of the subject tree a bit string b_n in which the ith bit (from the right) is 1 iff every path from the ancestor of n at distance i, through n, to every descendant of n, has a prefix which is a path string of the pattern we wish to match. Note that we do not need to use bit strings longer than the height of the pattern. There is a match of the pattern at node n iff b_n has a 1 in the rightmost position.

Example 9.2. Consider the tree pattern $a(a(b(v), c), a(v, v))$. Assume we wish to match it in the subject fragment shown in Figure 11. We should assign the bit string 100 to node 3, since we have a match of the path string $a1a1b1$, and also to node 4, because of the path string $a1a2c$. Note that both path strings are of length 3. To node 5 the bit string 010 is assigned, because the two path strings a2a1 and $a2a2$ match, both of length 2. To node 2 we assign the bit string 010, since every path originating

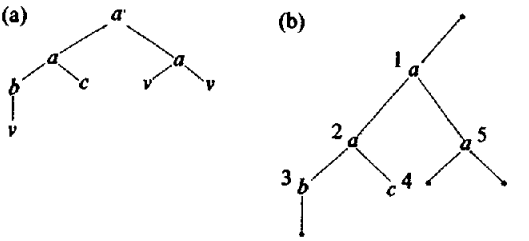


FIG 11 (a) Pattern (b) Subject.

at node 1, the ancestor of node 2 at distance 1, and going through node 2 has a prefix which is a path string in the pattern. Note that 010 can be obtained as the right shift by one of (100 ∩ 100), the intersection of the two bitstrings assigned to the two sons of node 2. Node 1 will be assigned the bitstring 001. The 1 at the extreme right signals the presence of a pattern match. □

Note that in this example bit strings of length 3 are used, since the length of the longest path string in the pattern is 3. We need to explain how these bit strings can be computed. During preprocessing we associate with each accepting state s a bit string b_s in which the ith bit is 1 iff a path string of length i is accepted. By carefully considering the techniques of [1] we can design this preprocessing step to require $O(patsize)$ time at most.

Traverse the subject tree in preorder as before. When reaching a node for the first time in the traversal, initialize b_n to b_s, where s is the corresponding state in the matching automaton. Then, when coming to n for the last time, that is, after all subtrees have been visited, update b_n by

$$b_n := b_n \cup \bigcap_j \text{rightshift}(b_{\text{son}_j(n)}),$$

where rightshift means a shift by one bit position to the right, introducing 0 on the left. This method then has, as worst case, $O(subsize + match)$ time requirement for matching, since we eliminated the work of procedure Tabulate.

Note that we need not associate bit strings with nodes permanently: Upon completing the traversal of a subtree rooted in n, the bit strings associated with the sons of n are no longer needed. Thus we may keep all bit strings in the traversal stack (plus *rank* additional cells). Similarly, we could have reduced the space requirements for Algorithm D by keeping the counters on the traversal stack.

10. *Bottom-Up Matching with Bit-String Operations*

Since most computers allow unions, intersections, and complements of sets represented as bit strings to be performed in a small fixed number of instructions, we explore the possibility of representing match sets by bit strings and computing them directly at match time, thus avoiding the costly table generation of Section 6.

Let F be a pattern forest and PF the set of all subpatterns in F.

Definition 10.1. Define the sets U_a for each a in the alphabet as follows;

$$U_a = \begin{cases} \{v\} & \text{if } a \text{ is nullary and not in PF,} \\ \{a, v\} & \text{if } a \text{ is nullary and in PF,} \\ \{t \text{ in PF} \mid t = a(t_1, \ldots, t_q)\} \cup \{v\} & \text{if } a \text{ is } q\text{-ary}, \quad q > 0. \end{cases}$$

Furthermore, define a set valued function on pattern sets by

$$\text{Father}_i(M) = \{t' \text{ in PF} \mid \text{son}_i(t') \text{ in } M\}.$$

We now recast Definition 4.2 as

Definition 10.2

(1) Match(a) = U_a if a is nullary.
(2) Match $(a(t_1, \ldots, t_q)) = (U_a \cap \text{Father}_1(\text{Match}(t_1)) \cap \cdots \cap \text{Father}_q(\text{Match}(t_q))) \cup \{v\}$.

Part (2) says that the subpatterns which match at $a(t_1, \ldots, t_q)$ are exactly v plus those trees within U_a whose sons match the $t_1, \ldots, t_q$. A table for the sets U_a is easily precomputed in a single pass over the patterns in F in $O(patsize)$ time and $O(sym)$ additional space. Now, if we can find a simple way to compute $\text{Father}_i(M)$, we may assign match sets in bit-string form to each node of the subject in a simple postorder traversal of the subject tree.

A direct computation of $\text{Father}_i(M)$ seems to require a loop through all subpatterns. We suggest therefore a hashing approach. We precompute a hash table for all match sets and store $\text{Father}_i(M)$ for $1 \leq i \leq rank$ at the table entry for M. Such a table consumes $O((set/load) \times rank)$, where *load* is the loading factor of the hash table, compared to $O(set^{rank} \times sym)$ for the tables described in Section 4.

Given a hashing function for the M, the precomputation of $\text{Father}_i(M)$ in the most straightforward way takes time

$$O(set \times rank \times patsize).$$

In time $O(set \times patno)$ we can add to each entry M a list of indices i such that the entire pattern p_i is in M. This list allows us to detect matches immediately from the match sets. The only additional problem is how to choose a suitable hashing function. Since we deal with a fixed forest of tree patterns, we would like to derive "perfect" hashing functions [32], that is, hashing functions which have no collisions on the set of keys. For this, we offer two alternatives.

In the case of simple pattern forests, we take advantage of the results of Section 5, which showed that all match sets have singleton base sets. We enumerate the patterns in PF in increasing order of subsumption, for example, a depth-first numbering of G_S. In this way the base-set subpattern is always represented by the leftmost nonzero bit in the string representation of the set. Since different match sets have different base sets, they must have different numbers of leading zeros. Our hashing function now simply counts the leading bits, thereby achieving a perfect minimal hashing function. Note that a practical implementation of this is possible, since on most computers there is an instruction to normalize floating-point numbers, which involves counting leading zero bits.

For nonsimple forests the work of Sprugnoli can be used [32]. His algorithms derive a perfect hashing function using multiplication, addition, and division, but the function does not guarantee a high loading factor. Unfortunately, there is no analysis of his algorithms, so the exact space and time bounds are not known. Further research is needed to investigate whether there are special properties of match sets which lead to minimal perfect hashing functions which can be derived in a reasonable amount of time.

Bit-string representation of match sets offers another advantage. Recall that the number of match sets may be exponential in the pattern size. Therefore we should control the table size in those cases. This is possible with the following observation about the Father function:

$$\text{Father}_i(M_1 \cup M_2) = \text{Father}_i(M_1) \cup \text{Father}_i(M_2).$$

TABLE II SPACE AND TIME COMPLEXITIES FOR PREPROCESSING AND MATCHING TECHNIQUES

Method	Restrictions	Preprocessing time	Matching time	Preprocessing space	Matching space excluding space occupied by output
Naive algorithm	None	None	$O(subsize \times patsize)$	None	$O(subsize + patsize)$
Bottom up with naive preprocessing	None	$O(set^{rank+1} \times sym \times patsize)$	$O(subsize + match)$	$O(set^{rank} \times sym)$	$O(subsize + set^{rank} \times sym)$
Bottom up with Algorithms A and B	Simple pattern forest	$O(patsize^2 \times rank + ht \times sym \times patsize^{rank})$	$O(subsize + match)$	$O(patsize^{rank} \times sym)$	$O(subsize + patsize^{rank} \times sym)$
Bottom up with Algorithm C	Simple binary forest	$O(patsize \times ht^2)$	$O(subsize \times ht^2 + match)$	$O(patsize^2)$	$O(subsize + patsize^2)$
Top down with *Algorithm D*	Patterns are full trees	$O(patsize)$	$O(subsize \times patno)$	$O(patsize)$	$O(subsize \times patno + patsize)$
	None	$O(patsize)$	$O(subsize \times suf)$	$O(patsize)$	$O(subsize \times patno + patsize)$
	Uniform cost for bit-string operations	$O(patsize)$	$O(subsize + match)$	$O(patsize^2)$	$O(subsize \times patsize + patsize^2)$
Bottom up, fixed hashing	Uniform cost for bit-string operations	$O(set \times (rank \times patsize + patno))$	$O(subsize + match)$ (average) $O(subsize \times set)$ (worst case)	$O((set/load) \times (rank + patno) + sym)$	$O(subsize + (set/load) \times (rank + patno) + sym)$
Bottom up, perfect hashing	Uniform cost for bit-string operations	Simple forest: $O(set \times (rank \times patsize + patno))$ General case ?	$O(subsize + match)$	Simple forest $O(set \times (rank + patno) + sym)$	$O(subsize + (set/load) \times (rank + patno) + sym)$
Bottom up with partitioned bit strings where $set \approx 2^{patsize}$	Uniform cost for bit-string operations	$O(part \times 2^{patsize/part} \times (rank \times patsize + patno))$	$O(subsize \times part + match)$ (average) $O(subsize \times part \times 2^{patsize/part})$ (worst case)	$O(part \times 2^{patsize/part}/load \times (rank + patno) + sym)$	$O(subsize + part \times 2^{patsize/part}/load \times (rank + patno) + sym)$

Thus we may partition the set PF into a fixed, chosen number *part* of blocks $P_1, \ldots, P_{part}$ and represent each match set M by the tuple,

$$\langle M \cap P_1, M \cap P_2, \ldots, M \cap P_{part} \rangle.$$

Then (1) and (2) of Definition 10.2 become

(1′) $\text{Match}(a) \cap P_j = U_a \cap P_j$.

(2′) $\text{Match}(a(t_1, \ldots, t_q)) \cap P_j = (U_a \cap \{\bigcup_{k=1}^{part} \text{Father}_1(\text{Match}(t_1)) \cap P_k\} \cap \cdots \cap \{\bigcup_{k=1}^{part} \text{Father}_q(\text{Match}(t_q)) \cap P_k\} \cup \{v\}) \cap P_j$.

For the analysis, let set_i be the number of match set segments in the ith partition block P_i:

$$set_i = |\{\text{Match}(t) \cap P_i | t \text{ in } S\}|.$$

We can then express the table size as

$$O\left(\frac{set_1 + \cdots + set_{part}}{load \times (rank + part)}\right),$$

and the matching time as $O(subsize \times part + match)$.

For the case where *set* is nearly $2^{patsize}$ and the partition sizes $|P_i|$ are each approximately equal, that is, *patsize/part*, the table size may be expressed as

$$O\left(\frac{part}{load} \times 2^{patsize/part} \times (rank + patno)\right).$$

This formula gives a good idea of the space–time trade-off involved. Given a set of patterns, the problem of choosing a good partition is as yet unexplored. Since it may lead to a clique problem (Theorem 4.3), it can perhaps only be approximated.

11. *Conclusions*

Table II summarizes the time and space complexities for the preprocessing and matching techniques we have discussed. The trade-offs are so complex that we cannot choose an all-round best method. Each of the techniques offers some strengths and has certain weaknesses.

As in the case of sorting, users of tree-matching algorithms must choose a strategy carefully, on the basis of special properties of the patterns and subjects involved, the number of different subjects expected (and their relationship, if any) for the same set of patterns, and the available time and space resources.

We note that our top-down algorithm is always better than the one of Karp et al. [18] and as good as the one of Overmars and van Leeuwen [27], although they have a different notion of matching in mind. It is only in especially space-limited situations that the naive matching algorithm should be chosen. The version of Lang et al. [24] might be an interesting alternative, but further experimentation seems necessary to understand better what practical advantages it has to offer.

For the quickest matching time, the bottom-up algorithm, driven by tables, is best. We have used it in our interpreter generator and feel that for this application, the additional matching speed justifies the added preprocessing time, as long as the table size stays reasonable. Our experience with the algorithm is confirmed by the work in [10]. When too many match sets are expected, we suggest the bit-string and hash-table methods which trade off space and time very flexibly.

REFERENCES

(Note. References [5, 19] are not cited in the text.)

1 AHO, A.V., AND CORASICK, M J. Efficient string matching: An aid to bibliographic search. *Commun. ACM 18*, 6 (June 1975), 333–340

2. AHO, A.V , HOPCROFT, J.E., AND ULLMAN, J D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass , 1974.

3. BAXTER, L.D. The complexity of unification. Ph.D. Dissertation, Dep. of Computer Science, Univ. of Waterloo, Waterloo, Ontario, Canada, 1976

3a BERRY, G , AND LÉVY, J -J Minimal and optimal computations of recursive programs. 4th ACM Symp. on Principles of Programming Languages, Los Angeles, Calif , 1977, pp 215–226

4 BOYER, R S., AND MOORE, J.S. A fast string searching algorithm *Commun ACM 20*, 10 (Oct 1977), 762–772

5 CARTER, J L , AND WEGMAN, M.N Universal classes of hashing functions Proc 9th Ann. Symp on Theory of Computing, Boulder, Colo , 1977, pp 106–112.

6 CHEW, P An improved algorithm for computing with equations Proc 21st IEEE Symp on Foundations of Computer Science, Syracuse, N.Y , 1980, pp. 108–117

7 COLLINS, G The SAC-1 system· An introduction and survey Proc 2nd ACM Conf on Symbolic and Algebraic Manipulation, Los Angeles, Calif , 1971, pp 144–152

8. COMMENTZ-WALTER, B A string matching algorithm fast on the average In *Automata, Languages and Programming*, Lecture Notes in Computer Science 71, H A Maurer, Ed , Springer-Verlag, Berlin, Heidelberg, New York, 1979, pp. 118–132.

9. DOWNEY, P J , SAMET, H , AND SETHI, R Off-line and on-line algorithms for deducing equalities Proc 5th Ann ACM Symp on Principles of Programming Languages, Tucson, Ariz , 1978, pp. 158–170.

10 GLASNER, I , MONCKE, U , AND WILHELM, R OPTRAN, a language for the specification of program transformations 6th G I Fachtagung uber Programmiersprachen, Darmstadt, W Germany, 1980, to appear in Lecture Notes in Computer Science

11. GOGUEN, J.A Some design principles and theory for Obj-0 Proc Int Conf on Mathematical Studies of Information Processing, Kyoto, Japan, 1978, pp 429–475.

12. GUTTAG, J , HOROWITZ, E , AND MUSSER, D. Abstract data types and software validation. ISI Rep 76-48, Univ. of Southern California, Los Angeles, Calif., 1976.

13. GUTTAG, J V , HOROWITZ, E , AND MUSSER, D R Abstract data types and software validation *Commun. ACM 21*, 12 (Dec 1978), 1048–1064

14 HOFFMANN, C M , AND O'DONNELL, M J An interpreter generator using tree pattern matching Proc. 6th Ann ACM Symp on Principles of Programming Languages, San Antonio, Texas, 1979, pp. 169–179

15. HOFFMANN, C.M., AND O'DONNELL, M.J. Programming with equations *ACM Trans Prog Lang Syst 4*, 1 (Jan 1982)

16 HUET, G., AND LANG, B Proving and applying program transformations expressed with second order patterns Tech Rep 266, IRIA Laboria, LeChesnay, France, 1977

17 HUET, G , AND LEVY, J -J Call by need computations in nonambiguous linear term rewriting systems Tech. Rep 359, IRIA Laboria, LeChesnay, France, 1979.

18 KARP, R , MILLER, R E., AND ROSENBERG, A Rapid identification of repeated patterns in strings, trees and arrays Proc 4th Ann ACM Symp on Theory of Computing, Denver, Colo., 1972, pp 125–136

19 KNUTH, D. *The Art of Computer Programming, Vol 3 Sorting and Searching* Addison-Wesley, Reading, Mass , 1973

20 KNUTH, D., AND BENDIX, P. Simple word problems in universal algebras In *Computational Problems in Abstract Algebra*, J Leech, Ed , Pergamon Press, Elmsford, N Y , 1970, pp 263–297.

21 KNUTH, D., MORRIS, J , AND PRATT, V Fast pattern matching in strings. *SIAM J Comput 6*, 2 (1977), 323–350

22 KOZEN, D Complexity of finitely presented algebras Proc 9th Ann. ACM Symp on Theory of Computing, Boulder, Colo , 1977, pp 164–177

23 KRON, H. Tree templates and subtree transformational grammars Ph D Dissertation, Univ of California, Santa Cruz, Calif , 1975

24 LANG, H -W , SCHIMMLER, M , AND SCHMECK, H Matching tree patterns sublinear on the average Tech Rep , Dep of Informatik, Univ Kiel, Kiel, W Germany, 1980

25 NELSON, G , AND OPPEN, D Fast decision procedures based on congruence closure *J ACM 27*, 2 (April 1980), 356–364

26 O'DONNELL, M J Computing in systems described by equations In *Computing and Systems De-*

scribed by Equations, Lecture Notes in Computer Science 58, G Goos and J Hartmanis, Eds., Springer–Verlag, 1977

27 OVERMARS, M H., AND VAN LEEUWEN, J Rapid subtree identification revisited Tech Rep CS-79-3, Univ of Utrecht, Utrecht, Netherlands, 1979

28 PATERSON, M S., AND WEGMAN, M Linear unification Proc 8th ACM Symp on Theory of Computing, Hershey, Pa., 1976, pp 181–186

29 ROBINSON, J A A machine-oriented logic based on the resolution principle. *J ACM 12*, 1 (Jan 1965), 23–41

30 ROSEN, B Tree-manipulating systems and Church–Rosser theorems *J ACM 20*, 1 (Jan 1973), 160–187.

31 SHOSTAK, R E An algorithm for reasoning about equality *Commun ACM 21*, 7 (July 1978), 583–585

32 SPRUGNOLI, R Perfect hashing functions A single probe retrieving method for static sets *Commun ACM 20*, 11 (Nov 1977), 841–850

33 STAFFORD, G Structure of the Eh compiler Master's Thesis, Dep of Computer Science, Univ of Waterloo, Waterloo, Ontario, Canada, 1977

34 WAND, M Algebraic theories and tree rewriting systems Tech Rep 66, Dep of Computer Science, Indiana Univ., Bloomington, Ind., 1977

RECEIVED MARCH 1979, REVISED NOVEMBER 1980, ACCEPTED DECEMBER 1980