Efficient Parallel Term Matching

Art Delcher
Simon Kasif

Department of Computer Science
Johns Hopkins University
Baltimore, MD 21218

January, 1988
Report JHU-CS-88/7

Efficient Parallel Term Matching

Arthur L. Delcher
Simon Kasif

Computer Science Department
The Johns Hopkins University
Baltimore, MD 21218

Abstract

In this paper we present several $O(N / \log^2 N)$-processor, $O(\log^2 N)$-time EREW PRAM parallel algorithms for term matching problems. Term matching is the special case of unification in which one of the terms is restricted to contain no variables. It has wide applicability to logic programming, term rewriting systems and symbolic pattern matching used in Artificial Intelligence. Anti-unification is the dual problem of unification in which one computes the most specific generalization of two terms. It has application to inductive inference and theorem proving. The algorithms presented are the first to have a processor$\times$time product of the same order as that of the sequential algorithm, thus ensuring optimal speed-ups using any number of processors up to $O(N / \log^2 N)$.

1. Introduction

Unification is a basic operation in logic programs. In the context of Artificial Intelligence, unification is an important special case of pattern matching. In sequential implementations of Prolog and other logic programming languages, the unification operation is not particularly time-consuming in practice. This is true, since at any point in the computation the interpreter unifies a single goal literal with a single procedure head. This operation often corresponds to binding a variable to a constant (pointer) or performing a

simple equality check.

This situation changes when AND-parallelism is allowed [3] [18]. Consider a goal of the form

$$:- p(X_1, X_2), p_2(X_2, X_3), p_3(X_3, X_4) \ldots$$

The sequential computation can match a single goal literal to a single procedure head in constant time, but the parallel computation is substantially more complex because of the shared variables among the goal literals. As a specific example, consider the goal

$$:- p(X_1, X_2), p(X_2, X_3), p(X_3, X_4), p(X_5, X_6), p(X_7, X_7), q(X_1, X_7).$$

and the program

$$p(X, X).$$
$$q(0, 1).$$

It is easy to see that the parallel unification of all the subgoals succeeds iff the node X_1 is not in the same connected component as the node X_7 in the graph below.

$$X_1 \text{ —— } X_2 \text{ —— } X_3 \text{ —— } X_4 \qquad X_5 \text{ —— } X_6 \qquad X_7$$

Of course, the same phenomenon occurs in the context of sequential computation if, for example,

$$f\ (\ p(X_1, X_2), p(X_2, X_3), p(X_3, X_4), p(X_5, X_6), p(X_7, X_7), q(X_1, X_7)\ )$$

is unified with

$$f\ (\ p(X, X), p(X, X), p(X, X), p(X, X), p(X, X), q(0, 1)\ )$$

but in practice this occurs far less frequently. This simple example illustrates how parallel computation may generate different computational demands. Thus, in the context of massively parallel machines we need to address the problem of designing an efficient parallel algorithm for matching large terms. In this case, one can show that if no function

symbols are allowed in the terms, we can perform the matching by using an algorithm similar to a parallel algorithm for computing connected components (transitive closure). Parallel connected component computation can be done in $O(\log^2 N)$ time. More importantly, the algorithm achieves nearly optimal speed-ups. Unfortunately, this is not true for general unification, and in fact, unification is known to be P-complete [6]. It is believed that no parallel algorithm can solve a P-complete problem in logarithmic time with a polynomial number of processors [8]. Additionally, there are no known practical parallel algorithms for general unification that achieve linear speed-ups for any given number of processors (see [21] for an interesting approach to general unification).

In this paper we address the problems of term-matching and anti-unification. *Term matching* is defined as matching an arbitrary logical term P to a fully ground term G. Term matching is an important special case of symbolic pattern matching. The utility of term matching in the context of logic programming was discussed in [13]. Term matching also has a wide applicability to term rewriting systems used in functional and equational programming. If both terms are represented as trees there are several parallel algorithms to perform the match (see Section 7). We describe an efficient parallel algorithm which runs in $O(\log^2 N)$ parallel time and uses $O(N / \log^2 N)$ processors. Thus, the processor$\times$time product is optimal for any given number of processors less than $\log^2 N$, *i.e.*, the algorithm achieves an optimal speed-up. The algorithm is well-suited for implementation on parallel models of computation with bounded degree networks.

Anti-unification is defined as computing a most specific generalization G of two terms T_1 and T_2, in the sense that T_1 and T_2 are both instances of G and G is an instance of any other term with this property. Anti-unification is useful as a model for

inductive inference [15], in theorem proving [17], in resolution and equation solving [10], and in transformations and optimizations of logic programs [14]. We describe an $O(\log^2 N)$-time $O(N / \log^2 N)$-processor EREW PRAM algorithm to compute the anti-unification of two terms expressed as trees.

The principal motivation behind the research reported in this paper is the development of efficient algorithms that are practical to implement. Thus, we emphasize algorithms that achieve optimal speed-ups with a given number of processors, in contrast to attaining optimal parallel time with a (possibly) large number of processors.

2. Preliminaries

The basic model of computation we use is a shared-memory parallel machine (PRAM) in which no two processors may either read from or write to a single memory location simultaneously (Exclusive Read/Exclusive Write or EREW). This is the weakest shared-memory model. We adopt this model because we believe it currently is the most suitable model for general purpose computing. In most cases one can map an EREW PRAM algorithm to a more realistic parallel network such as the butterfly or hypercube with at most logarithmic overhead [20].

We assume the reader is familiar with the unification problem. A *logical term* is defined recursively as follows: T is a logical term if T is:

- a 0-ary function (atom): a, b, c, ...
- a variable: X, Y, Z, ...
- an expression of the form $f(T_1, \ldots, T_n)$ where f is a n-ary function symbol and each T_i is a logical term.

We assume the terms are represented as labelled directed trees in the standard way (see Figure 1). Nodes in the tree are labelled with n-ary function symbols or variables. Edges

are labelled with integers 0, 1, 2, ... such that the edge from a node to its i-th argument is labelled with the integer $i-1$. Note that variables can only appear as leaves on the tree. With each node u in the tree we associate the following values:

- $Id(u)$ — a unique identifier for the node.
- $Label(u)$ — the label associated with the node, *i.e.*, function name or variable name. For function names, this includes the arity of the function, *i.e.*, its number of children in the tree.
- $Par(u)$ — a pointer to the parent of u in the tree.
- $Argnum(u)$ — the label on the edge between u and its parent.
- $Weight(u)$ — the number of nodes in the subtree rooted at u.
- $Level(u)$ — the depth of u in the tree.

3. Term Matching

The term matching problem is to unify terms P and G, where G is restricted to have no variables. We shall refer to P as the *pattern term*, and G as the *ground term*. The most important step in the term matching algorithm is to identify corresponding nodes in P and G. This correspondence between nodes is a one-to-one mapping $\leftrightarrow$ between the nodes of the two trees and is defined recursively by:

- Root $(P) \leftrightarrow$ Root (G).
- If $Par(u) \leftrightarrow Par(v)$ and $Argnum(u) = Argnum(v)$ then $u \leftrightarrow v$.

Once this correspondence has been computed we can solve the term matching problem by performing the following steps:

STEP 1: If any node u in the pattern term P has no corresponding node in G, then the terms do not match.

STEP 2: For each pair of corresponding nodes $u \leftrightarrow v$ where $Label(u)$ is a function name, we check whether $Label(u) = Label(v)$. If there are any mismatches, then P and G

do not unify.

STEP 3: For each pair of corresponding nodes $u \leftrightarrow v$ where $Label(u)$ is a variable, we identify the tree rooted at v with the variable name. We then compare all the trees identified with the same variable name for equality. Given two or more ground terms, it is not hard to check whether they are identical with an optimal EREW algorithm based on the arity of each node and depth-first numbering [4] [19].

Thus, the key to an efficient matching algorithm is performing the correspondence computation, which we describe in the next section. Without loss of generality we henceforth assume that P and G are binary trees. Otherwise, they can be converted to binary trees in a canonical fashion such that the binary trees correspond iff the original trees correspond.

4. The Basic Correspondence Algorithm

In this section we describe the basic ideas and the main procedures in our algorithm to identify corresponding nodes between the pattern tree P and the ground tree G.

The input to the algorithm consists of two logical terms, P and G, each represented as a binary tree, and together containing N nodes. P is an arbitrary logical term, and G is a fully ground (variable-free) term. The output of the algorithm will be either the set of correspondences between nodes in P and G, or else a failure indication if P and G do not match.

4.1. Identifying Corresponding Nodes

We first observe that node u in P corresponds to node v in G iff the sequence of edge labels on the path from u to the root of P exactly matches the sequence of edge

labels on the path from v to the root of G. This leads to the following algorithm described in [16]: For each node in P and G compute the sequence of edge labels on its path to the root. Then bring matching nodes together by sorting them according to the edge label sequences. Computing these sequences of edge labels for each node can be done in parallel with $O(N / \log N)$ processors in $O(\log N)$ time on an EREW PRAM using expression-tree evaluation techniques such as in [1] [9] [11]. The problem that arises is that lengths of these sequences may be $O(N)$ bits long, making the matching process difficult. Since there are only N nodes in the combined trees, however, there are at most N different sequences actually present. Thus, $O(\log N)$ bits suffice to give each sequence a distinguishing ID number. If such ID numbers can be assigned to the sequences, then the corresponding nodes can be determined in constant time per node by a simple table look-up. This is the idea behind the algorithm developed independently by [12] and [16] where, as the paths from each node to the root are being calculated, they are sorted and assigned $O(\log N)$-bit ID numbers. The resulting algorithm uses $O(N)$ processors and runs in $O(\log^2 N)$ time on an EREW PRAM. Thus, it is a factor of $O(\log^2 N)$ away from being processor$\times$time optimal. It is not clear how to achieve optimal speed-up using these methods. Our method is based on the divide-and-conquer technique proposed in [7].

4.2. Algorithm Outline

We begin our correspondence algorithm with a preprocessing phase in which the values $Level(u)$ and $Weight(u)$ are computed for each node in both P and G. This can both be done in $O(\log N)$ time with $O(N / \log N)$ processors on an EREW PRAM using techniques described in [4] [19].

The main algorithm consists of the following steps:

STEP 1: In P, find a path down from the root, such that when this path is removed, each remaining subtree has no more than half as many nodes as P.

STEP 2: Find the corresponding path in G. This will in turn decompose the target tree into relatively small subtrees corresponding to those remaining in the pattern tree. If there is no corresponding path in G, then report failure.

STEP 3: In parallel, recursively match the corresponding subtrees to each other.

These major steps are depicted in Figure 2. We now describe their EREW PRAM implementation.

STEP 1: First the total number of nodes in P is broadcast to each node in P. Then in parallel, each node u checks whether *Weight* (u) is at least half of that value. The nodes that succeed are the nodes on the desired path.

STEP 2: In parallel, each node u on the path in P broadcasts to all nodes on its same level in G the value of *Argnum* (u). Each node v in G then determines whether its value of *Argnum* (v) matches. Among these matching nodes in G (which are indicated by *'s in Figure 2a), there is now a unique path down from the root. The nodes along this unique path can identify themselves by testing whether all their ancestors in the tree are matching nodes using parallel expression-tree evaluation techniques. At the end of this step, there is exactly one node at each level in each tree. These nodes can then exchange and compare their information. If there is a discrepancy, the algorithm halts with a failure indication.

STEP 3: Each node that is a child of a node on the selected path now becomes the root of a subtree to be matched recursively. It identifies the root of its corresponding subtree from the information in its parent. In parallel, the same steps are now repeated recursively for each corresponding subtree pair.

4.3. Complexity Analysis

Each step in the above procedure takes $O(\log N)$ steps for $O(N / \log N)$ processors on an EREW PRAM using broadcasting and expression-tree evaluation techniques. Since at each stage the size of the pattern trees is being halved, the matching process is complete after $O(\log N)$ stages. Thus the total complexity of the above procedure is $O(\log^2 N)$ parallel time with $O(N / \log N)$ processors on an EREW PRAM. This is still a factor of $O(\log N)$ away from optimal. In the next section we show how to improve this to obtain optimal speed-up.

5. Achieving Optimal Speed-Up

In this section we show how to improve our procedure to run in $O(\log^2 N)$ time using $O(N / \log^2 N)$ processors on an EREW PRAM. The idea is to preprocess the original trees P and G to reduce their sizes by a factor of $\log N$. We then apply the procedure of the previous section to the reduced trees. Finally we add a postprocessing step to complete the matching process for all remaining nodes.

We start by marking every node u in the original trees P and G that has the following properties: (1) u is on level $k \lceil \log N \rceil$, for some positive integer k; and (2) the subtree rooted at u contains at least $\log N$ nodes. Such nodes can be determined in parallel in constant time per node based on the values of $Weight(u)$ and $Level(u)$.

Lemma 1: At most $N/\log N$ nodes are marked in the above step.

Pf: Allocate each unmarked node to its nearest marked ancestor. Then there are at least $\log N - 1$ nodes allocated to each marked node, *i.e.*, at most one of every $\log N$ nodes is marked. □

We now generate trees $\hat{P}$ and $\hat{G}$ from P and G by removing all unmarked nodes, making the marked nodes the children of their nearest marked ancestors in the original tree. Clearly, the new tree is of size at most $N/\log N$. We can label each edge (u, v) in $\hat{P}$ and $\hat{G}$ with an integer label in the range $0 \cdots N-1$ as determined by the sequence of 0/1 labels on the path connecting u and v in the original binary tree, P or G. These labels can be computed on a $O(N/\log N)$-processor EREW PRAM in $O(\log N)$ time by a simple top-to-bottom scan of the edges on each level, since there are at most $\log N$ levels involved.

Now consider the correspondence problem for $\hat{P}$ and $\hat{G}$. We first note that if P and G match, then every node u in $\hat{P}$ has its matching node v contained in $\hat{G}$. Therefore, we can match trees P and G, by first matching the two trees $\hat{P}$ and $\hat{G}$, and then use the correspondences obtained to match all the remaining nodes. These remaining nodes are grouped in trees of depth at most $O(\log N)$ under nodes whose correspondences have already been determined. Thus, the remaining nodes can be matched by parallel top-to-bottom scans of the $O(\log N)$-depth trees into which they are grouped. This requires $O(\log N)$ time and $O(N/\log N)$ processors on an EREW PRAM.

$\hat{P}$ and $\hat{G}$ together contain at most $N/\log N$ nodes. Therefore, the procedure of the previous section can match them in $\log^2 N$ time using $O(N/\log^2 N)$ processors. Note that $\hat{P}$ and $\hat{G}$ are not binary trees. However, none of the steps in the algorithm of

the previous section requires that the trees be binary. Our assumption that P and G were binary trees was made to simplify the process of selecting the nodes to include in $\hat{P}$ and $\hat{G}$ and to guarantee that their edge labels were in the range $0 \ldots N-1$.

Thus we have shown:

Proposition 1: The correspondence of two trees of size N can be determined in $O(\log^2 N)$ time with $O(N / \log^2 N)$ processors on an EREW PRAM.

6. Anti-Unification

In this section we show how the algorithm of Section 5 can be used to give an optimal $O(N / \log^2 N)$-processor, $O(\log^2 N)$-time EREW PRAM algorithm for the anti-unification problem, namely finding the most specific generalization of two given terms. In this section we assume that there are only finitely many different node labels present in the two terms.

6.1. Definitions

We define most specific generalization as follows: Given two terms, t_1 and t_2, their *most specific generalization* t_g is the unique term (up to variable renaming) such that both t_1 and t_2 are instances of t_g, and such that t_g is an instance of any other term of which both t_1 and t_2 are instances. t_g can be computed based on the following recursive algorithm described in [10]:

- If either of t_1 or t_2 is a variable, then t_g must be a variable.
- If $t_1 = f(r_1, r_2, \cdots, r_n)$, $t_2 = g(s_1, s_2, \cdots, s_m)$ and $f \neq g$ or $m \neq n$, then t_g must be a variable.
- If $t_1 = f(r_1, r_2, \cdots, r_n)$, $t_2 = f(s_1, s_2, \cdots, s_n)$, then $t_g = f(u_1, u_2, \cdots, u_n)$ where each u_i is the most specific generalization of r_i and s_i, *and the u_i are labelled consistently.* This means that if $r_i = r_j$ and $s_i = s_j$,

then u_i must be *identical* to u_j. For example, the most specific generalization of $t_1 = f\ (a\ ,\ a\ )$ and $t_2 = f\ (b\ ,\ b\ )$ is $f\ (X\ ,\ X\ )$ and not $f\ (X\ ,\ Y\ )$

6.2. Basic Algorithm

Following [12], this algorithm can be cast into the following form:

STEP 1: Find corresponding nodes in trees t_1 and t_2.

STEP 2: Mark each corresponding pair of nodes $u \leftrightarrow v$ that have matching labels. Then find each marked node with the property that all its ancestors are also marked. These nodes are precisely the non-variable nodes in the most specific generalization tree t_g.

STEP 3: Find all corresponding pairs of nodes $u \leftrightarrow v$ that are both unmarked and children of nodes selected for inclusion in t_g by the previous step. These nodes are precisely the nodes that will correspond to variables in t_g.

STEP 4: Determine consistent variable names for all the nodes selected in the preceding step.

Steps 1 through 3 can be done optimally using the techniques described in Sections 5 and 6 of this paper. We next describe an optimal parallel algorithm to compute consistent variable labels for step 4.

6.3. Computing Consistent Variable Names

At this point we have a collection of corresponding node pairs $u_i \leftrightarrow v_i$, $1 \leq i \leq k$. Each of these nodes is the root of a subtree that represents a logical term, and none of the subtrees overlap. Our problem is to assign a distinguishing variable ID to each pair $u_i \leftrightarrow v_i$ in such a way that the ID of $u_i \leftrightarrow v_i$ equals the ID of $u_j \leftrightarrow v_j$ iff the subtrees

rooted at u_i and u_j are identical and the subtrees rooted at v_i and v_j are identical.

We can begin by converting each pair of subtrees $u_i \leftrightarrow v_i$ into a single string by performing a pre-order traversal first of the nodes in u_i's subtree and then of the nodes in v_i's subtree (See Figure 3). This can be done in optimal parallel time using techniques described in [4] and [19]. Thus, from now on we can assume we have k strings, containing all together at most N symbols. Note that our problem could now be solved by sorting the strings, but that approach would not yield an optimal parallel algorithm.

Instead, we first partition these strings into two sets based on their lengths. Into one set we place those strings containing no more than $\log N$ symbols; the remaining strings are placed into the second set. We shall assign distinguishing ID's to each of these sets separately.

For the strings containing no more than $\log N$ symbols our task is easy. Since we have assumed that the symbol alphabet is finite, we can simply take as our ID the string itself. Each ID is thus $O(\log N)$ bits long. By assigning one processor to each string, this takes $O(\log N)$ time with $O(N / \log N)$ processors on an EREW PRAM.

For the longer strings we proceed as follows: First, partition each string into consecutive substring segments, each of length $\log N$. Then for each substring, in parallel, calculate an ID in the same fashion that we calculated ID's for the shorter strings. We now sort together all the substring ID's and replace each one by its ordinal position in the sorted list (not counting duplicates). There are at most $O(N / \log N)$ ID's to be sorted, so using an algorithm such as that in [5] we can accomplish this step in $O(\log N)$ time with $O(N / \log N)$ processors on an EREW PRAM.

At this point, each string is represented by a list of ID's. We now perform the following procedure until each string's list is reduced to a single ID.

STEP 1: In parallel, combine pairs of consecutive ID's in each string to generate a new ID. Each ID in an even-numbered position in a string can simply concatenate with the ID in the preceding odd-numbered position. Any odd ID at the end of a string can concatenate with a predetermined unique null-ID.

STEP 2: Sort together all the new ID's, and replace each by its ordinal position in the sorted list (not counting duplicates).

This procedure reduces the length of the list of ID's in each string by a factor of 2. Thus, after $O(\log N)$ iterations, each string is reduced to a single ID. An easy inductive argument on the length of the original list of ID's shows that two strings receive the same final ID iff they initially have identical lists of ID's.

We should note that it is not necessary to sort all ID's together in a single sort. We need only sort together those ID's with the same ordinal positions in the lists of their respective strings. As a result, we can replace the single sort described above, with several smaller sorts that can be conducted in parallel. In the worst case, however, the asymptotic complexity will not improve.

6.4. Complexity Analysis

At first glance it appears that the preceding algorithm goes through $O(\log N)$ sorting phases, where each phase uses $O(N / \log N)$ processors and $O(\log n)$ time. This is not actually the case, however, because at each stage we reduce the number of ID's to be sorted by a factor of 2. As a result, the entire algorithm consists of a total of $O(\log N)$

phases, but the total number of operations performed over all the phases is only $O(N)$, namely, N operations in the 1st phase, $N/2$ in the 2nd, $N/4$ in the 3rd, etc. As a result, using [2]:

Brent's Theorem: Any PRAM algorithm that runs in α phases, where phase i requires p_i processors and runs in $O(1)$ steps, can be implemented on a p-processor PRAM to achieve a speed of $O(\Sigma p_i/p + \alpha)$.

we can accomplish the entire procedure in $O(\log^2 N)$ time with $O(N/\log^2 N)$ processors.

Thus we have shown:

Proposition 2: The most specific generalization of two terms with combined size N can be computed in $O(\log^2 N)$ time with $O(N/\log^2 N)$ processors on an EREW PRAM.

7. Summary

In this paper we have described efficient parallel algorithms for term matching and anti-unification. Several other algorithms for the term-matching problem have been reported in the literature. The first parallel algorithms were developed in [6] and [7]. The main goal of these algorithms was achieving the best possible parallel time. Specifically, Dwork, Kanellakis & Stockmeyer [7] developed a $O(\log^2 N)$ CREW PRAM algorithm for matching two terms represented by trees using N^2 processors. Thus, the algorithm is not processor$\times$time optimal. The main advantage of this algorithm is the fact that it can be extended to matching terms represented by dags (a more compact representation). However, the large number of processors required, $O(N^5)$, makes the algorithm impractical. Ramakrishnan *et al.* [16] present an elegant algorithm for term matching for trees. This algorithm works in $O(\log^2 N)$ time on an EREW PRAM using N processors, which again is not processor$\times$time optimal. The algorithm uses $\log N$ phases, each requiring N

numbers to be sorted.

Kuper et al. [12] describe an N-processor, $O(\log^2 N)$-time CREW PRAM algorithm for anti-unification of two terms. This algorithm also uses sorting to identify corresponding terms, and to assign consistent labels.

The divide-and-conquer strategy we use is similar to the one reported in [7]. However, our implementation uses only a sublinear number of processors, whereas theirs is quadratic. We also use a more restricted model of computation (EREW PRAM). The principal advantages of our algorithms are these:

1. They implement a natural divide-and-conquer strategy.
2. After each parallel step the computation becomes more and more local, thus the algorithm is amenable for implementation on bounded degree networks.
3. The algorithms can be implemented naturally in an asynchronous language such as Concurrent Prolog (though optimal speed-up will not be preserved).
4. Many cases of non-matching terms are detected at an early phase of the algorithm.

In practice the algorithm is likely to perform better than the reported upper bound since either the paths that decompose the pattern term are relatively short and the complexity of the decomposition step is small, or the resulting trees are relatively small and the recursive problems can be solved quickly.

References

[1] Abrahamson, K., N. Dadoun, A. Kirkpatrick, and T. Przytycka, A Simple Parallel Tree Contraction Algorithm, Tech. Rep. 87-30, Dept. of Computer Science, University of British Columbia, Vancouver, BC, Canada, 1987.

[2] Brent, R. P., The Parallel Evaluation of General Arithmetic Expressions, *JACM* **21**, pp. 201-208, 1974.

[3] Clark, K. L. and S. Gregory, Parlog: A Parallel Logic Programming Language, Research Report Doc 8315, May 1983.

[4] Cole, R. and U. Vishkin, Deterministic Coin Tossing with Applications to Optimal Parallel List Ranking, *Information and Control* **70**, 1986.

[5] Cole, R., Parallel Merge Sort, *Proc. 27th Symp. Found. of Computer Science*, pp. 511-516, 1986.

[6] Dwork, C., P. C. Kanellakis, and J. C. Mitchell, On the Sequential Nature of Unification, *Journal of Logic Programming* **1**, 1, pp. 35-50, 1984.

[7] Dwork, C., P. C. Kanellakis, and L. Stockmeyer, Parallel Algorithms for Term Matching, *1986 Conference on Automated Deduction*, July 1986.

[8] Garey, M. R. and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Fransisco, CA., 1979.

[9] Gibbons, A. and W. Rytter, An Optimal Randomized Parallel Algorithm for Dynamic Expression Evaluation and Its Applications, *Symp. on Found. of Software Technology and Theoretical Comp. Sci.*, Springer Verlag, 1986.

[10] Huet, G., Resolution d'Equations Dans Des Languages D'Order 1, 2, ..., ω, *PhD Thesis*, Universite de Paris VII, 1976.

[11] Kosaraju, S. R. and A. L. Delcher, Optimal Evaluation of Tree-Structured Computations by Raking, *Proc. AWOC*, 1988.

[12] Kuper, G. M., K. W. McAloon, K. V. Palem, and K. J. Perry, Efficient Parallel Algorithms for Anti-Unification and Relative Complement. 1987.

[13] Maluszynski, J. and H. J. Komorowski, Unification-Free Execution of Horn Clause Programs, *Proc. of 2nd Logic Symposium*, pp. 78-86, 1985.

[14] Naish, L. and J.-L. Lassez, Most Specific Logic Programs, Tech. Rep., Melbourne University, 1984.

[15] Plotkin., G., A Note on Inductive Generalization., *Machine Intelligence* **5**, pp. 153-163, 1970.

[16] Ramesh, R., R. M. Verma, T. Krishnaprasad, and I. V. Ramakrishnan, Term Matching on Parallel Computers, *Proc. 14th ICALP*, 1987.

[17] Reynolds, J., Transformations on Inductive Generalization., *Machine Intelligence* **6**, pp. 101-124, 1971.

[18] Shapiro, E. Y., A Subset of Concurrent Prolog and its interpreter, TR-003, ICOT, 1983.

[19] Tarjan, R. E. and U. Vishkin, Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time, *SIAM Journ. Computing* **14**, pp. 862-874, 1985.

[20] Vishkin, U., Implementation of Simultaneous Memory Access in Models that Forbid It, *Journal of Algorithms* **4**, 1, pp. 45-50, March 1983.

[21] Vitter, J. S. and R. A. Simons, New Classes for Parallel Complexity: A Study of Unification and Other Complete Problems in P, *IEEE Transactions On Computers*, pp. 403-418, 1986.

Sample Term:

$$f\ (\ g(a,\ X,\ b),\ h(a,\ Y))$$

Corresponding Tree:

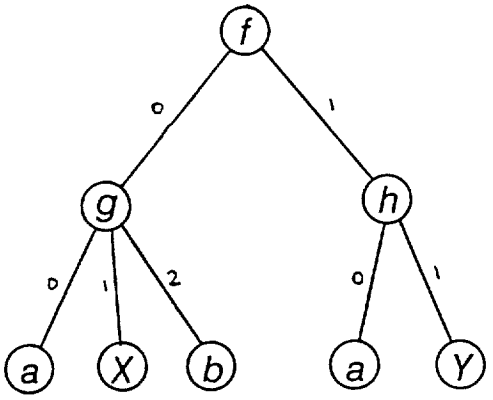


Figure 1. Representing logical terms as trees.

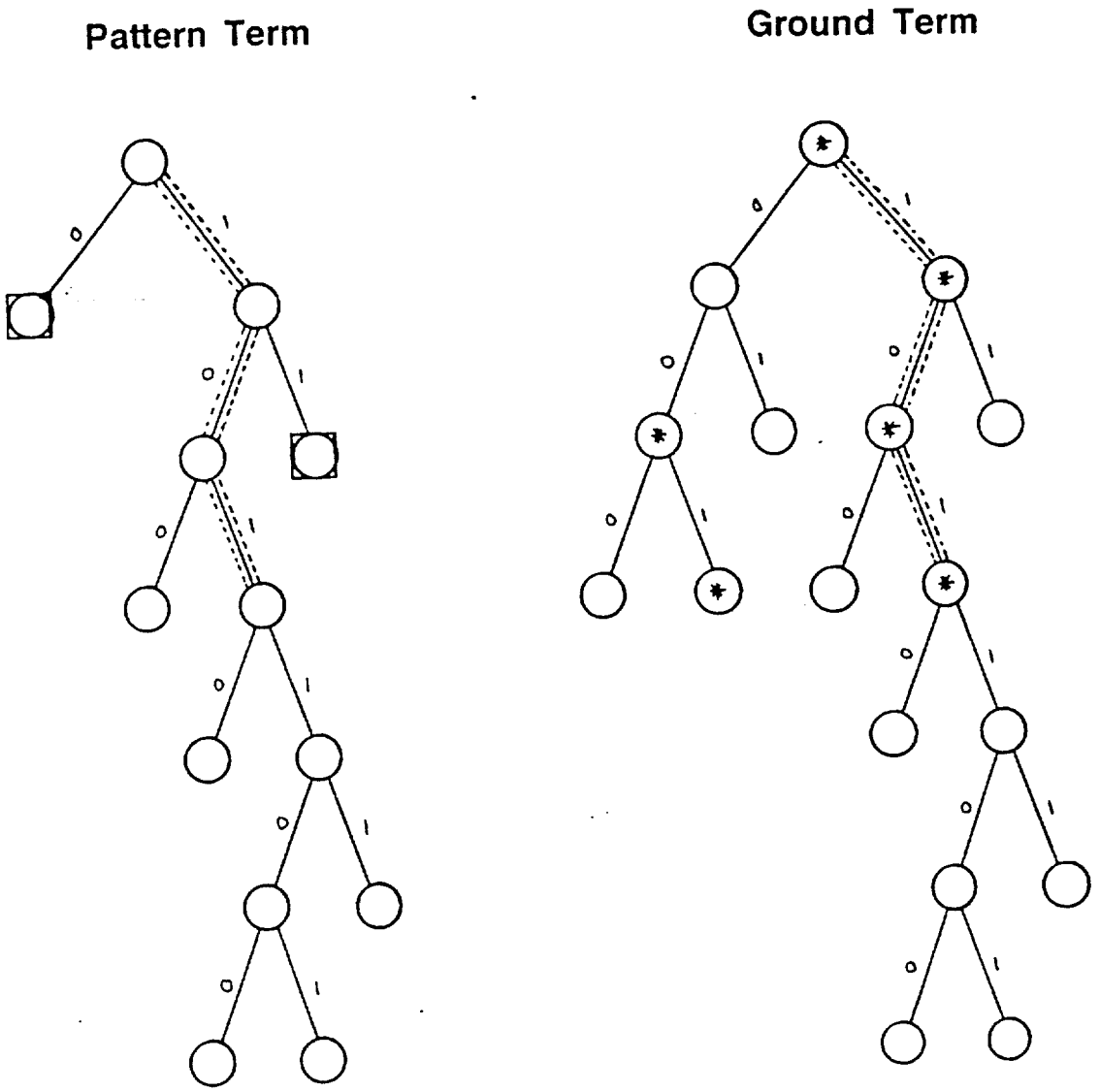


Figure 2a. Finding corresponding paths.

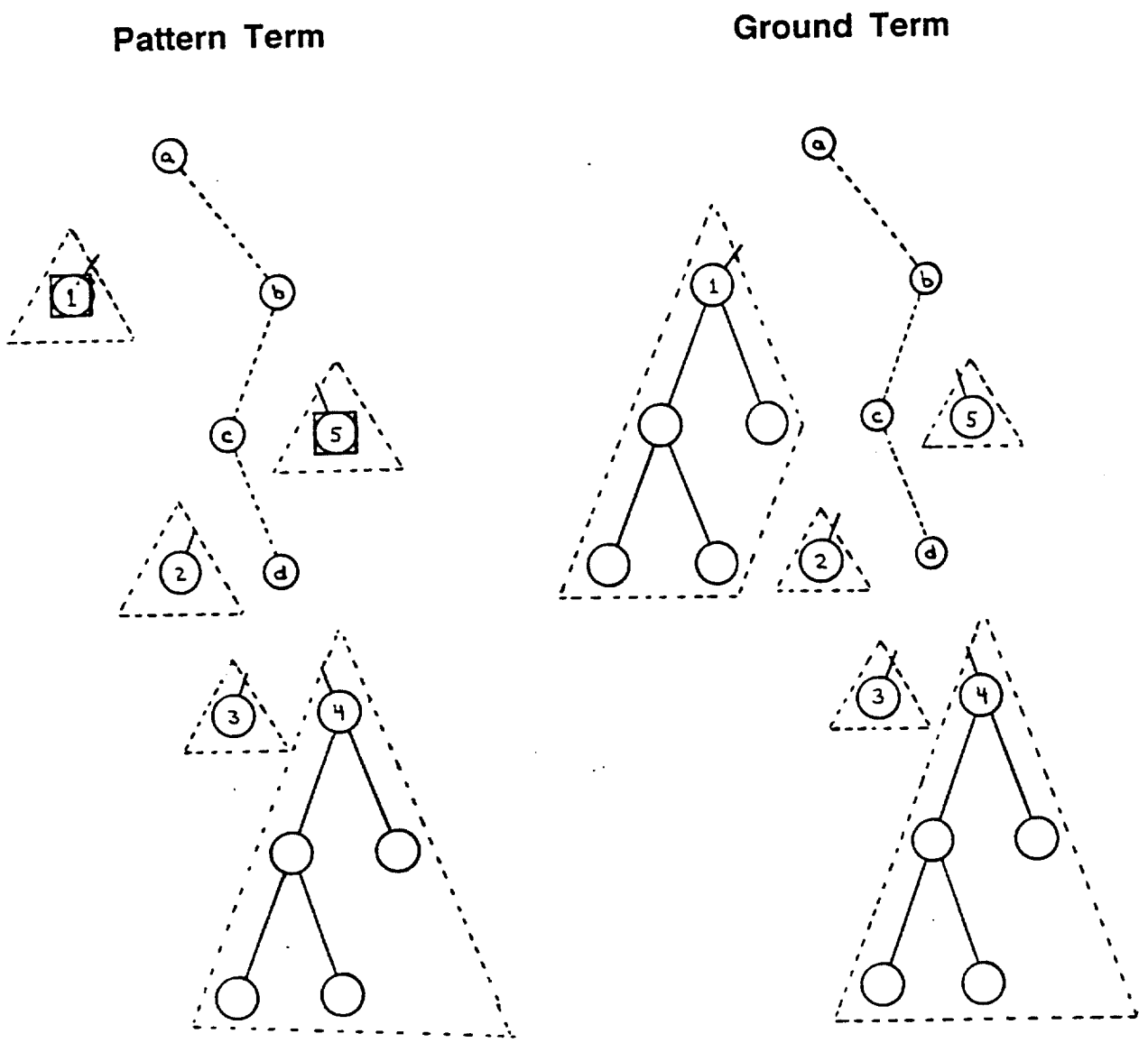
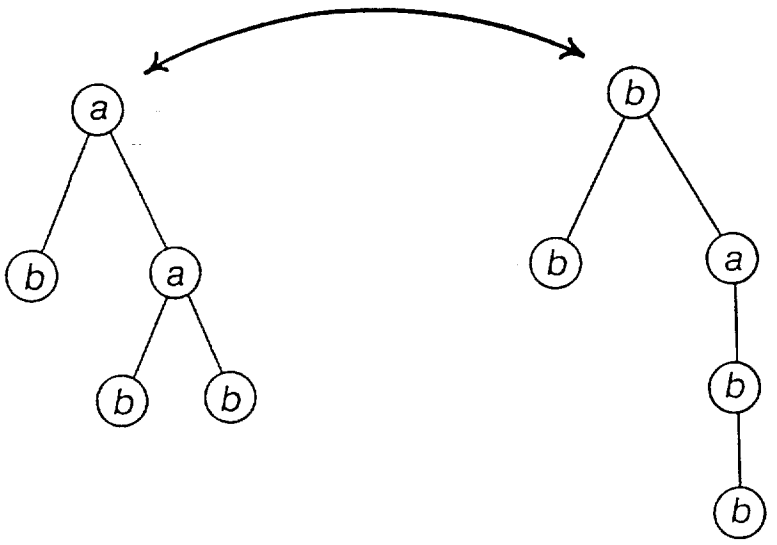


Figure 2b. Disconnecting subtrees.

Pair of Corresponding Subtrees:



Resulting String:

a-2	b-0	a-2	b-0	b-0	b-2	b-0	a-1	b-1	b-0

(Numbers indicate arity, *i.e.* number of children, of node)

Figure 3. Converting Subtree Pairs to Strings