RAPID IDENTIFICATION OF REPEATED PATTERNS IN STRINGS, TREES AND ARRAYS

by

Richard M. Karp*
Raymond E. Miller
Arnold L. Rosenberg

Mathematical Sciences Department
IBM T. J. Watson Research Center
Yorktown Heights, N. Y.

INTRODUCTION:

Many computational processes involve the detection of repeated patterns within a structure such as a string, tree or array. Parsing algorithms usually contain procedures for detecting matches in certain portions of the input string. Similarly, in the construction of symbol tables, matching entries are checked for. Then again, the optimization process in a compiler involves a search for repeated expressions. In processes which modify large files or data bases a check is often made to see if a given item is present, or to find all items in the file that contain a given item. When two lists of items are to be combined it is often desired that items with the same keys be placed together but that no other order of storing keys is required. Finally, it may be possible to compress the amount of storage required to store a structure with many repeated patterns by storing one copy of each pattern together with the locations of the pattern within the structure rather than storing the complete structure. In all of these problems a part of the process involves the detection of repeated patterns.

In this paper we look at a number of matching problems and devise general techniques for attacking such problems. In particular, we describe a strategy for constructing efficient algorithms for solving two types of matching problems. We use this strategy to develop explicit algorithms for these two problems applied to strings (where the patterns are substrings) and arrays (where the patterns are subarrays or blocks). We also develop algorithms for these and related problems for trees, where the patterns are subtrees. Certain special cases of these algorithms are also discussed.

Although we do not claim that these algorithms are optimal, we analyze each algorithm to estimate its computational cost. This provides some basis for choosing which algorithm is most desirable in any given situation.

I. THE TWO MATCHING PROBLEMS:

If we call the object (string, tree or array) we are dealing with a structure S, then the types of structures we are interested in are labeled, directed and oriented. The labels in the string or array, or on the nodes of the tree, can be assumed to be elements of some alphabet $\Sigma = \{\sigma_1, \sigma_2 \cdots, \sigma_s\}$. The repeated patterns are then substructures which can be specified by a single nonnegative integer parameter d which we call the depth of the substructure. (E.g., length d substrings, $d \times d$ subarrays, and depth d subtrees.) A related problem for strings is treated in [1].

Problem 1: Depth d Matches:

Find all depth d substructures of S which occur at least twice in S (possibly overlapping), and find the position in S of each such repeated substructure.

Problem 2: Maximum Matches:

Find the maximum depth D for which S has a repeated depth D substructure and solve Problem 1 for D.

In these problem statements the meanings of a structure S and a substructure of S are not precise. We now specify these meanings for the structures of interest here. For a string $x_1 x_2 \cdots x_n$ the direction and orientation are simply an ordering from the beginning to the end, and the positions in the string are denoted by the indices $1, 2, \cdots, n$. A substructure of depth d is a substring $x_i x_{i+1} \cdots x_{i+d-1}$ of length d. For a two dimensional array of size $n \times n$

$$\begin{array}{llll}
x_{n1} & x_{n2} & \cdots & x_{nn} \\
\vdots & \vdots & & \vdots \\
x_{21} & x_{22} & \cdots & x_{2n} \\
x_{11} & x_{12} & \cdots & x_{1n}
\end{array}$$

*IBM Research Consultant, Professor Computer Science Dept., University of California at Berkeley the direction and orientation are along the coordinate directions in increasing order of indices. The positions in the array are denoted by the ordered pairs of indices $\langle i,j \rangle$ $1 \leq i,\ j \leq n$. A substructure of depth d is a $d \times d$ block.

$$\begin{matrix} x_{i+d-1,j} & x_{i+d-1,j+1} & \cdots & x_{i+d-1,j+d-1} \\ \vdots & \vdots & & \vdots \\ x_{i+1,j} & x_{i+1,j+1} & \cdots & x_{i+1,j+d-1} \\ x_{i,j} & x_{ij+1} & \cdots & x_{i,j+d-1} \end{matrix}$$

This is readily generalized to higher dimensional arrays. Problem 1 for arrays can also be generalized to treat nonsquare blocks. In this paper, however, we restrict our attention to the case in which the size can be specified by a single depth parameter. The trees we consider are rooted, binary and oriented; the nodes may also be labeled by elements of Σ:

A <u>tree</u> T is specified by:

1. a finite set N (the <u>nodes</u>)
2. an element $n_0 \in N$ (the <u>root</u>)
3. partial functions $L: N \to N - \{n_0\}$
   $R: N \to N - \{n_0\}$

   and a function $F: N - \{n_0\} \to N$
   such that,

   (a) if $y = L(x)$ or $y = R(x)$ then $x = F(y)$;
   (b) for all x, and all positive integers k, $F^k(x) \neq x$. (Here $F^k(x)$ denotes the result of iterating F k times.)

If $L(x)$ and $R(x)$ are both undefined then x is a <u>leaf.</u> If there is an integer k such that for every leaf x $F^k(x) = n_0$ then the tree is a <u>full tree.</u> A depth d substructure of a tree T is a full subtree with $k = d$. Some simple examples illustrate matching problems on these structures.

<u>Example 1:</u> String: 001100101

Here $\Sigma = \{0,1\}$. For $d = 3$ there are repeated patterns 001 starting in positions 1 and 5 and no repeated patterns for $d = 4$.

<u>Example 2</u>: Here $\Sigma = \{a,b,c\}$

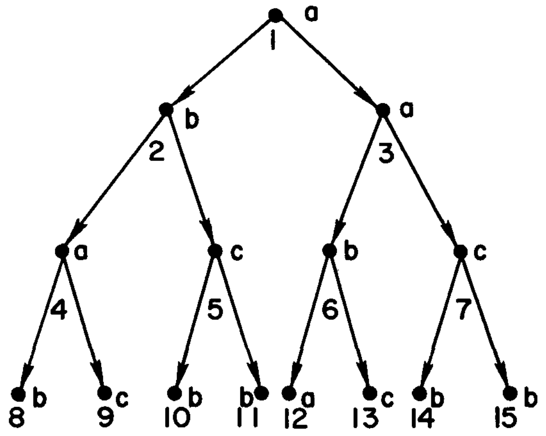
Array:
$$\begin{matrix} c & c & b & c & c \\ b & b & a & b & b \\ b & c & c & b & c \\ a & b & b & a & b \end{matrix}$$

A recurring 2×2 pattern: $\begin{matrix} b & c \\ a & b \end{matrix}$

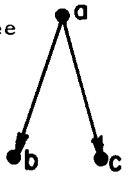
Its occurrences: $\langle 1,1 \rangle$, $\langle 1,4 \rangle$ and $\langle 3,3 \rangle$.

There are some other repeated 2×2 patterns but no repeated 3×3 patterns as the reader may check.

<u>Example 3:</u> Tree:

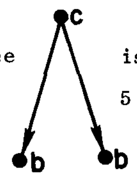


Here $\Sigma = \{a, b, c\}$, and the subtree



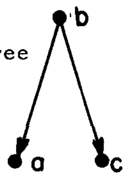
is repeated with roots at nodes 3 and 4.

Subtree



is repeated with roots at nodes 5 and 7.

Subtree



is repeated with roots at nodes 2 and 6. No subtrees of depth 3 are repeated.

II. FINDING REPEATED SUBSTRINGS IN STRINGS:

In solving Problems 1 and 2 for strings we accept as input any finite string S over the given alphabet Σ. We give two techniques for solving Problem 1; one of which also yields a solution to Problem 2. Throughout our description substrings are specified by length and by the position in S where they begin.

A: A Family of Equivalence Relations:

Our first solution technique employs a family of equivalence relations on the set of positions $\{1,2,\ldots,n\}$ of the input string $S = x_1x_2 \cdots x_n$ (each $x_i \in \Sigma$).

Definition 1 Given $S = x_1x_2 \cdots x_n$ positions i and j of S are k-equivalent ($k\in\{1,2,\ldots, n\}$ and $i,j \in \{1,2,\ldots,n-k+1\}$) -- written iE_kj -- precisely when the length k substrings of S starting at positions i and j are identical; that is, when $x_i \cdots x_{i+k-1} = x_j \cdots x_{j+k-1}$.

Henceforth, we assume iE_kj is well defined when used without further mention of the boundary conditions.

B: String Algorithms:

Our algorithms are based on the following observation.

Lemma 1: For integers i,j,a,b with $b \leq a$ we have $iE_{a+b}j$ precisely when iE_aj and $i+bE_aj+b$.*

Our first algorithm is based on Lemma 1.

Algorithm 1: To solve Problem 1 for length d substrings over string $S = x_1x_2 \cdots x_n$:
(1) Scan S to construct the relation E_1.
(2) Use Lemma 1 to construct, successively, the relations $E_2, E_4, E_8, \cdots, E_r$ where $r = 2^{\lfloor \log_2 d \rfloor}$.†
If d is a power of 2 then end: d=r; otherwise $d < 2r$.
(3) Use Lemma 1 to construct the relation E_d from the relation E_r.

An obvious modification of Algorithm 1 yields a solution to Problem 2.

Algorithm 2: To solve Problem 2 for input string $S = x_1x_2 \cdots x_n$:
(1) Scan S to construct the relation E_1.
(2) Use Lemma 1 to construct, successively, the relations $E_2, E_4, E_8, \cdots, E_r$, where r is the least power of 2 such that E_r is trivial (i.e., the identity relation).
(3) Perform a binary search to solve the problem. That is, use $E_{r/2}$ to get $E_{3r/4}$. If this latter is nontrivial use it to get $E_{7r/8}$; otherwise, use $E_{r/2}$ to get $E_{5r/8}$; and so on.

It should be clear that the final equivalence relation constructed by each algorithm solves the desired problem. In fact, the equivalence relation E_k classifies each substring of length k, whether it is a repeated substring or not.

It is a straight forward matter to verify that Algorithm 1 requires $\lceil \log_2 d \rceil$§ "constructions" of relations, while Algorithm 2 requires, at worst, $2\lfloor \log_2 n \rfloor$ such constructions. We claim that each such construction can be performed in order n "steps".¶ (In fact, the reader will see from our descriptions that constructing the relation E_i from its predecessor requires order n-i steps.)

C: Equivalence Relation Construction:

We organize the data representation for the classes of each equivalence relation E_k as follows. The classes of E_k are labeled 1, 2, 3, etc.; of course, at most $\underline{\min}\,(n, (\#\Sigma)^k)$ labels are needed. The relation E_k is represented as an n-k+1 place vector $v_1^{(k)}, v_2^{(k)}, \ldots v_{n-k+1}^{(k)}$, with each $v_i^{(k)}$ being the label of the equivalence class of E_k to which position i belongs.

Assume we are to construct E_{a+b} from E_a using the above representation. We use a scheme reminiscent of a radix sort. Say that E_a has e_a classes. Assume that we have at our disposal the vector $v^{(a)}$ stored as an indexed array; and further we have $2e_a$ pushdown stores, initially empty,

* An alternate equivalent lemma can be stated:
Lemma: For integers i,j,a,b we have $iE_{a+b}j$ precisely when iE_aj and $i+a\,E_bj+a$.

Lemma 1 is the preferred form to use in the algorithms to be derived since they require less storage space at no extra cost in operation count than analogous algorithms derived from the alternate lemma. This may be easily checked by the reader.

† $\lfloor q \rfloor$ denotes the integer part of q.

§ $\lceil q \rceil$ denotes the least integer greater than or equal to q.

¶ This claim is transparent for the construction of E_1, so we consider only the construction of higher index relations.

available; call them $P(1), \cdots P(e_a)$ and $Q(1), \cdots Q(e_a)$.

STEP 1: Sort the vector $v^{(a)}$ using the P-pushdown stores; that is, run through $v^{(a)}$, PUSHing index i onto $P(v_i^{(a)})$. This gives us an explicit representation of the classes of E_a, one class in each P-pushdown store.

STEP 2: In succession, POP each $P(i)$ until it is empty. As the number d is POPped from $P(i)$, PUSH it onto the Q-pushdown store $Q(v_{d+b}^{(a)})$ providing that $d+b \leq n-a+1$. This gives us the classes of E_a shifted so that integers d and e occur on the same Q-stack precisely when $d+b \; E_a \; e+b$. Note, moreover, that the entries on each stack $Q(i)$ appear in blocks of a-equivalent numbers.

STEP 3: Finally construct $v^{(a+b)}$: Successively POP each Q-stack until empty. Start with a variable class counter c initially set to 1. As each integer d is POPped from a given stack $Q(i)$ test whether or not $v_d^{(a)}$ is equal to $v_e^{(a)}$, where e is the integer just previously POPped from the same stack. If this is so, then $d \, E_a \, e$ and $d+b \, E_a \, e+b$ so $d \, E_{a+b} \, e$; therefore set $v_d^{a+b} = c$. Otherwise, we have $d+b \, E_a \, e+b$ but $d \not{E}_a \, e$; therefore, set $v_d^{(a+b)} = c+1$, and increment c to $c+1$. When stack $Q(i)$ is exhausted, increment c to $c+1$ before beginning to POP $Q(i+1)$. Whenever d is the first integer from a Q stack, set $v_d^{a+b} = c$ automatically.

The reader can easily verify the validity of the preceding procedure, hence of our claim that each relation construction can be effected in order n steps; of course, a "step" can involve accessing a linear array or POPping or PUSHing any of the required stacks.

D: Direct Calculation of d-Equivalence:

The solution to Problem 1 given by Algorithm 1 has the shortcoming of requiring one to save relation E_a where $a = \underline{\max} \{k \mid 2^k \leq d\}$, d being the desired length. We describe now a method for calculating d-equivalence directly. Moreover, if both d and the alphabet size s are sufficiently small, one can justifiably claim this method to operate in linear time.

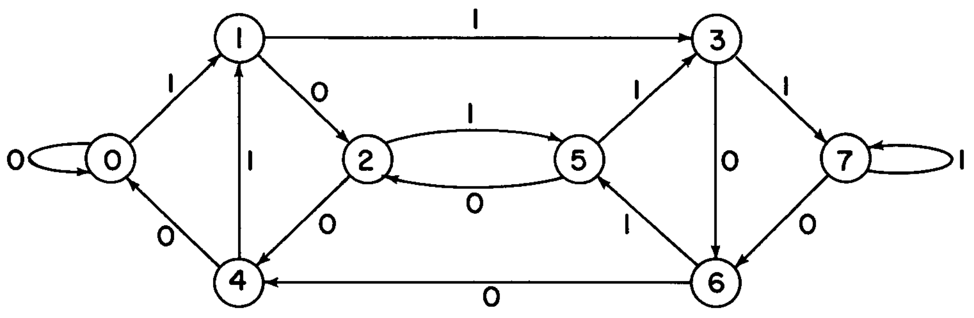
The method employs a class of labeled directed graphs which we shall designate (s, d)-graphs,* s being the size of the alphabet Σ, and d the length of the repetitions sought.

Definition 2 For given integers s, d, the (s, d)-graph $G_{s,d}$ is specified by:

(1) the set of nodes $V_{s,d} = \{0, 1, \cdots, s^d - 1\}$;

(2) for $k \in \{0, 1, \cdots, s-1\}$, the k-labeled edges $\{\langle v, \; s \cdot v + k (\mathrm{mod}\; s^d)\rangle \mid v \in V_{s,d}\}$

As an example, $G_{2,3}$ appears as follows:



A brief consideration of the (s, d)-graphs leads one to the following solution to Problem 1.

Algorithm 3: To solve Problem 1 for length d and for input string $S = x_1 \, x_2 \cdots x_n$ over alphabet $\Sigma = \{\sigma_1, \sigma_2, \cdots, \sigma_s\}$.

Conventions: Imagine that we have, associated with each node v of $G_{s,d}$, a bucket B_v capable of containing any finite set of integers. In the course of visiting nodes of $G_{s,d}$ and of scanning S, let us denote by N the node most recently visited and by P the position in S most recently scanned. Further, assign to each $\sigma \in \Sigma$ a unique "value" denoted $|\sigma|$, from the set $\{0, \cdots, s-1\}$.

Procedure:

(1) Scan the first d symbols $x_1 x_2 \cdots x_d$ of S. (Halt if $n < d$.) Determine the "start-node" of S in $G_{s,d}$ to be node

$$N = \sum_{k=1}^{d} |x_k| \cdot s^{d-k}.$$

Set P to d.

* These graphs are also called DeBruijn graphs.

(2) Drop P-d+1 into bucket B_N.

(3) Scan symbol P+1 of S, call it x_{P+1}, if it exists. If $n-d < P+1$, then halt. Otherwise, respecify P by P+1 and N by $(s\cdot N + |x_{P+1}|)(\bmod s^d)$. Go to Step (2).

Upon halting, each bucket B_v will contain the starting positions in S of the string S(v), where S(v) is that substring of S which under the value function yields the length d s-ary representation of integer v.

It is clear that two options emerge in implementing this algorithm. One can opt to conserve space by constructing on the fly those nodes of $G_{s,d}$ which are actually visited in processing S. The cost of a transition under this option is a computation of the form $s\cdot N+|x| \pmod{s^d}$. Alternatively, one can "precondition" the computation by constructing a transition table for $G_{s,d}$ and use this table to effect transitions. At a cost of s^d locations, transitions are thereby simplified. If the algorithm is to be performed on a large number of inputs, the preconditioned version is by far the preferred one.

E: Special Cases:

One can easily construct variants of Problems 1 and 2, which are solved by simple variants of our three algorithms. We mention two such variants.

Problem 3: Given strings S and T over Σ find all occurrences of S in T.

This problem can be viewed as finding the $E_{\text{length}(S)}$-class containing 1 of the string S \$ T, where $\$ \notin \Sigma$. Either algorithm 1 or 3 thus yields a solution.

Problem 4: To decide, given a length d and a string S, whether or not S has a length d repeated pattern.

This problem is most easily solved by a variant of Algorithm 3 which, in place of buckets, drops pebbles at nodes of $G_{s,d}$ as they are visited. Thus one merely traverses $G_{s,d}$ as in the algorithm, halting with a positive response if a pebble-laden node is ever visited (i.e. if any node is ever revisited). A variant of Algorithm 1 is also possible, to determine whether d-equivalence is, or is not, trivial.

III. FINDING REPEATED BLOCKS IN ARRAYS:

In solving Problems 1 and 2 for arrays we accept as input any finite $n\times n$ array S having entries from the given alphabet Σ. The generalization to higher dimensional arrays should be obvious from our description of the two dimensional case. We look for repeated square $d\times d$ blocks (or subarrays) in S.

Our algorithms are based on the equivalence relation technique described in the previous section. Throughout our description a $d\times d$ block is specified by its size d and its starting position (lower left corner) $\langle i,j\rangle$ in S.

Definition 3 Given an $n\times n$ array S, $S=[x_{ij}]$ $(i,j\in\{1,2,\cdots,n\})$ we say that positions $\langle i,j\rangle$ and $\langle k,\ell\rangle$ are b-equivalent $(b\in\{1,2,\cdots,n\}$ and $i,j,k,\ell\in\{1,2,\cdots,n-b+1\})$ -- written as $\langle i,j\rangle E_b \langle k,\ell\rangle$ -- precisely when the $b\times b$ blocks of S having starting positions $\langle i,j\rangle$ and $\langle k,\ell\rangle$ are identical; that is, when

$$x_{mn} = x_{pq} \qquad \text{for} \qquad \begin{array}{l} m\in\{i,i+1,\cdots,i+b-1\} \\ n\in\{j,j+1,\cdots,j+b-1\} \\ p\in\{k,k+1,\cdots,k+b-1\} \\ q\in\{\ell,\ell+1,\cdots,\ell+b-1\}. \end{array}$$

Whenever $\langle i,j\rangle E_b \langle k,\ell\rangle$ is used later in this section we shall assume it is well defined without stating the end conditions.

A: Array Algorithms :

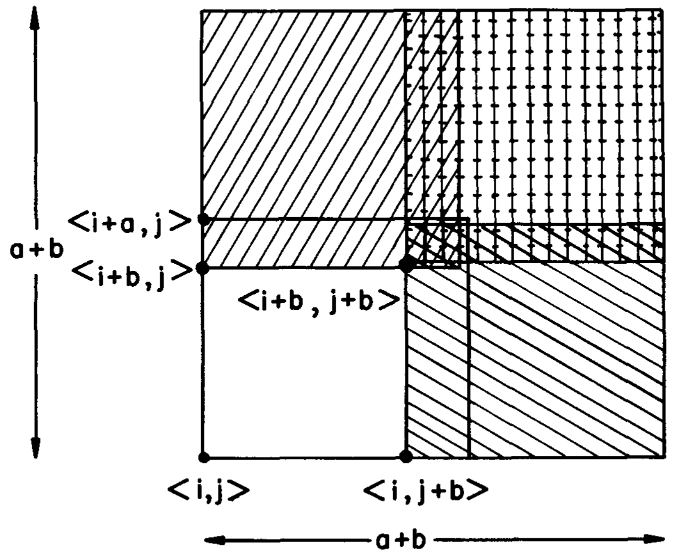
Our algorithms are based on the following lemma.

Lemma 2: For integers i, j, k, ℓ, a, b with $b\le a$ we have $\langle i,j\rangle E_{a+b}\langle k,\ell\rangle$ precisely when all four of the following a-equivalences hold.

(i) $\langle i,j\rangle E_a \langle k,\ell\rangle$ (iii) $\langle i,j+b\rangle E_a \langle k,\ell+b\rangle$

(ii) $\langle i+b,j\rangle E_a \langle k+b,\ell\rangle$ (iv) $\langle i+b,j+b\rangle E_a \langle k+b,\ell+b\rangle$

This result can be readily seen through the following diagram.



With the condition that $a \geq b$ the four $a \times a$ blocks completely "cover" the $(a+b) \times (a+b)$ block so that equivalence of the pieces implies equivalence of the whole block. Of course, alternative "covering" lemmas should readily occur to the reader.

With this lemma we can solve Problems 1 and 2 in exactly the same manner as done for strings in Algorithms 1 and 2. The constructions of new equivalence relations is done as before except that to get the next relation a successive refinement over four relations is required rather than over two relations.

Let the equivalence relations indicated by conditions (i) through (iv) of Lemma 2 be called R_1, R_2, R_3 and R_4, respectively, and be represented like we previously discussed for strings. To construct the new relation E_{a+b} we first combine R_1 with R_2 giving an equivalence relation R'. Next we combine R' with R_3 giving R''; finally combining R'' with R_4 gives E_{a+b}.

Thus, the algorithm to solve Problem 1 is readily seen to require $3\lceil \log_2 d \rceil$ combinations of relations, and the algorithm to solve Problem 2 requires at most $6\lfloor \log_2 n \rfloor$ such combinations. From the method of combining relations we see that the combination of two relations to form a relation E_a requires order $(n-a)^2$ steps.

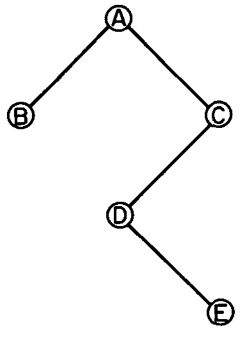
IV. TREES:

A: Unlabeled Trees :

In this section we generalize the results of Section II by investigating the identification and classification of repeated occurrences of a given tree as a subtree of a larger tree. The trees we consider are rooted, binary and oriented.

We retain the notational conventions of Section I; hence a tree is specified by the set N, the element $n_0 \epsilon$ N, and the functions L, R, and F.

Example :



$N = \{A, B, C, D, E\}$ $n_0 = A$

x	L(x)	R(x)	F(x)
A	B	C	-
B	-	-	A
C	D	-	A
D	-	E	C
E	-	-	D

L(x) is called the left son of x, R(x), the right son of x, and F(x), the father of x. If $y = L(x)$ for some x then y is a left son; if $y = R(x)$ for some x then y is a right son. Every node except n_0 is either a left son or a right son. If L(x) and R(x) are both undefined then x is a leaf. If $x = F^k(y)$

then x is a descendant of y and y is an ancestor of x. Every node is a descendant of n_0. If $n_0 = F^k(x)$ then k is the depth of x, denoted by $\delta(x)$.

We are concerned with strings of 0's and 1's. The symbol Λ denotes the null sequence and $\cdot$ denotes concatenation.($\cdot$ is often omitted when no confusion can result.) If x is a string then $\ell g(x)$ denotes the length of x, x^R denotes the reversal of x, and ${}_\ell x$ denotes the string

$${}_\ell x = \begin{cases} x \text{ if } \ell g(x) \leq \ell \\ \text{the prefix of } x \text{ of length } \ell \text{ if } \ell g(x) > \ell. \end{cases}$$

We associate with each node x a string $s(x)$ indicating how x is reached from the root n_0; $s(x)$ is called the sequence of x. The recursive definition of $s(x)$ is as follows:

$$s(n_0) = \Lambda \qquad s(L(x)) = 0\cdot s(x) \qquad s(R(x)) = 1\cdot s(x).$$

The equivalence relation E_ℓ on N is defined by:

$$x E_\ell y \quad \text{if} \quad {}_\ell[s(x)] = {}_\ell[s(y)].$$

We assume that T is represented by a data structure which permits $F(x)$, $L(x)$ or $R(x)$ each to be computed in one step. Clearly, a variant of Algorithm 1 in Section II can be used to compute the relation E_ℓ in $O(\#(N) \log \ell)$ steps.

The unique tree U_k which has 2^k leaves, each of depth k, is called the full tree of depth k. A tree with exactly one leaf is called a path.

In order to define the expected execution time of certain algorithms we shall wish to consider trees drawn at random from a probability distribution over the set of all trees with a given number of nodes. We do not define a specific distribution, but we do assume that two trees which have the same structure but differ in their left-right orientation (i.e., they have the same root and set of nodes, and the same function F) are equally probable. This implies that, if a node x of depth k is chosen at random in a random tree, then the probability that $s(x)$ is a particular string of k 0's and 1's is 2^{-k}.

We often deal with several trees simultaneously; entities associated with these trees will be distinguished by primes.

The trees T and T' are called isomorphic if there is a one-one correspondence

$$\varphi: N \to N' \text{ such that, for all } x \in N, \ s(x) = s'(\varphi(x)).$$

The tree T' is called a subtree of T rooted at y if:

(a) $N' \subseteq N$ (b) for each $x \in N'$, $s(x) = s'(x) \cdot s(y)$

A subtree of T is determined by its set of nodes. No two subtrees of T with the same root are isomorphic.

In this section we construct algorithms to solve the following problem:

Problem 5: Given trees T and T', find $R(T, T')$, the set of roots of subtrees of T which are isomorphic to T'.

First we consider two special cases.

Case 1: $T' = U_k$

The following simple algorithm solves Problem 5: Examine the nodes of T ~~in~~ such an order that, for every x, x is examined before $F(x)$ is examined. Assign each node x a number $N_k(x)$ as follows:

If $L(x)$ is undefined or $R(x)$ is undefined then $N_k(x) = 0$, else $N_k(x) = \underline{\min}(k, N_k(L(x))+1, N_k(R(x))+1)$.

Then $R(T, T') = \{x \mid N_k(x) = k\}$.

Case 2: T' is a path with leaf z of depth ℓ. Then apply the following algorithm.

(1) Combine T and T' into a single tree T'', such that n_0 and n'_0, the roots of T and T', are the left and right sons of n''_0, the root of T''.

Specifically, $T'' = (N \cup N' \cup \{n''_0\}, n''_0, L'', R'', F'')$

where
$$L''(x) = \begin{cases} n_0, & x = n''_0 \\ L(x), & x \in N \\ L'(x), & x \in N', \end{cases} \quad R''(x) = \begin{cases} n'_0, & x = n''_0 \\ R(x), & x \in N \\ R'(x), & x \in N', \end{cases} \quad F''(x) = \begin{cases} n''_0, & x \in \{n_0, n'_0\} \\ F(x), & x \in N - \{n_0\} \\ F'(x), & x \in N' - \{n'_0\}. \end{cases}$$

(2) Compute the equivalence relation E''_ℓ over N''.

(3) $R(T, T') = \{F^{\ell}(x) \mid x \in N \text{ and } x E''_{\ell} z\}$.

Hence, in this case, Problem 5 can be solved in $O((\#(N)+\ell) \log \ell)$ steps.

The key to our general algorithms for solving Problem 5 is the following observation.

Lemma 3: $x \in R(T, T')$ if and only if, for every leaf $y \in N'$, there exists a $z \in N$ such that

$$F^{\delta'(y)}(z) = x \quad \text{and} \quad {}_{\delta'(y)}[s(z)] = s'(y)$$

Let $A' = \{s'(y) \mid y \text{ is a leaf of } T'\}$.

For each $x \in N$ define $C(x) = \#(\{(z,k) \mid F^k(z) = x \text{ and } {}_k[s(z)] \in A'\})$.

Then the following is a restatement of Lemma 3.

Lemma 4: $x \in R(T, T')$ if and only if $C(x) = \#(A')$.

Thus our central problem is to give an efficient algorithm for computing the function $C(x)$.

We offer two algorithms. Let $\bar{\ell}$ denote the maximum depth of any leaf of T'. The first algorithm is appropriate when $2^{\bar{\ell}} << \#(N)$.

Algorithm 4:

(1) (Preprocessing) For each of the $2^{\bar{\ell}+1}-1$ strings X of length less than or equal to $\bar{\ell}$, compute

$$M(X) = \{k \mid k \leq \ell g(X) \text{ and } {}_kX \in A'\}.$$

(2) Compute the equivalence relation $E_{\bar{\ell}}$ over N. For each equivalence class, choose a representative w and compute $M({}_{\bar{\ell}}\, s(w)) = \mu(w)$.

(3) For each $z \in N$, set $C(z) = 0$.

(4) For each $z \in N$, let w be the representative of the equivalence class containing z; for each $k \in \mu(w)$, increment $C(F^k(x))$ by 1.

(5) Stop. $x \in R(T, T')$ if and only if $C(x) = \#(A')$.

The execution of Algorithm 4 requires $O(2^{\bar{\ell}} + \#(N) \log \bar{\ell} + \sum_x C(x))$ steps. Below we obtain an upper bound on $\sum_x C(x)$ and also an upper bound on the expected value of $\sum_x C(x)$, assuming that T is drawn at random from trees with $\#(N)$ nodes.

Our second algorithm for computing $R(T, T')$ depends on an important construction. We can construct from T' a unique tree $\tilde{T}$ such that there is an "anti-isomorphism" from T' onto $\tilde{T}$; i. e., there is a one-one function φ from N' onto $\tilde{N}$ such that, for each $x \in N'$, $\tilde{s}(\varphi(x)) = [s'(x)]^R$.

Figure 1 gives two examples of this construction.

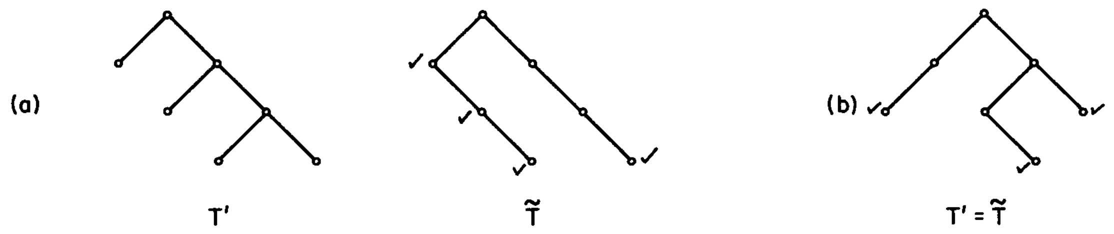


Figure 1 -- Two examples of an anti-isomorphism

Each node $\varphi(x) \in \tilde{N}$ such that x is a leaf of T' is marked. In the figure a check mark is placed beside each marked node.

The following algorithm determines $R(T, T')$.

Algorithm 5:

(1) (Preprocessing) Construct $\tilde{T}$ and determine the marked nodes.

(2) Examine the nodes of $\tilde{T}$ in such an order that, for every x, $F(x)$ is examined before x. Associate with each node x a set $P(x) \subseteq N$ as follows:

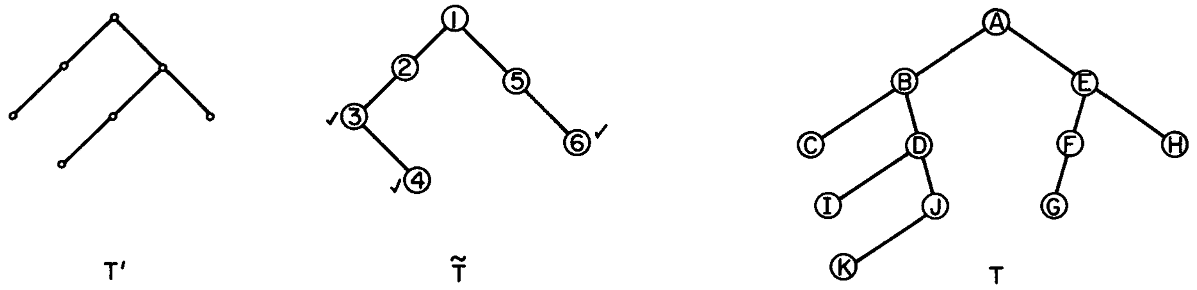
$$P(\tilde{n}_0) = N$$

$$P(\tilde{L}(x)) = \{F(y) \mid y \in P(x) \text{ and } y \text{ is a left son}\}.$$

$P(\tilde{R}(x)) = \{F(y) \mid y \in P(x) \text{ and } y \text{ is a right son}\}.$

(3) STOP. $R(T,T') = \bigcap_{\{x \mid x \text{ is marked}\}} P(x)$

Example 4:



$P(1)$	$= \{A, B, C, D, E, F, G, H, I, J, K\}$	$P(3)$	$= \{A, E\}$
$P(2)$	$= \{A, B, E, F, D, J\}$	$P(4)$	$= \{A\}$
$P(5)$	$= \{B, A, E, D\}$	$P(6)$	$= \{A, B\}$

$$R(T, T') = P(3) \cap P(4) \cap P(6) = \{A\}.$$

Let $\lambda(T)$ denote the number of leaves of T.

Theorem 1: Suppose T is chosen at random from the trees with $\#(N)$ leaves. Then

i) The expected value of $\Sigma\#(P(x))$ is $\leq \#(N) \cdot \sum_{n \in \tilde{N}} 2^{-\tilde{\delta}(x)}$

ii) The expected value of $\sum_{x \in N} C(x)$ is $\leq \#(N) \cdot \sum_{x \text{ a leaf of } T'} 2^{-\delta'(x)}$

iii) The expected value of $\#(R(T,T'))$ is $\leq [\#(N) \cdot \sum_{x \text{ a leaf of } T'} 2^{-\delta'(x)}] \div \lambda(T')$

These results permit an easy evaluation of the expected execution times of Algorithm 4 and Algorithm 5.

It is also possible to place an upper bound on the number of occurrences in T of subtrees isomorphic to T', knowing only the number of nodes of T. For each node $x \in N'$, let $\Delta(x)$ denote the number of descendants of x (recall that x is a descendant of itself). Define

$$\theta(T') = 1 + \max_{x \in N'} [\underline{\min} (\Delta(L'(x)), \Delta(R'(x))],$$

where $\Delta(L'(x)) = 0$ if $L'(x)$ is undefined, and $\Delta(R'(x)) = 0$ if $R'(x)$ is undefined.

Theorem 2: $\#(N) \geq \#(N') + (R(T,T')-1) \ \underline{\max} \ (\theta(T'), \lambda(\tilde{T}))$

Continuing Example 4, we have: $\theta(T') = 3, \ \lambda(\tilde{T}) = 2, \ \#(N') = 7.$

Thus $\#(N) \geq 7 + 3(R(T,T')-1) = 4 + 3R(T,T')$.

Any tree with at least two subtrees isomorphic to T' must have at least 10 nodes.

B: Labeled Trees:

We now consider trees in which each node is labeled with one of a finite set of values. A labeled tree is a pair (T,f). such that $T = (N, n_0, L, R, F)$ is a tree and f is a (total) function with domain N. The labeled trees (T',f') and (T'', f'') are isomorphic if the trees T' and T'' are isomorphic and, moreover, the isomorphism $\varphi: N' \to N''$ between these trees preserves the labeling; i.e., for all $x \in N'$, $f'(x) = f''(\varphi(x))$.

If (T,f) is a labeled tree and (T',f') is a labeled tree such that T' is a subtree of T and, for all $x \in N'$, $f'(x) = f(x)$, then (T',f') is a labeled subtree of (T,f). The labeling function f' is said to be inherited from (T,f).

We seek algorithms to solve the following two problems.

Problem 6: Given the labeled trees (T',f') and (T'',f''), determine $R((T',f'),(T'',f''))$, the set of roots of the labeled subtrees of (T',f') which are isomorphic to (T'',f'').

Problem 7: Given the labeled tree (T,f) and the tree T', find all the subtrees of T isomorphic to T', and classify them into their isomorphism types as labeled trees. A more precise statement of the requirement is as follows: for $x \in R(T,T')$, let $T^{(x)}$ be the subtree rooted at x and isomorphic to T', and let $f^{(x)}$ be the labeling that $T^{(x)}$ inherits from f. Compute a function $h_{T,f,T'}$, with domain $R(T,T')$, such that $h_{T,f,T'}(x) = h_{T,f,T'}(y)$ if and only if $(T^{(x)},f^{(x)})$ is isomorphic to $(T^{(y)},f^{(y)})$.

When the identities of T, f and T' are clear from context, we write h instead of $h_{T,f,T'}$.

Before describing the algorithms we propose for the solution of Problems 6 and 7, we require an alternative way of talking about equivalence relations over a finite set D. We consider functions with domain D. All that will be important about such a function f is the equivalence relation it determines over D, in which x and y are equivalent iff $f(x) = f(y)$. Two functions which determine the same equivalence relation (and hence differ only in the names of the elements in their images) will be regarded as identical. Now, given functions f and g, the function $f \times g$ has the following defining property

$(f\times g)(x) = (f\times g)(y)$ if and only if $f(x) = f(y)$ and $g(x) = g(y)$.

The operation '$\times$' is commutative and associative.

The construction of $f\times g$ is essentially the process of taking the common refinement of two equivalence relations, and the algorithm given in Section II performs this process in $O(\#(D))$ steps.

The following is a simple method to solve Problem 6 when T'' is a full tree U_k.

Algorithm 6:

(1) Join (T',f') and (T'',f'') into a single labeled tree (T,f) specified as follows:

i) $T = (N, n_0, L, R, F)$ where $N = N' \cup N'' \cup \{n_0\}$, $n'_0 = L(n_0)$ and $n''_0 = R(n_0)$;

ii) Both (T',f') and (T'',f'') are labeled subtrees of (T,f), and $f(n_0)$ is distinct from all other labels.

For $i = 0, 1$:

(2) Define the labeling functions: $\sigma_i(x) = \begin{cases} 1-i & \text{if } x = n''_0 \\ 0 & \text{is } x \text{ is a left son} \\ 1 & \text{if } x \text{ is a right son} \\ 2 & \text{if } x = n_0 \end{cases}$

The i's take care of matching n''_0 with both left and right sons.

and define $g_i = f \times \sigma_i$.
Consider the labeled trees (T, g_i).

(3) Compute the following functions with domain the set of nodes of depth $> k$:

$$h_i(x) = g_i(x) \times g_i(F(x)) \times g_i(F^2(x)) \times \cdots \times g_i(F^k(x)).$$

Let E_i be the equivalence relation induced by h_i on its domain.

(4) For each equivalence class C_i in the relation E_i, compute $F^k(C_i) = \{F^k(x) \mid x \in C_i\}$.

(5) $R((T',f'),(T'',f'')) = \bigcup_{i\in\{0,1\}} [\bigcap_{\{C \mid n''_0 \in F^k(C)\}} F^k(C)] - \{n''_0\}$.

Most of the work in this algorithm occurs in Step 3; Algorithm 1 given in Section II shows that Step 3 can be performed in $O(\#(N) \log k)$ steps.

We next consider the solution of Problem 7 when both T and T' are full trees. Suppose $T = U_\ell$ and $T' = U_k$. We also assume that the following convenient addressing scheme is used for $T = (N, n_0, L, R, F)$:

$N = \{1, 2, \cdots, 2^{\ell+1} - 1\}$ $\quad n_0 = 1 \quad$ $L(x) = 2x$, $R(x) = 2x+1$, $F(x) = \lfloor \frac{x}{2} \rfloor$.

Note that $R(T,T') = \{x \mid 2^k x \le 2^{\ell+1} - 1\}$.

The idea of the following algorithm is to "push" all the information about node labels in each copy of U_k down to the leaves (Step 1), across to the leftmost leaf (Step 2) and then up to the root (Step 3).

Algorithm 7:

(1) This step computes a sequence of functions $g_1, g_2, \cdots, g_k$. The domain D_j of $g_j = \{z | z$ is a multiple of $2^j\}$. For $j = 1, 2, \cdots, k$

Set $g_j(x) = f(x) \times f(\frac{x}{2})$, $x \in D_j$ Set $f(z) = g_j(z)$, $z \in D_j$

(2) This step computes a sequence of functions $e_1, e_2, \cdots, e_k$. The domain of e_j is D_j. For $j = 1, 2, \cdots, k$

Set $e_j(x) = f(x) \times f(x + 2^{j-1})$, $x \in D_j$. Then set $f(x) = e_j(x)$

(3) Set $h_{U_\ell, f', U_k}(x) = f(2^k x)$, $x \in R(U_\ell, U_k)$.

Algorithm 7 requires $O((2 - \frac{1}{2^k}) \cdot \#(N))$ steps.

Next we consider general approaches to the solution of Problem 7 when T' is an arbitrary tree. The next algorithm is appropriate when R(T, T') is known and has small cardinality.

Algorithm 8:

(1) (Preprocessing) Construct a sequence $x_1, x_2, \cdots, x_p$ of elements of N' such that

(a) $x_1 = n'_0$ (b) each element of N' occurs in the sequence

(c) for each i, $x_{i+1} = L(x_i)$ or $x_{i+1} = R(x_i)$ or $x_{i+1} = F(x_i)$.

Standard methods of traversing trees serve to construct such sequences. The minimum length of such a sequence is $2\#(N') - 1$. Let $i_1, i_2, \cdots, i_n$ be the positions in the sequence where a node occurs for the first time, where $n = \#(N')$.

(2) For each $y \in R(T, T')$:

Let φ be the isomorphism from T' onto a subtree of T rooted at y.

Construct the sequence $y_1, y_2, \cdots, y_p$ such that $y_i = \varphi(x_i)$, $i = 1, 2, \cdots, p$. This can be done by traversing the subtree in the same manner that T' was traversed to produce the sequence $x_1, x_2, \cdots, x_p$; i.e.,

$$y_{i+1} = \begin{cases} L(y_i) & \text{if } x_{i+1} = L(x_i) \\ R(y_i) & \text{if } x_{i+1} = R(x_i) \\ F(y_i) & \text{if } x_{i+1} = F(x_i). \end{cases}$$

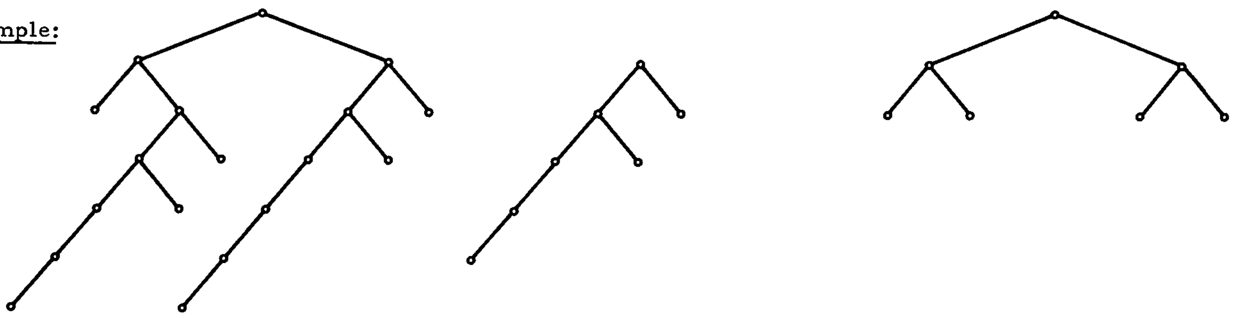
(3) For $y \in R(T, T')$

Set $h_{T,f,T'}(y) = f(y_{i_1}) \times f(y_{i_2}) \times \cdots \times f(y_{i_n})$.

This algorithm can be executed in $O((2\#(N')) - 1) \cdot \#(R(T, T'))$ steps once R(T, T') is known.

Finally, we present a theorem which suggests a recursive approach to the computation of $h_{T,f,T'}$. This approach is especially applicable when T' itself contains a large number of isomorphic copies of a subtree T''. The idea is that we can "push" the information contained in the copies of T'' up to the roots of these copies, and then throw away those nodes in T' whose information has been pushed to such a root, obtaining a smaller tree T''', which may be considered instead of T'.

Example:



T' T'' T'''

Let T'' be a subtree of T', and let (T, f) be a labeled tree. Suppose we compute $h_{T,f,T''}$ and define a new labeling function for T:

$$f'(x) = \begin{cases} f(x) \times h_{T,f,T''}(x) & , \quad x \in R(T, T'') \\ f(x) & , \quad x \notin R(T, T''), \end{cases}$$

where the image of f is assumed to be disjoint from the image of $f \times h_{T,f,T''}$.

Let $\bar{N} = \{x \in N' \mid x$ is an ancestor of some $y \in N'$ such that either $y \in R(T', T'')$ or y does not occur in any subtree of T' isomorphic to $T''\}$.

Let T''' be any subtree of T' whose set of nodes N''' contains $\bar{N}$.

Theorem 3.

$$h_{T,f,T'} = h_{T,f',T'''}$$

The possibilities for recursive application of Theorem 3 should be obvious.

REFERENCE:

1. Morris, James H. and Vaughan Pratt, "A Linear Pattern Matching Algorithm" Report #40, Computing Center, University of California at Berkeley, 1970.