Università di Pisa
Dipartimento di Informatica

Technical Report: TR-04-13

Bottom-up Subtree Isomorphism for Unordered Labeled Trees

Fabrizio Luccio   Antonio Mesa Enriquez
Pablo Olivares Rieumont   Linda Pagli

June 24, 2004

Address: via F. Buonarroti 2, 56127 Pisa, Italy.   Tel: +39 050 2212700   Fax: +39 050 2212726

Bottom-up Subtree Isomorphism for Unordered Labeled Trees *

Fabrizio Luccio † Antonio Mesa Enriquez ‡
Pablo Olivares Rieumont § Linda Pagli ¶

June 24, 2004

Abstract

A subtree P of a given tree T is a *bottom-up subtree* of T if, for each internal vertex u of P, all the children of u in T are also vertices of P. Our problem is deciding if a *pattern tree* P of m vertices is a bottom-up subtree of a *text tree* T of n vertices, $m \leq n$, and, in the affirmative case, finding all the occurrences of P in T. If the vertices are labeled, the labels in the corresponding positions in P and T must match. We consider unordered rooted trees with labeled vertices, which are reconducted to simpler ordered trees through an ordering algorithm. Processing time depends on the label format. If the labels are single characters of a *constant* or of an *n-integer* alphabet Σ (respectively, $|\Sigma|$ = constant, or Σ is composed of integers in the range $[1 \div n]$), the problem is solved in $O(m + \log n)$ time and $\Theta(m)$ additional space, after a preprocessing of T is done in $\Theta(n)$ time and $\Theta(n)$ additional space. The number of occurrences of P in T does not appear in the search time because all such occurrences can be directly reconstructed from a constant output information. For more complex labels the running times slightly increase. In particular if the labels are sequences of characters (e.g. as in XML files) the running time becomes a function of the total length of all the labels in T and P.

Although directed to the "simple problem" of exact subtree matching, our work is the first one to solve it in linear time for unordered trees. Moreover our approach is the one of dictionary search. Regarding T as a static text on which several queries are made, P as the contents of one such a query, and assuming $m = o(n)$, the response time for each P is sublinear in the size of the overall structure.

*This work has been supported in part by a Cooperation Program with the University of Havana sponsored by UNESCO and the Italian Foreign Ministry.

†Dipartimento di Informatica, Università di Pisa, Italy. luccio@di.unipi.it
‡Universidad de la Habana, Cuba. tonymesa@matcom.uh.cu.
§Universidad de la Habana, Cuba. olivares@matcom.uh.cu.
¶Dipartimento di Informatica, Università di Pisa Italy. pagli@di.unipi.it.

Keywords: Rooted tree, Bottom-up subtree, Subtree isomorphism, Tree pattern matching, Design of algorithms.

1 The problem of subtree isomorphism

The problem of subtree matching has to do with the detection of the occurrences of a *pattern tree* P of m vertices as a subtree of a *text tree* T of n vertices, $m \leq n$. The literature on this problem is rather abundant and sometimes confusing in the problem definition itself. A recent good reference is the book of Valiente [17] from which we will inherit some terminology.

A basic distinction is between rooted and non-rooted trees, and, in the former family, between ordered and unordered trees (e.g. trees that, for each non-leaf vertex v, have, or do not have, an ordering imposed among the children of v). An exact definition of subtree is crucial. Our work refers to unordered rooted trees, for which we adopt the definitions of [17]. Let $T = (V, E)$ and $S = (W, F)$ be unordered rooted trees, with $W \subseteq V$. If $F \subseteq E$ then S is a *subtree* of T. In particular if $W = V$ and $F = E$ the two trees T and S are *isomorphic*. A stricter definition is the following. For $v \in V$ let *children(v)* be the set of children of v in T (if v is not a leaf). We have:

Definition 1 (From [17]) *S is a* bottom-up unordered subtree *of T if for any vertex $v \in W$ that is not a leaf of T we have children(v) $\subset W$.*

Intuitively if v is the root of a bottom-up subtree S then all the discendants of v in T are also contained in S, although the children of any vertex of S may appear in different orders in T and in S. Definition 1 can be immediately extended to ordered trees where, in each level, corresponding vertices in T and S must appear in the same order. In most papers a bottom-up subtree is simply called a subtree. We will often do this here, as our work refers to bottom-up subtrees only.

Another distinction in the study of subtree matching is between *exact* matching or *isomorphism*, and *approximate* matching. In the former case we look for subtrees S of T which are isomorphic with P. In the other case some edit operations are allowed on P for rendering P and S isomorphic. A distinction is also made between labeled and unlabeled trees. Here we consider the exact matching. The general case is the one of trees with labeled vertices, as edge labels may be assigned to the destination vertices, and unlabeled trees can be seen as labeled trees with all identical labels. As each vertex includes its label as an integral part, all the definitions given above hold unchanged for labeled and unlabeled trees. In particular, in the labeled case two matching vertices of S and P must have the same label. The complexity of the matching problem, however, depends on the type of alphabet used, and if the labels are single characters or strings.

Let us comment briefly on the different versions of the problem. The problem of (non-rooted) tree isomorphism ($m = n$) was implicitly solved in linear time with the famous Hopcroft and Tarjan's algorithm for planar graph isomorphism. Two specific linear algorithms for rooted tree isomorphism were given in [18] and [1]. Subtree isomorphism ($m \leq n$) is more difficult. We have:

1. Subtree isomorphism of rooted trees. After the pioneer work of [15], it was shown in [13] how to solve the bottom-up subtree problem for ordered trees in $\Theta(n)$ time, coding P and T as strings and applying a string matching algorithm to them. In [17] several types of subtrees are considered. In particular bottom-up subtree isomorphism for ordered trees is solved in $O(n)$ time using $O(m)$ additional space; and for unordered trees is solved in $O((n+m)^2)$ time using $O(n+m)$ additional space; or in $O(n+m)$ expected time using $O(n+m)$ additional space. All these works refer to unlabeled trees, but can be also applied if the vertices are labeled with single characters of a constant-size alphabet. Our work applies to bottom-up subtree isomorphism for unordered trees with different types of labels. The problem will be solved quite efficiently, as we shall see.

2. Subtree isomorphism of non-rooted trees. After the original works of [14, 2], a recent paper [16] shows how to solve the problem in $O(n\,m^{1.5}/\log m)$ time.

3. Approximate tree matching. This problem asks for a definition of *edit distance* between trees, as it was cleverly treated in [19] for rooted ordered trees. Many variations of the problem have been then investigated, see for example [11, 5]. Approximate matching is much more difficult than exact matching and will not be treated here.

In the present work we consider unordered rooted trees. First we reduce all trees to the case of ordered ones, then we search subtrees using a string representation of trees together with some advanced data structures. For trees with simple labels we preprocess T in $\Theta(n)$ time and $\Theta(n)$ additional space, then we can make any number of (bottom-up) searches for bottom-up occurrences of different patterns of m vertices in $\Theta(m+\log n)$ time and $\Theta(m)$ additional space for each pattern. For certain families of labels the algorithm takes further steps that may increase its running time. For labels consisting of string of characters, our procedure also includes a new very simple optimal algorithm for sorting strings. Before presenting our technique we have to discuss how to handle different families of labels, and how to reorder a tree. This will be done in the following subsections. Subtree isomorphism will then be treated in section 2.

1.1 About alphabets.

In principle an *alphabet* Σ is a collection of elements or *symbols*. In practice symbols are coded in digital form, that carries a variety of consequences. Different applications more or less implicitly refer to different families of objects treated as symbols, with crucial implications on the time needed for comparing two symbols or two sequences of symbols, sorting a set of symbols or sequences, etc. This is also connected with the model of computation considered. In particular, the commonly adopted

RAM word-model assumes that each symbol can be contained in a "computer word", and two words can be compared in $\Theta(1)$ time [1].

In view of this variety of definitions, we shall now state the characteristics of the alphabets whose symbols will be used as tree vertex labels in our work. We have two major distincions, according to the time needed for symbol comparison.

- **Simple alphabets**. The comparison of any two elements of an alphabet Σ can be done in constant time. We consider three major members of this family:

  1. *Constant alphabet.* We have $|\Sigma| = \Theta(1)$.
  2. *n-integer alphabet.* We have $|\Sigma| \leq n$, where n is a parameter of the problem at hand. The symbols may be coded with integers in the range $[1 \div n]$, or with arbitrary binary strings of length $\leq \log n$. The constant time comparison of two symbols is then consistent with the RAM word-model for computer words of length $\log n$. [1]
  3. *General alphabet.* No further assumption.

- **String-alphabets**. The elements of an alphabet Σ are string σ_i of symbols of a simple alphabet Γ. Two elements $\sigma_1, \sigma_2 \in \Sigma$ can be compared in time $\Theta(\ell)$, where ℓ is the length of the shorter of the two strings. We have:

  4. *Constant string-alphabet.* Γ is a constant alphabet.
  5. *m-integer string-alphabet.* Γ is an m-integer alphabet. Note that the comparison between σ_1 and σ_2 is done in $\Theta(\ell)$ time in the RAM word-model.
  6. *General string-alphabet.* Γ is a general alphabet.

1.2 The AHU algorithm for canonical ordering of trees

In section 3.2 of their seminal book [1], Aho, Hopcroft and Ullman presented an algorithm (called *AHU* here) for deciding whether two unordered unlabeled trees are isomorphic. *AHU* is based on assigning to each level i of the two trees an ordered sequence of integers representing the structures of the subtrees rooted at that level (the given trees are isomorphic if their sequences coincide for all i).

Although not done in [1], the sequence of integers associated to a level of a tree can also be used for reordering the vertices at that level, thereby bringing the tree to an ordered form that we shall call *canonical.* We use this strategy for ordering trees

[1] An n-integer alphabet is often defined as having $|\Sigma| = O(n^s)$, with $s = \Theta(1)$. In this case comparing two symbols requires constant time s in the RAM word-model. Such an extended definition can be adopted without major changes to our theory.

labeled with a constant alphabet. The following Algorithm 1 is the new version of AHU adapted to our needs. [2]

Algorithm 1 *Canonical reordering of a tree T labeled with a constant alphabet Σ (AHU approach).*

tree levels are numbered $0, 1, \ldots h$ starting from the root;
n_i is the number of vertices at level i, with $n_0 = 1$;
$n = \sum_i n_i$ is the total number of vertices of T;
labels are coded with positive integers $\leq |\Sigma|$;

- *Strategy*: proceed bottom-up level by level, starting from level h; assign an integer ≥ 1 to each vertex, such that two vertices at the same level that are roots of isomorphic (bottom-up) subtrees have the same integer (see Lemma 1 below). To do this:

1. assign a two-component tuple $(\lambda, 0)$ to each leaf of T, where λ is the (integer) label of the corresponding leaf;
2. sort the tuples at level h with Radix Sort, and assign consecutive integers to the vertices (leaves) corresponding to the sorted tuples, thus forming an ordered list L_h (the same integer is assigned to vertices with identical tuples);
3. for each level i, $h-1 \geq i \geq 0$ assume inductively that all the vertices at level $i+1$ are numbered with consecutive (possibly repeated) integers in the range $[1 \div n_{i+1}]$, and let L_{i+1} be the ordered list of such integers. Do the following:
   (a) to each non-leaf vertex v at level i associate a tuple t_v of integers as follows: start with tuples containing only the label of v, scan L_{i+1}, and append each of its elements at the end of the tuple of its father. Now reorder the children of each non leaf vertex v following the order specified in t_v (recall that the elements following the label in t_v correspond to such children);
   (b) for $i \neq 0$: sort the set of all the tuples at level i with the tuple-sorting Algorithm 3.2 of [1], and assign consecutive integers $1, 2, \ldots$ to the vertices corresponding to the sorted tuples, thus forming the ordered list L_i (the same integer is assigned to vertices with identical tuples).

As an example consider the tree levels $i+1$ and i shown in figure 1. For clarity labels are shown with letters, so the tuples are sorted lexicographically. We start from $L_{i+1} = 1\ 2\ 2\ 3\ 4\ 4\ 5\ 5\ 6\ 7$. Step 1 of Algorithm 1 has already built the two-component tuples $(e, 0)$, $(b, 0)$, $(e, 0)$ associated with the leaves at level i. Step 3(a) builds the tuples associated with the non-leaf vertices at level i, as shown in the figure. (The children of such vertices are then reordered according to their tuples, e.g., the children of the first vertex to the left are reordered as 1, 5, 6). Step 3(b) sorts the tuples at level i in the order $(a, 4, 4, 7)$, $(b, 0)$,

[2] In [1] it was already suggested to extend AHU for handling labeled trees, by inserting vertex labels into the tuples. The original suggestion, however, needs some changes for being applicable. See also the following footnote 3.

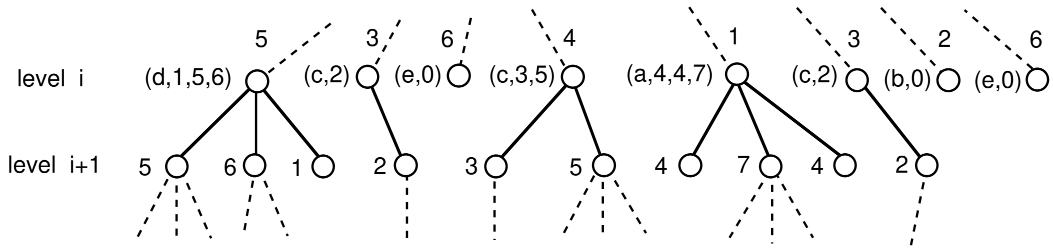


Figure 1: Tuple formation at level i with steps 3(a,b) of Algorithm 1. After tuple ordering, the vertices at level i are numbered in the list $L_i = 1\ 2\ 3\ 3\ 4\ 5\ 6\ 6$. Alphabetic characters represent vertex labels.

$(c, 2)$, $(c, 2)$, $(c, 3, 5)$, $(d, 1, 5, 6)$, $(e, 0)$, $(e, 0)$. Then the tuples are numbered from 1 to 6 (both occurrences of $(c, 2)$ are numbered 3, and both occurrences of $(e, 0)$ are numbered 6), to form the list $L_i = 1\ 2\ 3\ 3\ 4\ 5\ 6\ 6$. These numbers are associated with the vertices as shown in the figure.

As for *AHU* the complexity of Algorithm 1 is $\Theta(n)$. In fact steps 1 and 2 require $\Theta(n)$ time. At each iteration step 3(a) requires $\Theta(n_i + n_{i+1})$ time. For step 3(b), recall that Algorithm 3.2 of [1] sorts a set of tuples of total number of elements ℓ, composed of integers $\leq w$, in time $\Theta(\ell + w)$. Here we have: $\ell = n_i + |L_{i+1}| = n_i + n_{i+1}$ and $w \leq max(n_{i+1}, |\Sigma|)$, then the step takes time $\Theta(n_i + n_{i+1} + |\Sigma|)$. Therefore the total time required by the algorithm is $\Theta(n) + \sum_{i=0}^{h-1} \Theta(n_i + n_{i+1} + |\Sigma|) = \Theta(n)$, as $|\Sigma|$ is a constant.

The correctness of Algorithm 1 for our purposes will be discussed in subsection 1.4.

1.3 Ordering with different label alphabets

As we have seen, unordered trees labeled with a constant alphabet are handled with Algorithm 1 by inserting the label of each vertex v as the first component of the tuple t_v. The same method allows extending Algorithm 1 to build the canonical ordering of trees labeled with different alphabets, however the running time may change. Let the vertex labels be symbols of a **simple alphabet** Σ. We have:

1) Σ is a constant alphabet. As already seen this case is solved in time $\Theta(n)$ with Algorithm 1, with the symbols of Σ coded with positive integers $\leq |\Sigma|$.

2) Σ is an n-integer alphabet. Let the elements of Σ be coded with positive integers $\leq n$. In order to attain $\Theta(n)$ total time the following preprocessing is needed:

Algorithm 2 *Renumbering the labels of a tree T.*

tree levels are numbered 0, 1, ... h starting from the root;
labels are coded with positive integers $\leq n$;

1. traverse T, and for each vertex v build a pair of integers (i, λ) where i is the level of v and λ is the label of v;

2. sort the pairs (i, λ) with Radix Sort to form an ordered sequence S that is a concatenation $S_0 S_1 \ldots S_h$, where each subsequence S_i contains all the pairs with the same value of i, ordered on the value of λ;

3. for $1 \leq i \leq h$ associate to the pairs of S_i, hence to the corresponding vertices, new consecutive integers $\gamma = 1, 2, \ldots n_i$.

- *Comment.* The tree is now ready to be processed with Algorithm 1 using the integers γ as labels.

Algorithm 2 clearly requires $\Theta(n)$ time. After Algorithm 2 is applied the tree is ordered with Algorithm 1 using the integers γ as labels. The tuple sorting algorithm of step 3(b) now deals with $\ell = n_{i+1} + n_i$ integers of maximum value $w \leq max(n_i, n_{i+1})$, hence the total time required by the algorithm is $\sum_{i=0}^{h-1} \Theta(n_i + n_{i+1}) = \Theta(n)$ as before. [3]

3) Σ is a general alphabet. We now have a lower bound of $\Omega(n \log n)$ for canonical ordering, as required for example by a tree with only two levels, with the $n-1$ vertices at level 1 having different labels. The ordering can be attained in $\Theta(n \log n)$ time by sorting the tree labels, renaming them with integers in the range $[1 \div n]$, and then applying Algorithms 2 and 1 as in case 2 above.

Let now the vertex labels be elements of a **string-alphabet** Σ, that is, strings built on a simple alphabet Γ. And let L be the total length of the labels (i.e., the total number of symbols of Γ contained in all the labels). Again we can attain canonical ordering of the tree by assigning to the vertices new integer labels in the range $[1 \div n]$, then applying Algorithms 2 and 1 as in case 2 above. As the these two algorithms require $\Theta(n)$ time, the overall time depends on the previous numbering phase, that in turn is done by previous sorting of the labels. We have:

4) Σ is a constant or an n-integer string-alphabet. Treat the labels as tuples, and sort them with the usual tuple sorting algorithm in $\Theta(L + n) = \Theta(L)$ time. The overall algorithm also requires $\Theta(L)$ time.

5) Σ is a general string-alphabet. Similarly to case 3 above, $\Omega(L + n \log n)$ is a lower bound to the time of sorting the labels, where the additional term L is due to the fact that, in the worst case, all the symbols contained in the labels must be examined. A matching upper bound of $\Theta(L + n \log n)$ can be attained with an extension of Merge-Sort to strings, that we propose in the following Algorithm 3.

[3]The suggestion given in [1] for handling labels was in fact referred to an n-integer alphabet, but does not include the preprocessing indicated here. As a result the tuples at each level contain integers with values up to n, and the total time for tree isomorphism is $O(n \cdot h)$. This was not pointed out in [1] and has caused some incorrect evaluations in successive papers based on *AHU*.

Algorithm 3 *Ordering a set S of n strings of total length L, composed of symbols of a general alphabet.*

- adopt the structure of Merge-Sort, merging in pairs ordered groups of strings. At each merging phase i, $1 \geq i \geq \log n$, let $k = 2^{i-1}$ and assume inductively that:
  - S has been already divided into $h = \frac{n}{k}$ ordered groups of sequences $G_1, \ldots, G_h$, with $G_j = s_j^1, \ldots, s_j^k$, $1 \leq j \leq h$;
  - to each s_j^t, with $t > 1$, is associated the length g_j^t of the longest common prefix between s_j^t and s_j^{t-1};
- for each pair G_j, G_{j+1}, with j odd, do the following:
  1. symbolwise compare s_j^1 with s_{j+1}^1; extract (and put in the merged list) the smallest of the two (say s_j^1) and associate to the other one (say s_{j+1}^1) the length (g_{j+1}^1) of the longest common prefix between s_j^1 and s_{j+1}^1;
  2. for the rest of the merge we will always have the smallest string in each group associated with the length g of its longest common prefix with the last string extracted. Let the top strings in G_j, G_{j+1} be s_j^v, s_{j+1}^w. The one to be extracted is the one with the larger value of g (say $g_j^v > g_{j+1}^w$, then extract s_j^v). Note that the string that was not extracted (s_{j+1}^w) keeps its value of g, because this value equals the length of the longest common prefix with the string just extracted. In case $g_j^v = g_{j+1}^w$, scan s_j^v and s_{j+1}^w and compare the succeding characters for finding and extracting the smallest string; then increase the value of g of the larger one by the number of equal characters found in the scanning.

An example of the execution of point 2 of the algorithm is shown in Table 1. In two consecutive steps, two strings are moved into the merged list, one from G_j and then another from G_{j+1}. In the latter step the two strings under comparison ($rscac$ in G_j and $rscab$ in G_{j+1}) have the same value of $g = 2$, then they are scanned from position 3 until a different character is found in position 5. The smaller $rscab$ is then extracted, and the value of g of $rscac$ is increased to 4.

To evaluate the time required by Algorithm 3 note that the value of g of each string never decreases, and it is bound by the string length. Each time two characters $a \in s_j^v$, $b \in s_{j+1}^w$ are compared, either the value of g of the string that will be eventually found the largest will be increased (for $a = b$), or one of the two strings is extracted. Hence the total number of character comparisons is at most $L + n \log n$, and the total number of g's comparisons is $n \log n$, for a total worst-case time of $\Theta(L + n \log n)$.

It must be noted that sorting strings built on a general string-alphabet in time $\Theta(L + n \log n)$ is not a novelty, as this could be done by proper adaptation of other algorithms developed for external memory [6], or for parallel computing [9]. Such algorithms, however, are exceedingly complicated for our purposes, while Algorithm 3 above is straightforward. In summary all the results found here for canonical ordering of labeled trees are reported in Table 2.

step	G_j	g_j	G_{j+1}	g_{j+1}	Merge	g_M
	$rsbt$	3	$rscab$	2	$\cdots$	$\cdots$
	$rscac$	2	vt	0	rsa	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$rsbs$	2
1	$rscac$	2	$rscab$	2	$\cdots$	$\cdots$
	$\cdots$	$\cdots$	vt	0	rsa	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$rsbs$	2
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$rsbt$	3
2	$rscac$	4	vt	0	$\cdots$	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	rsa	$\cdots$
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$rsbs$	2
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$rsbt$	3
	$\cdots$	$\cdots$	$\cdots$	$\cdots$	$rscab$	2

Table 1: Two consecutive steps in the execution of Algorithm 3.

	Simple Alphabets	String Alphabets
constant	$\Theta(n)$	$\Theta(L)$
n-integer	$\Theta(n)$	$\Theta(L)$
general	$\Theta(n \log n)$	$\Theta(L + n \log n)$

Table 2: Times for canonical ordering of a labeled tree of n vertices. L is the total number of symbols contained in string labels. All the results are provably optimal.

1.4 The soundness of canonical ordering

To make use of canonical ordering for comparing unordered trees we have to prove that isomorphic trees or subtrees are ordered in the same way. We will focus on the integers associated to the vertices in step 3(b) of Algorithm 1. Recall that these integers account for the shape and the labels of the subtree rooted at the vertex.

Let the algorithm be applied to an unordered tree T of height h. By the discussion of the previous subsection, T may be labeled with any of the alphabets of points 1 to 5 above. Let v, w be two vertices of T at the same level; let T_v, T_w be the bottom-up subtrees of T rooted at v and w; and let k_v, k_w be the integers associated to v and w by Algorithm 1. Our first lemma proves a statement already posed inside the algorithm:

Lemma 1 *$k_v = k_w$ iff T_v and T_w are isomorphic.*
Proof By induction on the level i of v, w.
Basis. True for $i = h$, where only leaves are present.
Induction step. Letting the result hold for $i + 1$, with $h \geq i + 1 \geq 2$, we prove that it holds for i. (1) If part. If T_v is isomorphic with T_w, the subtrees rooted at the children of v and w are pairwise isomorphic. Therefore by the inductive hypothesis the same tuples, hence the same integers, are associated to v and w. (2) Only if part. If $k_v = k_w$ then the same tuple was assigned to v and w. That is, the subtrees rooted at the children of v and w are pairwise isomorphic by the inductive hypothesis. Q.E.D.

We also have:

Lemma 2 *If $k_v \neq k_w$, the relation $k_v < k_w$ or $k_v > k_w$ depends only on T_v and T_w.*
Proof Let t_v, t_w be the tuples associated with the two vertices. If v and w have a different label (i.e. they differ in the first element of the tuples), the relation between k_v and k_w depends only on these labels and the result trivially holds. If v and w have equal labels we proceed by induction on the level i of the two vertices. Recall that after a tuple is assigned to a vertex x the chidren of x corresponding to the tuple elements are ordered by the algorithm.
Basis. True for $i = h - 1$ by immediate inspection of all the possible cases (note that $k_v \neq k_w$ cannot occur for $i = h$).
Induction step. Letting the result hold for $i+1$, with $h-1 \geq i+1 \geq 2$, we prove that it holds for i. Since $k_v \neq k_w$, then t_v and t_w have a common prefix of length $\ell \geq 1$ and start differing in position $\ell + 1$. W.l.o.g. assume that $k_v < k_w$. Then either: (1) the integer r_v in position $\ell + 1$ in t_v is smaller that the corresponding integer r_w in t_w; or: (2) t_v has length ℓ and t_w is longer. In case (1) the relation $k_v < k_w$ depends only on $r_v < r_w$, and, by the inductive hypothesis, these two integers are function only of the two corresponding subtrees of T_v, T_w. In case (2) the relation $k_v < k_w$ depends only on the fact that, after reordering the children of v and w, the ℓ subtrees of v are equal to the leftmost ℓ subtrees of w by Lemma 1, but w has more children. Q.E.D.

Note that the integers assigned to the vertices of any subtree T_v of T depend on the shape and the labels of all the subtrees rooted at the same level of v, or at a higher level. However, Lemmas 1 and 2 show that the *ordering* induced on the children of v depends only on the vertices of T_v. As this property repeats recursively for all the descendants of v, we immediately have:

Theorem 1 *In the canonical ordering of a tree T, the ordering induced on the vertices of any subtree T_v of T depends only on the vertices of T_v.*

From this follows:

Corollary 1 *Two isomorphic subtrees T_v, T_w of a tree T (resp., of two different trees T, U) become identical after T is rearranged (resp., T and U are rearranged) in canonical order.*

The order induced on a subtree T_v after ordering T is the canonoical order of T_v, since is the same order that would be obtained working on T_v alone.

2 Bottom-up subtree search as a dictionary problem

As already pointed out, our approach to the unordered subtree isomorphism problem starts with canonical ordering of the trees as shown in the previous section. This allows conducting the subtree search on ordered trees. Therefore we now discuss the ordered case, starting with simple labels built on a constant alphabet. The global algorithm for unordered trees, and its extension to other alphabets, will follow.

Given two rooted ordered trees T and P of n and $m \leq n$ vertices, respectively, labeled with the characters of a constant alphabet Σ, we must find all the subtrees of T, if any, that are identical to P. In the example of figure 2 we have two subtrees of T matching with P, that must be reported at the output.

As known from the introduction of this paper, the problem for ordered trees was solved in [13] in linear time. However several different pattern trees $P_1 \ldots P_k$ may be searched for in T, as in a *dictionary problem* where T is given initially as a static text, and $P_1 \ldots P_k$ represent successive queries. Indeed this was presented in [8] as *the* tree matching problem, although directed to approximate matching. Applying the best algorithms known thus far would lead to a total computing time of $\Theta(kn)$. We shall see, however, that a suitable preprocessing of T, done in $\Theta(n)$ time, allows to answer any successive query for P_i in $\Theta(m_i + \log n)$ time where m_i is the size of P_i. Note that, although P_i may match with $\Theta(n/m_i)$ distinct subtrees of T, this number does not appear in the search time because all the occurrences of P will be reconstructible from a constant output information.

Let $0 \notin \Sigma$. Assign an ordering to the characters of $\Sigma \cup \{0\}$, with 0 being the first (i.e., the "smallest") of all. [4] This induces a lexicographic ordering on the strings W_i built on $\Sigma \cup \{0\}$. We write $W_x < W_y$ if W_x precedes W_y in that ordering. Let a *preorder string* W be defined as one with the following recursive property:

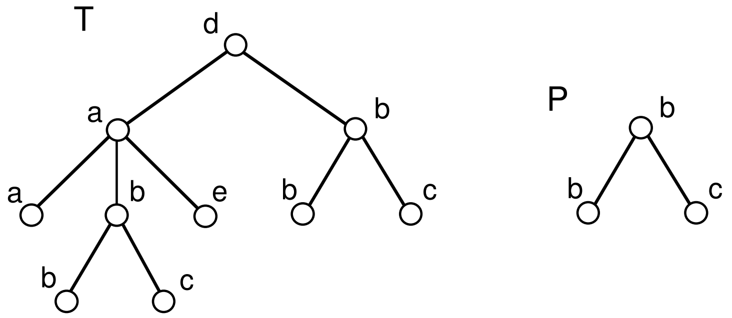
$$W = \ell\ 0 \qquad \text{or} \qquad W = \ell\ W_1 \ldots W_h\ 0 \tag{1}$$

where $\ell \in \Sigma$ and $W_1 \ldots W_h$ are preorder strings. A (nonempty) ordered tree of n vertices can be represented as a preorder string of $2n$ characters [12]. The tree is traversed in preorder. For each vertex encountered, the corresponding label ℓ is entered in W, and for each return to the previous level a 0 is entered in W. Letting h be the number of children of the root, the substrings $W_1 \ldots W_h$ in equation (1) are the preorder strings recursively associated to the subtrees of the root. Figure 2 shows the preorder string W for the tree T, with $n = 10$. Immediate properties of W are the following:

1. Preorder strings contain as many characters of Σ as 0's.
2. Each proper prefix of a preorder string has more characters of Σ than 0's.
3. There is a one to one correspondence between ordered trees and preorder strings.
4. If an ordered tree T has preorder string W, the preorder string of any subtree of T is a substring (i.e. a portion of consecutive characters) of W starting with the label of the subtree root.

Point 4. above implies that the preorder strings of the subtrees of a given tree may totally but not partially overlap, e.g. see figure 2. Once a tree is represented as a preorder string W, a *suffix array* A can be built in linear time and space, to specify the starting positions in W of its $2n$ suffixes. The main property of A is that, scanning the array, the suffixes of W are retrieved in lexicographic order. See for example [4] for suffix arrays properties and treatment, and [10] for a discussion on the construction of A with different alphabets. Note that for a preorder string W the n suffixes starting with 0 are pointed to by the first half of A, because they precede lexicographically the n suffixes starting with a vertex label. We consider only the second half of A and call it the *subtree array* of W. See figure 3 where the suffixes pointed by the entries of A (indicated as A-suffixes) are reported for clarity but are not part of A. We are now interested at pointing to substrings of W corresponding to subtrees, instead of suffixes. Since two subtrees may be identical, and correspond to identical substrings, we naturally extend the ordering notation for strings to include equality (symbol $\leq$). We have:

[4] If the symbols of Σ are coded with positive integers there is an implicit ordering relation among them, including the 0. For clarity, in the examples we use alphabetic characters as labels.



x	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20
W	d	a	a	0	b	b	0	c	0	0	e	0	0	b	b	0	c	0	0	0

Figure 2: Sample trees T and P. W is the preorder string of T and x is the string position.

Lemma 3 *Let T be an ordered tree, W be its preorder string, and A be the subtree array of W. Then the elements $A[1], .., A[n]$ respectively point to the preorder strings $W_1, \ldots, W_n$ of the subtrees of T, with $W_1 \leq W_2 \leq \ldots \leq W_n$.*
Proof By construction $A[1], .., A[n]$ point to the suffixes $S_1, .., S_n$ of W which start with vertex labels, with $S_1 < S_2 < \ldots < S_n$. Therefore A also points to the preorder strings $W_1, \ldots, W_n$ of the subtrees of T, where each W_i is a prefix of S_i. We must prove that $W_1 \leq W_2 \leq \ldots \leq W_n$. In fact $W_1, .., W_n$ constitute the most significant parts of $S_1, .., S_n$ when ordering these suffixes, being their prefixes. By points i) and ii) of lemma **??** no W_i can be a proper prefix of W_j for $i \neq j$. Then for each value of i, $1 \leq i \leq n-1$, the most significant mismatching character which establishes the relation $S_i < S_{i+1}$ must lie inside W_i and W_{i+1}, or outside both such strings. In the first case we have $W_i < W_{i+1}$. In the second case we have $W_i = W_{i+1}$. Q.E.D.

Once the subtree array A has been built, a subtree P of m vertices can be searched for in T by building the preorder string W_P for P and then searching all the occurrences of W_p in W. This is done with the aid of A, with the standard technique used for suffix arrays. A binary search is done on A. For each element $A[i]$ encountered in this search, the string W_P is compared with the substring of W starting at $A[i]$. The characters of the two strings that are found to be matching from left to right are exempted from future comparisons. This allows to determine, in $O(m + \log n)$ time, two bounding indexes i, j of A such that, for $i \leq k \leq j$, each element $A[k]$ points to an occurrence of W_P in W. Note that the total number of occurrences $j - i$ is known at this point, while listing all of them would require an extra time depending on their number. For a subtree not occurring in T the binary search obviously reports a failure.

For the example of figure 2 we have $W_P = b\,b\,0\,c\,0\,0$. This string is searched for

y	A	$A - suffix$
01	03	$\underline{a\;0}\;b\ldots 0$
02	02	$\underline{a\;a\;0\;b\;b\;0\;c\;0\;0\;e\;0\;0}\;b\;\ldots 0$
03	15	$\underline{b\;0}\;c\;0\;0\;0$
04	06	$\underline{b\;0}\;c\;0\;0\;e\ldots 0$
05	14	$\underline{b\;b\;0\;c\;0\;0}\;0$
06	05	$\underline{b\;b\;0\;c\;0\;0}\;e\ldots 0$
07	17	$\underline{c\;0}\;0\;0$
08	08	$\underline{c\;0}\;0\;e\ldots 0$
09	01	$\underline{d\;a\ldots\ldots\ldots 0}$
10	11	$\underline{e\;0}\;0\ldots 0$

Figure 3: The subtree array A for the tree T of figure 2. y is the array position. $A[y]$ is the starting position in W of a suffix starting with a vertex label. The suffixes of W, with underlined prefixes corresponding to subtrees, are shown for clarity. Note that these suffixes appear in lexicographic order.

in W with the aid of the subtree array A of figure 3, until the bounding indices 05, 06 are found. In fact $A[05] = 14$ and $A[06] = 05$, which are the two positions in W where W_P starts.

We can specify our method as follows.

Algorithm 4 *Preprocessing an unordered text tree T labeled with a constant alphabet.*

1. put T in canonical order with Algorithm 1;
2. build the preorder string W for T;
3. build the subtree array A for W.

Algorithm 5 *Searching an unordered labeled pattern tree P into T.*

1. put P in canonical order with Algorithm 1;
2. build the preorder string W_P for P;
3. search for W_P in W through a binary search on A;
4. if the search of step 3. is successful report the two bounding indexes, otherwise declare that P does not occur in T.

The correctness of our method stems from Corollary 1. In fact if a subtree S of T is isomorphic with P, S and P are transformed into identical trees after applying Algorithm 1 to T and P. From our previous discussion, and applying algorithms 4 and 5, it follows immediately:

Alphabet	Preprocessing T		Searching P	
	time	space	time	space
constant	$\Theta(n)$	$\Theta(n)$	$O(m + \log n)$	$\Theta(m)$
n-integer	$\Theta(n)$	$\Theta(n)$	$O(m + \log n)$	$\Theta(m)$
general	$\Theta(n \log n)$	$\Theta(n)$	$O(m \log m + \log n)$	$\Theta(m)$

Table 3: Time and space for treating trees labeled with simple alphabets.

Alphabet	Preprocessing T		Searching P	
	time	space	time	space
constant	$\Theta(L_T)$	$\Theta(L_T)$	$O(L_P + \log n)$	$\Theta(L_P)$
n-integer	$\Theta(L_T)$	$\Theta(L_T)$	$O(L_P + \log n)$	$\Theta(L_P)$
general	$\Theta(L_T + n \log n)$	$\Theta(L_T)$	$O(L_P + m \log m + \log n)$	$\Theta(L_P)$

Table 4: Time and space for treating trees labeled with string-alphabets. L_T and L_P are the total numbers of symbols contained in the labels of T and P, respectively.

Theorem 2 *Given two rooted unordered trees T and P of n and m vertices respectively, labeled with a constant alphabet, all the occurrences of P as a subtree of T can be determined in $O(m + \log n)$ time, after preprocessing T in $\Theta(n)$ time. The additional space is $\Theta(n)$ for T and $\Theta(m)$ for P.*

The extension of our method to trees labeled with the other alphabets considered in Section 1 is straightforward. In particular T and P must be preprocessed with Algorithm 2 before applying Algorithm 1 in step 1 of Algorithms 4 and 5; and Algorithm 3 must be preliminarily run for general string alphabets. Note that the integers γ produced by Algorithm 2 are used for the canonical ordering only, while the additional structures W and A must be built on the true labels. From this we can express all our results as follows:

Corollary 2 *Given two rooted unordered trees T and P of n and m vertices respectively, labeled with a simple or a string-alphabet, all the occurrences of P as a subtree of T can be determined in time and space as reported in Tables 3 and 4.*

Note that the search time for P, for a general simple or string alphabet, includes a term $m \log m$ that accounts for the canonical ordering of P and absorbs the term m of the search.

3 Conclusion

We have shown how the bottom-up instance of the exact subtree matching problem for unordered rooted trees can be efficiently solved, first transforming the trees into a canonically ordered version, then working on the simplified problem of ordered trees. This latter problem is solved with a proper preprocessing of the text tree T, where the key data structure is a subtree array built as an extension to trees of the well known suffix array for strings. Then the search of any pattern tree P can be efficiently performed. A substantial part of the overall process is aimed at building the canonical order, as this strongly simplifies the search phase.

The treatment of string labels has also required to define a new simple optimal algorithm for string sorting in a general alphabet.

As in the seminal paper [8], a tree T is given initially as a static structure on which several different pattern trees can be searched for. In our proposal the trees are labeled with different alphabets, but the basic technique holds with minor modifications for all of them. The results in time and space are summarized in Tables 3 and 4. Note that the search time for a pattern P does not include the number Occ of different occurrences of this tree in T, where Occ is upper bounded by n/m, because all such occurrences can be reconstructed from the (constant size) output of the algorithm.

In exact tree matching there is probably no room for further improvement. Much work, instead, has to be done for approximate tree matching. We are currently investigating on the possible extension of our approach to the approximate case, at least to some relevant instances of it.

References

[1] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading 1974.

[2] M.J. Chung. $O(n^{2.5})$ time algorithms for the subgraph homeomorphism problem on trees. *Journal of Algorithms* 8 (1987) 106-112.

[3] R. Cole, R. Hariharan, and P. Indyk. Tree pattern matching and subset matching in deterministic $O(n \ \log^3 n$ time. *Proc. ACM-SIAM Symp. on Discrete Algorithms, SODA'99.* ACM Press (1999) 245-254.

[4] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, New York 1994).

[5] M. Dubiner, Z. Galil, and E. Magen. Faster tree pattern matching. *Journal of the ACM* 41 (1994) 205-213.

[6] P. Ferragina ans R. Grossi. The String B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM* 46 (1999) 236-280.

[7] R. Grossi. A note on the subtree isomorphism for ordered trees and related problems. *Information Processing Letters* 32 (1989) 271-273.

[8] C.M. Hofmann and M.J. O'Donnell. Pattern matching in trees. *Journal of the ACM* 29 (1982) 68-95.

[9] J.F. Jaja, K.W. Ryu, and U. Vishkin. Sorting strings and constructing digital search trees in parallel. *Theoretical Computer Science* 154 (1996) 225-245.

[10] J. Karkkainen and P. Sanders. Simple linear work suffix array construction. *Proc. International Colloquium on Automata, Languages, and Programming, ICALP'03.* Springer-Verlag LNCS 2719 (2003) 943-955.

[11] S.R. Kosaraju. Efficient tree pattern matching. *Proc. 30th annual IEEE Symp. on Foundations of Computer Science, FOCS'89.* IEEE Press (1989) 178-183.

[12] F. Luccio and L. Pagli. Approximate matching for two families of trees. *Information and Computation* 123 (1995) 111-120.

[13] E. Mäkinen. On the subtree isomorphism problem for ordered trees. *Information Processing Letters* 32 (1989) 271-273.

[14] D.W. Matula. Subtree isomorphism in $O(n^{5/2})$. *Annals of Discrete Mathematics* 2 (1978) 91-106.

[15] S.W. Reyner. An analysis of a good algorithm for the subtree problem. *SIAM Journal on Computing* 6 (1977) 730-732.

[16] R. Shamir and D. Tsur. Faster subtree isomorphism. *Journal of Algorithms* 33 (1999) 267-280.

[17] G. Valiente. *Algorithms on Trees and Graphs.* Springer-Verlag, Berlin 2002.

[18] V.N. Zemlyachenko. Determining tree isomorphism. In: *Voprosy Kibernetiki, Proc. Seminar on Combinatorial Mathematics*, Moskow (1971) 54-60.

[19] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing* 18 (1989) 1245-1262.