Università di Pisa
Dipartimento di Informatica

Technical Report: TR-01-14

Exact Rooted Subtree Matching in Sublinear Time

Fabrizio Luccio   Antonio Mesa Enriquez   Pablo Olivares Rieumont
Linda Pagli

July 9, 2001

Address: Corso Italia 40, 56125 Pisa, Italy.
Tel: +39 050 2212700
Fax: +39 050 2212726

Exact Rooted Subtree Matching in Sublinear Time

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

July 9, 2001

The problem of exact subtree matching is the one of deciding if a *pattern tree* P of m vertices is a 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. We consider ordered and non-ordered rooted trees with labeled vertices (the case of unlabeled vertices is a special case of this), and show how the problem can be solved in $\Theta(m + \log n)$ time once a proper data structure is built for T in a preprocessing phase which requires $\Theta(n)$ time and space. Regarding T as a static text on which several queries are made, P as the contents of one such query, and assuming $m = o(n)$, we can speak of search time sublinear in the size $m + n$ of the overall structure. 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.

Keywords: Rooted tree, Subtree matching, Subtree isomorphism, Tree pattern matching, Design of algorithms.

1 Subtree isomorphism and related problems

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 basic distinction is between rooted and non-rooted trees, and, in the former family, between ordered and non-ordered trees. Another basic distinction is between *exact* matching (or isomorphism), and *approximate* matching. In the former case we look for subtrees S of T which are identical to P. In the other case some edit operations are allowed to transform P into S. These operations may refer to label mismatchings, or, more interestingly, to differences in shape.

A less crucial distinction is between labeled and unlabeled trees. 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. In general unlabeled and labeled trees have been studied in the context of exact and approximate matching, respectively. Let us comment briefly on the different versions of the problem.

- **Isomorphism, or exact matching**. The problem of tree isomorphism ($m = n$) was implicitly solved in linear time with the famous Hopcroft and Tarjan's algorithm for planar

*Dipartimento di Informatica, Università di Pisa.
†Facultad de Matemática y Computación, Universidad de la Habana.
‡Facultad de Matemática y Computación, Universidad de la Habana.
§Dipartimento di Informatica, Università di Pisa.

graph isomorphism. A nice alternative linear solution can be found in [1], Ch.3. Subtree isomorphism ($m \leq n$) is more difficult.

1. Subtree isomorphism of rooted trees. After the pioneer work of [15], it was shown in [13] how to solve the problem for ordered trees in $\Theta(n)$ time, coding P and T as strings and applying a string matching algorithm to them. All known algorithms apply to ordered trees. In the present note we use a string representation of trees together with some advanced data structures, to solve the problem for ordered and non ordered trees. Our approach requires preprocessing T in $\Theta(n)$ time, and then allows to make any number of searches for different patterns each in $\Theta(m + \log n)$ time.
2. Subtree isomorphism of non-rooted trees. After the pioneer works in [14, 4], a recent paper [16] shows how to solve the problem in $O(n\, m^{1.5}/\log m)$ time.

- **Approximate matching**. The problem of approximate tree matching asks for a definition of distance between trees. This was cleverly treated in [17] for rooted ordered trees, where the operations of label change, vertex insertion and vertex deletion were defined. The distance between two trees S, T was then defined as the minimum total cost of a sequence of such operations to transform S into T. The transformation, together with some relevant variations of the problem, was constructed in [17] in more than quadratic time. Approximate matching of a pattern P with subtrees of T, often called **tree pattern matching**, was more widely studied. All the works mentioned below apply to rooted ordered trees.

1. Approximate subtree matching with equal vertex degree. The seminal paper is [9], where the importance of this area was thoroughly enlightened. The vertices of P and T are labeled with characters of the same alphabet, however, some of the leaves of P may be labeled with a special character ν not appearing in T. P matches with a subtree S of T if each vertex labeled ν can be replaced in P with a proper subtree of T, such that the tree P' thus formed is isomorphic to S and the labels of the corresponding vertices of P' and S match. Note that all the original vertices of P with a label different from ν have the same degree of the corresponding vertices in S, hence in T. The solution proposed in [9] requires $O(n\, m)$ time. The problem was then solved in [11] in $O(m + k\, n)$ time, where k is the number of labels ν in P.

   A different improvement was done in [10] where a problem very close to the one above is solved in $O((n\, m^{0.75}\, poly\log(m))$ time. The only difference is that ν labels are not used. Any leaf x of P may be replaced with a subtree of T to form S, provided that x and the root of the subtree substituting it have the same label. This problem was generalized in [12] where deletions of subtrees form P were also allowed, to get a new tree isomorphic to S. If up to k subtrees can be inserted or deleted in P, the new upper time bound is $O(m + k\, n)$. Label mismatches between vertices of P and S were also considered in [12], with an upper time bound of $O(m + max(k, h)\, n)$ where h is the maximum allowed number of such mismatches. In fact if the differences between P and S are limited to label mismatches the problem can be solved in linear time (see [7]).
2. Approximate subtree matching with different vertex degree. An immediate improvement over the result of [10] was given in a conference presentation later published in [6]. The problem now considered is slightly more general because a non-leaf vertex x of P may have smaller degree than the corresponding vertex y of S, but the children of x must be put into correspondence with the leftmost children of y. The algorithm requires $O((n\, m^{0.5}\, poly\log(m))$ time. A new approach presented in [5] attains a time bound of $O(n \log^3 n)$, thus improving over the result of [6] for m not too small. The use of different data structures leads to very good practical performances [2], although the required time is quadratic in the worst case.

In a recent work [3] the problem is made much more general, by allowing the children of x to be put into correspondence with any subset of children of y provided the ordering among siblings is maintained. The time bound increases to $O(n\,\ell)$, where ℓ is the number of leaves of P.

In this note we study exact subtree matching for rooted trees, ordered and non-ordered. For generality our trees are labeled.

2 Exact subtree matching: ordered trees

Given two rooted ordered trees T and P whose vertices are labeled with the characters of a finite alphabet Σ, we must find all the subtrees of T, if any, that are identical to P. In the example of figure 1, the two subtrees of T matching with P must somehow be reported.

As known from the previous section, the problem can be solved in time linear with the total size $n + m$ of the input, practically $\Theta(n)$ since $m \leq n$. 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 $P_1 \ldots P_2$ represent successive queries. Indeed this was presented in [9] as *the* tree matching problem, although directed to approximate matching. Applying the 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. 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 reconstructable 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. 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:

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

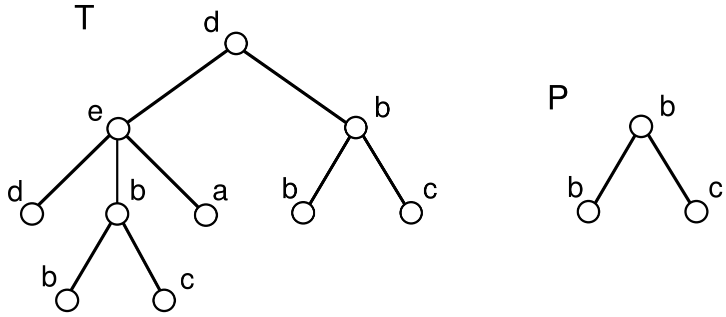
where $\ell \in \Sigma$ and $W_1 \ldots W_h$ are preorder strings. A (nonempty) ordered tree of n vertices can be represented as a preorder string of $2n$ characters, obtained traversing the tree in preorder [12]. For each vertex encountered, the corresponding label ℓ is entered in W, and for each return 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 1 shows the preorder string W for the tree T, with $n = 10$. By an easy inductive argument we have:

Lemma 1 *i) Preorder strings contain the same number of characters of Σ and 0's.*
ii) Each proper prefix of a preorder string has more characters of Σ than 0's.
iii) There is a one to one correspondence between ordered trees and preorder strings.
iv) 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 root label of the subtree.

Point iv) of the lemma implies the the preorder strings of the subtrees of a given tree may totally but not partially overlap. In the example of figure 1 the subtree of T with root label e correspond to the substring $W[2..13] = e\ d\ 0\ b\ b\ 0\ c\ 0\ 0\ a\ 0\ 0$. The subtree (single leaf) with root label a correspond to the substring $W[11..12] = a\ 0$ which is totally contained in the former one.

If the tree is unlabeled, W is built entering 1's instead of the labels. All what follows can be applied to unlabeled trees with this simple modification.

Once a tree is represented as a preorder string W of length $2n$, some data structures designed for string search may be used. In particular a *suffix array* A can be built in linear time and space, specifying the starting positions in W of its $2n$ suffixes. The main property of A is that,



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

Figure 1: Sample trees T and P. W is the preorder string of T and x is the string position. F specifies the parent's position of each vertex in W. C points to a list of children's positions. Z is the sorted preorder string of T.

scanning the array, the suffixes of W are retrieved in lexicographic order. That is, for $i < j$ we have $W[A[i]..2n] < W[A[j]..2n]$ (see for example [8]). Note that for a preorder string W the n suffixes starting with 0 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 2 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:

Lemma 2 *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 in the ordering of $S_1, .., S_n$, being their prefixes. By points i) and ii) of lemma 1 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 $W_i = W_{i+1}$. □

Lemma 2 and its proof can be checked on the example of figure 2. 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 following technique used for suffix arrays [8]. A binary search is done on A. For each element $A[i]$

y	A	$A-suffix$	B	$B-suffix$
01	11	$\underline{a\ 0}\ 0\ldots0$	09	$\underline{a\ 0}\ b\ldots0$
02	15	$\underline{b\ 0}\ c\ 0\ 0\ 0$	12	$\underline{b\ 0}\ c\ 0\ 0\ d\ldots0$
03	06	$\underline{b\ 0}\ c\ 0\ 0\ a\ldots0$	03	$\underline{b\ 0}\ c\ 0\ 0\ e\ldots0$
04	14	$\underline{b\ b\ 0\ c\ 0\ 0}\ 0$	11	$\underline{b\ b\ 0\ c\ 0\ 0}\ d\ldots0$
05	05	$\underline{b\ b\ 0\ c\ 0\ 0}\ a\ldots0$	02	$\underline{b\ b\ 0\ c\ 0\ 0}\ e\ldots0$
06	17	$\underline{c\ 0}\ 0\ 0$	14	$\underline{c\ 0}\ 0\ d\ldots0$
07	08	$\underline{c\ 0}\ 0\ a\ldots0$	05	$\underline{c\ 0}\ 0\ e\ldots0$
08	03	$\underline{d\ 0}\ b\ldots0$	17	$\underline{d\ 0}\ 0\ 0$
09	01	$\underline{d\ e\ldots\ldots\ldots0}$	01	$\underline{d\ b\ldots\ldots\ldots0}$
10	02	$\underline{e\ d\ldots a\ 0\ 0}\ b\ldots0$	08	$\underline{e\ a\ldots d\ 0\ 0}\ 0$

Figure 2: Two subtree arrays A and B for the tree T of figure 1. y is the array position. $A[q]$ (resp. $B[q]$) is the starting position in W (resp. Z) of a suffix starting with a vertex label. The A-suffixes of W and the B-suffixes of Z, with underlined prefixes corresponding to subtrees, are shown for clarity. Note that these suffixes appear in lexicographic order.

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 occurence 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 1 we have $W_P = b\ b\ 0\ c\ 0\ 0$. This string is searched for in W with the aid of the subtree array A of figure 2, until the bounding indices 04, 05 are found. In fact $A[04] = 14$ and $A[05] = 05$, which are the two positions in W where W_P starts. We can specify our method as follows.

Algorithm 1 *Preprocessing of text tree T (ordered trees).*

1. build the preorder string W for T;
2. build the subtree array A for W.

Algorithm 2 *Searching P into T (ordered trees).*

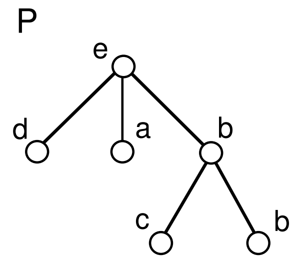
1. build the preorder string W_P for P;
2. search for W_P in W through a binary search on A;
3. if the search of step 2. is successful report the two bounding indexes, otherwise declare that P does not occur in T.

From our discussion, and applying algorithms 1 and 2, it follows immediately:

Theorem 1 *Given two rooted ordered trees T and P of n and m vertces respectively, all the occurrences of P as a subtree of T can be determined in $O(m+\log n)$ time, after preproxcessing T in $\Theta(n)$ time.*

3 Exact subtree matching: non-ordered trees

Finding subtree matchings for non-ordered trees may seem hard because, for each vertex v of the pattern tree, any permutation of the subtrees descending from v is relevant in in the comparison



x	01	02	03	04	05	06	07	08	09	10	11	12
Z_P	e	a	0	b	b	0	c	0	0	d	0	0

Figure 3: A non-ordered pattern tree P and its sorted preorder string Z_P.

with the text tree. Tree P of figure 3, for example, matches with a subtree of T in figure 1 if proper permutations are applied. We now show that our proposed approach can be extended to solve the new problem without increasing the asymptotic time complexity.

First we extend the lexicographic ordering from strings to ordered trees. For two ordered trees T_1, T_2 with preorder strings W_1, W_2, we let $T_1 \leq T_2$ if $W_1 \leq W_2$. For an (ordered or non-ordered) tree T consider the ordered trees $T_1, T_2, ...$ obtined from T with all possible subtree permutations, and let k be the index for which $T_k \leq T_i$ for all i. The basic structure to be used for T is its *sorted preorder string* Z, defined as the preorder string of T_k. Two sorted preorder strings Z and Z_P are shown in figures 1 and 3, for the trees T and P, respectively. It is now clear that, for (ordered or non-ordered) P and T, P occurs in T if and only if Z_P occurs in Z (e.g. this happens with the trees of figures 1 and 3 where $Z_P = Z[8..19]$). Therefore the technique proposed in the previous section can be used unchanged for non-ordered tree matching, if applied to sorted preorder strings. The problem is building such strings without increasing the asymptotic time bounds.

We do not pretend the following method to be the most practical, however, we want to show that the data structures introduced before are useful also in this case. In particular we make use of the ordering built in A. The reader may follow our discussion on the trees T of figures 1 and 2, and P of figure 3. Preprocessing T starts again with the construction of W and A. Then a *parent vector* F is built, containing poiters to the parents of the vertices of T. Specifically, if $W[x]$ represents a vertex v different from the root, and $W[y]$ represents the parent of v, we put $F[x] = y$. Then a vector C is built, containing pointers to lists of the children of the vertices of T. Specifically, if $W[x]$ represents a non leaf vertex v, and $W[y_1], ..., W[y_k]$ represent the children of v in the order in which they are encountered scanning A, then $C[x] =$ points to the list $y_1, ..., y_k$. (In figure 1 such lists are columns of integers. For example vertex e is represented in $W[2]$ and has children d, b, a, respectively represented in $W[3], W[5], W[11]$. Scanning A we first encounter $A[1] = 11$, and since the parent of $W[11]$ is in position $F[11] = 2$ we append 11 to the list pointed by $C[2]$. Later we encounter $A[5] = 5$ and $A[8] = 3$, both with parent in 2, and append 5 and 3 to the list $C[2]$. Note that the order of the list elements reflects the lexicographic order of the subtrees whose roots are pointed to by such elements). From W and C we can finally build the sorted preorder sequence Z for T with the procedure SORTW specified below. Finally we build the new subtree array B for Z.

A pattern tree P is now searched for building its sorted preorder sequence Z_P with the same technique used for Z, and then searching the occurrences of Z_P in Z by means of B.

Let the elements of the lists pointed by C have the format: Key[p],Next[p]. We specify our method as follows:

Algorithm 3 *Preprocessing of text tree T (non-ordered trees).*

1. build the string W and the array A for T with algorithm 1;
2. build the parent vector F with an elementary scanning of W;
3. { *building the lists pointed by C* }

   for $i \leftarrow 1$ **to** n **do**

       if $A[i] \neq 1$ { *$W[1]$ is not the root of T* }

           then append $A[i]$ to the list pointed by $C[F[A[i]]]$;
4. { *building the sorted preorder string Z from W and C* }

   $Z[1] \leftarrow W[1]$, $i \leftarrow 1$, SORTW(1);
5. build the subtree array B for Z.

Procedure SOTRTW(p)

$p \leftarrow C[p]$

while $p \neq NIL$ **do**

    $i \leftarrow i+1$, $Z[i] \leftarrow W$[Key[p]]

    SORTW(Key[p])

    $p \leftarrow$ Next[p]

$i \leftarrow i+1$, $Z[i] \leftarrow 0$.

Algorithm 4 *Searching P into T (non-ordered trees).*

1. build the sorted preorder string Z_P for P with algorithm 3, steps 1 to 4, applied to P;
2. search for Z_P in Z through a binary search on B;
3. if the search of step 2. is successful report the two bounding indexes, otherwise declare that P does not occur in T.

We can apply algorithm 3 to build the basic data structures for T, then algorithm 4 for any successive search of a pattern tree P into T. The correctness of the two algorithms can be easily verified. We also have:

Theorem 2 *Given two rooted non-ordered trees T and P of n and m vertces respectively, all the occurrences of P as a subtree of T can be determined in $\Theta(m + \log n)$ time, after preproxcessing T in $\Theta(n)$ time.*

Proof i) Preprocessing T with algorithm 3. Each step of the algorithm takes $\Theta(n)$ time. Namely: Step 1: see the previous section. Step 2: immediate. Step 3: each of the n iterations of the **for** cycle requires a constant number of accesses to A, F and C. Note that the lists built in this step require total space $\Theta(n)$. Step 4: the procedure SORTW recursively scans the positions of W with label $\neq 0$ (assignment $Z[i] \leftarrow W$[Key[p]], if $p \neq NIL$), and iteratively scans the positions of Z where 0 has to be entered (assignment $Z[i] \leftarrow 0$, if $p = NIL$). Step 5: as for step 1.
ii) Searching P with algorithm 4. Step 1 takes time $\Theta(m)$ as proved for algorithm 3. Steps 2 and 3 take time $\Theta(m + \log n)$, see previous section. $\square$

4 Conclusion

We have shown how the exact subtree matching problem for rooted trees can be efficiently solved both for ordered and non-ordered trees, once a proper preprocessing of the text has been carried out. The key data structure is a subtree array built as an extension to trees of the well known suffix array for strings.

As in the seminal paper [9], the text tree T is given initially as a static structure on which several different pattern trees $P_1 \ldots P_k$ can be searched for. This assumption has been recently resumed in the context of approximate tree matching [2], and allows preprocessing T before the search starts. In our proposal such a preprocessing takes $\Theta(n)$ time, and answering any successive query takes $\Theta(m + \log n)$ time. This figure does not include the number of different occurrences of each P in T, where this number may be of order $\Theta(n/m)$, because all such occurrences can be reconstructed from the (constant sized) 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.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison Wesley, Reading 1974.

[2] C. Chauve Pattern matching in static trees. Research Report RR 1254-01, LaBRI University of Bordeaux I, 2001.

[3] C. Chauve Tree pattern matching with a more general notion of occurrence of the pattern. Submitted manuscript, 2001.

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

[5] 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.

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

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

[8] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, Cambridge 1997.

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

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

[11] F. Luccio and L. Pagli. An efficient algorithm for some tree matching problems. *Information Processing Letters* 39 (1991) 51-57.

[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] 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.