Tree Pattern Matching for Linear Static Terms

Cedric Chauve

LaCIM and Département d'Informatique
Université du Québec à Montréal

Proceedings of SPIRE 2002, A. Laender and A. Oliveira eds., volume 2476 of
Lecture Notes in Computer Science, Springer-Verlag, p 160-169, 2002.

Tree Pattern Matching for Linear Static Terms

Cedric Chauve

LaCIM and Département d'Informatique
Université du Québec à Montréal
Case Postale 8888, Succursale Centre-Ville
H3C 3P8, Montréal (QC), Canada
chauve@lacim.uqam.ca

Abstract. In this paper, we present a simple algorithm for pattern matching within a family of trees called linear terms, that have many applications in the design of programming languages, theorem proving and symbolic computation for example. Our algorithm relies on the representation of a tree by words. It has a quadratic worst-case time complexity, which is worse than the best known algorithm, but experimental results on uniformly distributed random binary terms suggest a linear expected time and interesting practical behavior.

The tree pattern matching (TPM) problem consists of finding all the occurrences of a pattern tree P as subtree of a target tree T. Aside from its theoretical interest (it is a natural generalization of the classical pattern matching in words (WPM) problem), it has applications in several areas of computer science from design of programming languages [1, 9] to computational molecular biology [15, 19]. In this paper, we consider a particular family of trees, called linear terms, which is very common in the field of compiler design and theorem proving for instance. We describe a very simple algorithm to search a set of pattern trees in a fixed (*static*) target tree. Our algorithm relies on encoding trees by words and on a classical data structure in the field of word algorithms, the *suffix tree*, and seems to have a good behavior in practice. In Section 1, we present a quick survey of linear terms and of the TPM problem for this family of trees. We continue by a precise description of our algorithm, followed by an experimental study. We conclude with some problems.

1 Tree Pattern Matching and Linear Terms

Linear terms. Let Σ be a finite alphabet and V a symbol not belonging to Σ. We associate to every symbol of Σ an integer value called its arity. We call a *target term* an ordered tree such that every vertex is labeled by a symbol of Σ and has a degree (its number of children) equal to the arity of its label. The definition of a *linear pattern tree* is similar to the one of a target term, except that the leaves (vertices of degree 0) of a linear pattern term can be labeled either by the symbols of Σ of arity 0 or by the symbol V. Terms are used, for instance, to represent boolean expressions (see Figure 1) in the field of automatic theorem proving, or derivation trees related to a grammar in the analysis of programs or in compiler design.

Tree pattern matching for linear terms. Let T be a target term and P a linear pattern term, of respective size (number of vertices) n and m. For a vertex v of T, there is an occurrence of the pattern P at v if there exists a one-to-one map from the vertices of P to the vertices of T such that:

- the root of P maps to v,
- if an internal vertex x maps to y then x has the same label as y (the two vertices have the same degree) and the i^{th} child of x maps to the i^{th} child of y,
- if a leaf x maps to y then either x is labeled with V or x and y have the same label.

The TPM problem for linear terms consists of computing the set of vertices of T where there is an occurrence of P.

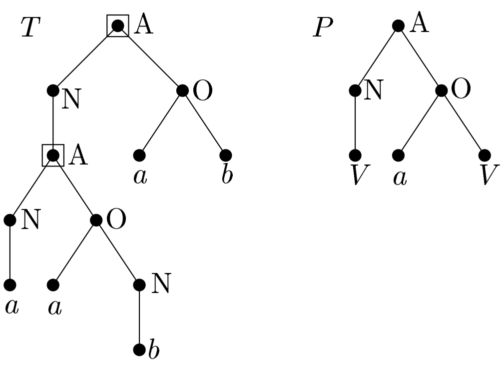


Fig. 1. A target term T, representing a parameterized boolean expression (A stands for And, O for Or and N for Not), and a linear pattern term P. There are two occurrences of P in T, at the vertices marked with a square.

Previous results. There is an immediate algorithm for solving the TPM problem, usually called the *naïve algorithm*, consisting of a traversal of T, calling at every visited vertex v a recursive procedure testing if there is an occurrence of P at v (see [17] for example). This algorithm is very simple to implement and achieves a quadratic $O(n\ m)$ time complexity[1], but Flajolet and Steyaert proved that the expected time is in $\Theta(n+m)$ [17].

The first non trivial algorithms were proposed by Hoffmann and O'Donnell [9], but their algorithms still have a quadratic time complexity (later, their bottom-up algorithm has been improved by Cai, Paige and Tarjan [3], and Thorup [18]). The first algorithm with a subquadratic time complexity is due to

[1] Unless specified, in this paper we consider the worst-case time complexity.

Kosaraju [11], who proposed an algorithm running in $O(n\ m^{3/4}\ \mathrm{polylog}(m))$ time. This result has been improved successively by Dubiner, Galil and Magen [8], Cole and Hariharan [5], and Cole, Hariharan and Indyk [6], who achieved an almost linear $O(n\ \log^3 n)$ time complexity.

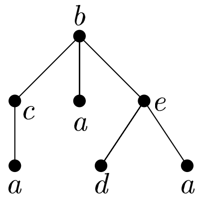
The static case of the pattern matching problem. In a pattern matching problem, one usually distinguishes two cases. In the *dynamic* case, the target structure is not fixed (it can be modified between the search of different patterns) and the algorithm is based on a preprocessing of the pattern structure (for words we can cite the Knuth-Morris-Pratt and Boyer-Moore algorithms [10, 2]). In the *static* case, we assume that the target is fixed (if it is inserted in a data base for example) and that many patterns could be searched through this target tree. Thus we can preprocess the target in order to speed-up the pattern matching phase. For the WPM problem, the preprocessing of the target consists of computing an auxiliary data structure indexing the suffixes of the target, like the suffix tree, the suffix automaton or the suffix array (see [7, chapters 5 and 6]). For words, this preprocessing is linear (in space and time) in the size of the target and allows to compute the positions of the occurrences of P in T in time almost linear in the size of P ($O(|P|\ \log|\Sigma|)$ with a suffix tree or a suffix automaton, and $O(|P| + \log|T|)$ with a suffix array). This property makes this method very useful in the case of numerous pattern searches in the same target.

Motivation and results. In the context of the TPM problem for terms, we can notice that there are only two algorithms dealing with the static case, but that these two algorithms suffer from some limitations. The bottom-up algorithm of Hoffmann and O'Donnell [9] has an exponential time preprocessing in the general case and is very complicated to implement. The algorithm of Shibuya [16], based on a new suffix tree for a tree data-structure, allows only to treat particular instances of this problem: the target term must be a k-ary tree (every vertex other than a leaf has degree exactly k, for a given integer k) and the pattern term must be a balanced k-ary tree. Moreover, we should add that all the known algorithms for the TPM problem are quite sophisticated and difficult to implement efficiently, which often leads people to use the naive algorithm, which is very simple and very efficient in practice.

Our aim in this paper is to present a simple algorithm for the TPM problem for linear terms dedicated to the static case. The algorithm we propose relies on a classical representation of terms as words proposed by Luccio and Pagli for some tree matching problems [12–14] and on the classical suffix tree data structure. They have the following properties: the preprocessing phase has a $O(n\ \log|\Sigma|)$ time complexity and $O(n)$ space complexity, and the pattern matching phase has a quadratic $O(n\ m)$ time complexity. This time complexity is the same as the naive algorithm, but it appears from experiments that our simple algorithm seems to have a linear expected time and a better behavior than the naive algorithm.

2 The Algorithm

Terms and words. Our algorithm is based on the representation of a term T of size n by two words of size n, W_T and L_T, already used by Luccio and Pagli [12]. For a term T, if x is the i^{th} vertex of T visited during a preorder traversal of T, then $W_T[i]$ is the label of x and $L_T[i] = i + |T_x|$, where $|T_x|$ is the number of vertices in the subtree of T rooted at x.



$W_T = b.c.a.a.e.d.a$
$L_T = 8.4.4.5.8.7.8$

Fig. 2. A term T and the associated words W_T and L_T

Notation 1. In the rest of this section we consider two terms T and P, respectively on the alphabets Σ and $\Sigma \cup \{V\}$. T has size n and P has size m.

Definition 1. We say that the k^{th} prefix of W_P ($W_P[1 \cdots k]$, $1 \leq k \leq m$) matches the factor $W_T[i \cdots j]$ of W_T ($1 \leq i \leq j \leq m$) if there are k integers $i_1 = i < i_2 < \cdots < i_k = j$ such that

- for every $\ell \in [k]$, $W_P[\ell] \neq V \Rightarrow W_T[i_\ell] = W_P[\ell]$,
- for every $\ell \in [k]$, $W_P[\ell] \neq V \Rightarrow i_{\ell+1} = i_\ell + 1$, and $W_P[\ell] = V \Rightarrow i_{\ell+1} = L_T[i_\ell]$.

Let v be a vertex of T and i be the position of the corresponding symbol in W_T. It is clear that there is an occurrence of P in T at v if and only if there is a factor $W_T[i \cdots j]$ of W_T such that the m^{th} prefix of W_P matches $W_T[i \cdots j]$. The following algorithm for the TPM problem is a simple iterative translation of the naive recursive algorithm in terms of words following from the above remark.

Algorithm 1: Naive algorithm for terms

for every $1 \leq i \leq n$ such that $W_T[i] = W_P[1]$ **do**
  $j := 2$ and $k := i + 1$;
  while $j \leq m$ **do**
1   **if** $W_T[k] = W_P[j]$ **then** $j := j + 1$ and $k := k + 1$;
    else if $W_P[j] = V$ **then** $j := j + 1$ and $k := L_T[k]$;
    else exit the **while** loop;
  if $j = m + 1$ **then** there is an occurrence of P at the i^{th} vertex of T;

Notation 2. We denote by comp_1 the number of comparisons between a symbol of W_T and a symbol of W_P (that is the number of times the step 1 of the above algorithm is performed). It is clear that this parameter gives also the asymptotic time complexity of the algorithm.

Suffix trees and pattern matching in terms. In the rest of this section, we propose an improvement of this algorithm, using the suffix tree of the word W_T.

We recall that the suffix tree of a word W, denoted by $ST(W)$, is a compressed digital search tree indexing the suffixes of W (see [7, chapter 5] for a survey on the suffix tree of a word and its applications). Every edge of $ST(W)$ is labeled by a factor of W, that can be represented by the position in W of its first and last symbols ((i, j) for a factor $W[i \cdots j]$). For a vertex u of $ST(W)$, we denote by $w(u)$ the word obtained by the concatenation of the labels of the edges on the path from the root of $ST(W)$ to u. For a word W and a vertex u of a suffix tree, we say that u accepts W if and only if $w(u) = W$ (see Figure 3).

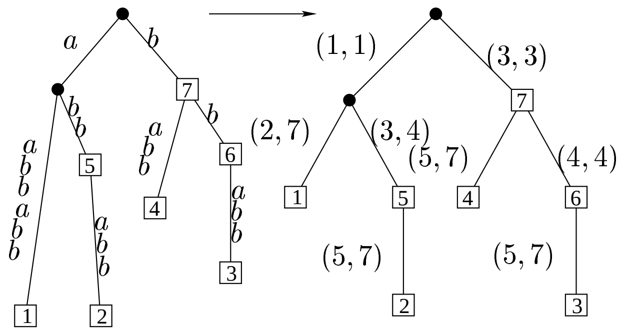


Fig. 3. The suffix tree of the word *aabbabb*

The correctness of the algorithm we propose below relies on the notion of valid pair in the suffix tree of the word encoding a target term. Valid pairs will be used to represent the occurrences of a pattern tree in the target tree in such a suffix tree.

Definition 2. A *valid pair of* $ST(W_T)$ is a pair (e, ℓ) where:

- e is an edge of $ST(W_T)$ (say labeled by (b, f)),
- ℓ an integer satisfying $0 \leq \ell \leq (f - b)$.

We denote by $w(e, \ell)$ the word $W_T[(b - |w(u)|) \cdots (b + \ell)]$.

Notation 3. From now, for a valid pair (e, ℓ) (resp. (e', ℓ'), (e'', ℓ'')) we denote by u and v (resp. u' and v', u'' and v'') the vertices incident by the edge e (resp. e', e''), where u (resp. u', u'') is the parent of v (resp. v', v''), and by (b, f) (resp. (b', f'), (b'', f'')) the label of this edge.

Definition 3. We say that a word W *matches a valid pair* (e, ℓ) of a suffix tree $ST(W_T)$ if W is equal to $w(e, \ell)$.

Example 1. In the suffix tree of Figure 3, if we denote by e the edge incident to the vertices labeled by 6 and 3, then the word *bbab* matches the valid pair $(e, 2)$.

The following property is the central to our pattern matching algorithm.

Property 1. Let $W_P[1 \cdots k]$ $(k < m)$ be a prefix of W_P (e, ℓ) and (e', ℓ') be two valid pairs of $ST(W_T)$ such that

- $W_P[k+1] = V$,
- the k^{th} prefix of W_P $(W_P[1 \cdots k])$ matches (e, ℓ),
- v' is a descendant of v,
- $w(e, \ell)$ is a proper prefix of $w(e', \ell')$.

If we denote by h the integer $|w(u')| - |w(u)| - (\ell + 1)$ (hence $W_P[1 \cdots k]$ matches $W_T[b' - h - 1 \cdots b']$), then $W_P[1 \cdots k+1]$ matches (e', ℓ') if and only if $L_T[b' - h] = b' + \ell' + 1$.

The algorithm. The principle of our algorithm is to read the word W_P in $ST(W_T)$, as usual when searching a suffix tree for the occurrences of a word, the variable V being handled in a special way defined following Property 1. Hence we need two procedures to traverse $ST(W_T)$: the first, called **Reading**, is the classical procedure which reads a word in a suffix tree, but when the current symbol of W_P equals the variable V, it calls the **Jump** procedure, which performs a recursive traversal of the subtree of $ST(W_T)$ rooted at the current vertex of this suffix tree, computing valid pairs accepting the prefix of W_P ending on this symbol V. The fact that the following algorithm computes all the occurrences of the pattern term P in the target term T is a direct consequence of Property 1.

Algorithm 2: Pattern matching in terms

Input: W_T, L_T, W_P and $ST(W_T)$
let r be the root of $ST(W_T)$;
if an edge e such that $W_T[b] = W_P[1]$ leaves r **then Reading**$(1, (e, 0))$;

Procedure Reading$(j, (e, \ell))$
$k := j$;
while $k \leq m$, $\ell \leq (f - b)$ and $W_T[b + \ell] = W_P[k]$ **do**
  $k := k + 1$ and $\ell := \ell + 1$;
if $k > m$ **then**
  (* *the word W_P has been completely read* *) traverse the subtree of $ST(W_T)$ rooted at v: visiting a vertex accepting the suffix $W_T[i \cdots n]$ indicates that there is an occurrence of P in T rooted at the vertex corresponding to the symbol $W_T[i]$;
else if $\ell > (f - b)$ **then**
  (* *the current edge e has been completely read* *) **if** an edge e' such that $W_T[b'] = W_P[k]$ leaves v **then Reading**$(k, (e', 0))$;
  if $W_P[k] = V$ **then for** every edge e'' other than e' leaving v **do**
    Jump $(k, e'', 0)$;
else (* *we exit the loop because $W_T[b + \ell] \neq W_P[k]$* *) **if** $W_P[k] = V$ **then Jump**$(k, e, -\ell)$;

Procedure Jump(k, e, h)
$j := L_T[b-h]$ (* *j: position of the next symbol of W_T to read (Property 1)**);
if $j \leq (f+1)$ **then** (* *this jump stops in the current edge e* *) **Reading**$(k+1, (e, j-b))$;
else (* *we must continue the jump in the suffix tree* *) **for** every edge e' leaving v **do** **Jump**$(k, e', h+(f-b+1))$;

Analysis of the algorithm. The preprocessing of the target (computing W_T, L_T and $ST(W_T)$) can be done in $O(n)$ space and $O(n \log|\Sigma|)$ time, using classical results on encoding of trees and suffix trees.

The time complexity of the pattern matching phase depends on three parameters:

- the number comp_2 of comparisons between symbols of the words W_T and W_P (which is the number of executions of the test $W_T[b+\ell] = W_P[k]$ in the loop marked 1 in **Reading**),
- the number of calls to **Jump**,
- the number of vertices of $ST(W_T)$ visited during the step 2 of **Reading**.

First, we notice that the number of vertices of $ST(W_T)$ visited during step 2 of **Reading** is at most two times the number of occurrences of P in T, which is less than or equal to n

Next, if we notice that between two consecutive calls to **Jump** on the same edge of $ST(W_T)$ there has been at least one comparison between symbols of W_T and W_P, we can state that the number of such calls is at most comp_2 plus the number of edges of $ST(W_T)$ visited during the pattern matching phase (which is at most $2n$).

Finally, the number comp_2 of comparisons between a symbol of W_T and a symbol of W_P is at most comp_1. This last observation allows us to say that the time complexity of our algorithm is $O(n + \mathrm{comp}_1)$, i.e. $O(n\ m)$.

In fact, we can express this time complexity in a more precise way. Let us denote, for any symbol $x \in \Sigma$, by $ST_x(W_T)$ the set of leaves u of $ST(W_T)$ such that the first letter of $w(u)$ is the symbol x. If the root of P is labeled with x, it is easy to see that

- $\mathrm{comp}_2 <= m\ |ST_x(W_T)|$,
- the number of calls to **Jump** is at most $\mathrm{comp}_2 + |ST_x(W_T)|$,
- the number of vertices of the suffix tree visited by the step 1 of **Reading** is at most $2\ |ST_x(W_T)|$.

Hence, if x is the label of the root of P, the time complexity of the pattern matching phase is $O(m\ |ST_x(W_T)|)$.

Finally, it follows from the fact that the complexity of the pattern matching phase is $O(n + \mathrm{comp}_1)$ and from the result of Flajolet and Steyaert [17] about the naive algorithm that, with uniformly distributed random terms, the expected time is at most $\Theta(n+m)$. Our experimental results (Section 3) suggest that the expected time of our algorithm is linear, with a better behavior than the naive algorithm.

Remark 1. We can slightly improve Algorithm 2, in time and space, by noticing that in the suffix tree $ST(W_T)$, we don't need, for a suffix $W_T[i \cdots n]$ of W_T, to store the entire suffix, but only the part $W_T[i \cdots L_T[i]-1]$, which represents the subtree of T rooted at the vertex corresponding to $W_T[i]$. Hence, we can remove some states in $ST(W_T)$, with the consequence that a vertex of this modified suffix tree can accept many identical words (corresponding to identical subtrees of T). This operation can clearly be done in linear time after the construction of the classical version of $ST(W_T)$ and allows to save some space (the suffix tree has less vertices) and some time during the execution of the step 2 of the **Reading** procedure.

3 Experimental Results

In this section, we present the experimental results obtained with the two algorithms presented above (the naive algorithm, which is known to be fast in practice, and our improved suffix tree algorithm).

Our implementation was done on a Pentium 400 MHz running LINUX, with the C language (using GCC 2.91.66). The time spent in every procedure was recorded using the GPROF 2.9.1 program. We focused on the pattern matching phase and not on the preprocessing phase (i.e. converting the trees in words and computing the suffix tree).

Like Flajolet and Steyaert [17], we considered binary terms (every internal vertex has degree 2 or 0) and three cases, according to the probability p for the leaves of a random pattern to be labeled with the variable V: if $p = 1/(|\Sigma_0|+1)$ where Σ_0 is the set of symbols of Σ having arity 0, if $p = 1/2$ or if $p = 1$ (this last case maximises the number of calls to **Jump**). Finally, our methodology was to generate at random thousands of targets of size 1000 or 10000 and search twenty small random patterns (of size from 4 to 40 vertices) in every target.

For the naive algorithm, we recorded the following statistics: the number comp_1 of comparisons between symbols of W_T and W_P (line 1 of the naive algorithm) and the time t_1 spent in this algorithm. For the improved algorithm, we recorded the following statistics: the number comp_2 of comparisons between symbols of W_T and W_P (line 1 of the improved algorithms), the number jumps of calls to the **Jump** procedure and the time t_2 spent during the pattern matching phase.

Finally, we may add that we used an implementation of suffix trees as binary trees, which is equivalent to consider that the branching structure (the structure used to store and retrieve the edges leaving a node) is an unsorted list.

In the following table, we give the average value of comp_1, comp_2, jumps and the value t_1/t_2 (we display from right to left, in each cell of the table, the values of these statistics for $p = 1/(|\Sigma_0|+1)$, $p = 1/2$ and $p = 1$).

These experimental data suggest that our algorithm is more efficient, in practice, than the naive algorithm, even if, when $p = 1$, the two algorithms have a similar behavior.

n	$\|\Sigma\|$	comp$_1$	comp$_2$	jumps	t_1/t_2
1000	2	3561/3318/3318	1294/536/536	817/297/297	$1,18/2,81/2,81$
1000	3	1237/1115/1075	459/178/123	1091/381/257	$1,19/2,82/4,16$
1000	4	647/577/545	242/91/50	1255/437/231	$1,21/2,83/4,79$
1000	5	404/359/334	151/57/26	1362/475/209	$1,25/2,86/5,83$
1000	6	334/297/274	124/47/18	1444/509/182	$1,3/2,9/6,52$
10000	2	35604/33166/33166	11153/4446/4446	7238/2575/2575	$1,01/2,28/2,28$
10000	3	12380/11137/10731	3985/1467/987	9730/3272/2206	$1,04/2,46/3,38$
10000	4	6458/5759/5434	2080/757/391	11103/3745/1949	$1,09/2,51/4,32$
10000	5	4035/3583/3335	1301/471/200	12156/4113/1743	$1,11/2,56/5,08$
10000	6	3339/2960/2727	1077/391/138	12922/4414/1583	$1,12/2,58/5,99$

Table 1. Experimental results for binary terms. First column: number of symbols comparisons with Algorithm 1. Second column: number of symbols comparisons with Algorithm 2. Third column: number of calls to **Jump** in Algorithm 2. Fourth column: time spent by Algorithm 1 / time spent by Algorithm 2.

4 Conclusion

In this paper, we propose a simple algorithm for matching linear patterns in static terms. They are, as far as we know, the first devoted to this precise problem and seem to be very efficient in practice. We believe that these simple algorithms are good candidates for the case where many pattern matching requests are dealing with the same target term. Among the works remaining to do on this problem of pattern matching in static trees, we can highlight some problems.

It would be interesting to study the practical behavior of our algorithm on real data and not only random binary terms.

Among other extensions, we can think to apply the principle of our algorithm in the design of algorithms for other TPM problems dealing with other kind of trees (like the non-linear pattern matching in terms, or the TPM problem, for general ordered trees, introduced in [4]).

Can we improve the theoretical analysis of our algorithm in order to understand its good practical behavior? For instance, it should be possible to study the average behavior of the parameter comp$_2$ using the method of Flajolet and Steyaert [17].

From an implementation point of view, it would be efficient to give a non recursive version of our algorithms, which implies using a $O(n)$ extra-space but would surely greatly improve their performances.

References

1. A. V. Aho and M. Ganapathi. Efficient tree pattern matching : an aid to code generation. In *ACM Symposium on Principles of Programming Languages, POPL'85*, pages 334–340. Assoc. Comput. Mach. Press, 1985.
2. M. S. Boyer and J. S. Moore. A fast string-searching algorithm. *Comm. ACM*, 20:762–772, 1977.

3. J. Cai, R. Paige, and R. E. Tarjan. More efficient bottom-up multi-pattern matching in trees. *Theoret. Comput. Sci.*, 106(1):21–60, 1992.
4. C. Chauve. Tree pattern matching with a more general notion of occurrence of the pattern. *Inform. Process. Letters*, 82(4):197–201, 2002.
5. R. Cole and R. Hariharan. Tree pattern matching and subset matching in randomized $O(n \log^3 m)$ time. In *ACM Symposium on Theory of Computing, STOC'97*, pages 66–75. Assoc. Comput. Mach. Press, 1997.
6. R. Cole, R. Hariharan, and P. Indyk. Tree pattern matching and subset matching in deterministic $O(n \log^3 n)$-time. In *ACM-SIAM Symposium on Discrete Algorithms, SODA'99*, pages 245–254. Assoc. Comput. Mach. Press, 1999.
7. M. Crochemore and W. Rytter. *Text algorithms.* Oxford University Press, New York, 1994.
8. M. Dubiner, Z. Galil, and E. Magen. Faster tree pattern matching. *J. Assoc. Comput. Mach.*, 41(2):205–213, 1994.
9. C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *J. Assoc. Comput. Mach.*, 29(1):68–95, 1982.
10. D. E. Knuth, J. H. Jr. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.
11. S. R. Kosaraju. Efficient tree pattern matching. In *IEEE Symposium on Foundations of Computer Science, FOCS'89*, pages 178–183. IEEE Comput. Soc. Press, 1989.
12. F. Luccio and L. Pagli. An efficient algorithm for some tree matching problems. *Inform. Process. Lett.*, 39(1):51–57, 1991.
13. F. Luccio and L. Pagli. Simple solutions for approximate tree matching problems. In *Theory and Practice of Software, TAPSOFT '91*, volume 493 of *Lecture Notes in Comput. Sci.*, pages 193–201. Springer, 1991.
14. F. Luccio and L. Pagli. Approximate matching for two families of trees. *Inform. and Comput.*, 123(1):111–120, 1995.
15. B. A. Shapiro and K. Zhang. Comparing multiple RNA secondary structures using tree comparisons. *Comput. Appl. Biosci.*, 6:309–318, 1990.
16. T. Shibuya. Constructing the suffix tree of a tree with a large alphabet. In *International Symposium on Algorithms and Computation, ISAAC'99*, volume 1741 of *Lecture Notes in Comput. Sci.*, pages 225–236. Springer, 1999.
17. JM. Steyaert and P. Flajolet. Patterns and pattern-matching in trees: an analysis. *Inform. and Control*, 58(1-3):19–58, 1983.
18. M. Thorup. Efficient preprocessing of simple binary pattern forests. *J. Algorithms*, 20(3):602–612, 1996.
19. K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.