King's College London

Department of Informatics

Nine-Month Progress Report

Author:
Michalis Christou

Supervisors:
Prof Costas Iliopoulos
Prof Maxime Crochemore

June 2011

Abstract

This report contains results that have been accepted in conference proceedings during the first nine months of my PhD. So far I have studied a lot around strings and problems around them. More specifically I am showing an $\mathcal{O}(|y|)$ time algorithm that computes the minimal left-seed array of a given string y, an $\mathcal{O}(|y| \log |y|)$ time algorithm that computes the minimal right-seed array of a given string y, and a linear-time solution to compute the maximal left-seed/right-seed array by detecting border-free prefixes of the given string. Additionally I am presenting a problem on trees, how to output all subtree repeats of a tree in linear time using a string representation of the tree.

Keywords: algorithms, strings, trees, periodicity, covers, seeds

Acknowledgments

Thanks are extended to my family which has always been close to me and to my supervisors, Prof Costas Iliopoulos and Prof Maxime Crochemore, for their continuous supervision and guidance through the completion of this report.

Disclaimer

I declare the following to be my own work, unless otherwise referenced, as defined by the Universitys policy on plagiarism.

Michalis Christou

Contents

1. Introduction

Strings appear in many fields of Mathematics and Computer Science like combinatorics on words, pattern matching, data compression and automata theory (see [29, 30]), because of their paramount importance in several applications and their theoretical aspects.

Throught this report I am showing results around my research in strings that have been accepted in conference proceedings during the first nine months of my PhD. More specifically in Section 2 I am showing some results around quasiperiodicity in strings, an $\mathcal{O}(|y|)$ time algorithm that computes the minimal left-seed array of a given string y[8], an $\mathcal{O}(|y| \log |y|)$ time algorithm that computes the minimal right-seed array of a given string y[7], and a linear-time solution to compute the maximal left-seed/right-seed array by detecting border-free prefixes of the given string[7, 8]. Additionally I am presenting a problem on trees in Section 3, how to output all subtree repeats of a tree in linear time using a string representation of the tree[6]. Finally, I give some future proposals and a brief conclusion in Section 4.

2. Left-seed and right-seed arrays

2.1. Introduction

The notion of periodicity in strings is well studied in many fields like combinatorics on words, pattern matching, data compression and automata theory (see [29, 30]), because it is of paramount importance in several applications, not to talk about its theoretical aspects.

The concept of quasiperiodicity is a generalization of the notion of periodicity, and was firstly defined by Apostolico and Ehrenfeucht in [3]. In a periodic repetition, the occurrences of the single periods do not overlap. In contrast, the quasiperiods of a quasiperiodic string may overlap. We call a factor u of a nonempty string y a cover of y, if every letter of y is within some occurrence of u in y. Note that we consider the *aligned covers*, where the cover u of y needs to be a border (i.e. a prefix and a suffix) of y. *Seeds* are regularities of strings strongly related to the notion of cover. They were first defined and studied by Iliopoulos, Moore and Park in [23]. The notion of seed is an extension of the definition of a cover of a string, as it is a cover of a superstring of the word.

A fundamental problem is to find all the covers of a string y of length n. Linear-time algorithms were given by Moore and Smyth in [32], and by Li

and Smyth in [28]. An $\mathcal{O}(\log(\log n))$-time work-optimal parallel algorithm was given later by Iliopoulos and Park in [22]. The corresponding problem on seeds is of much higher algorithmic difficulty, and at the moment, no linear-time algorithm is known. The fastest algorithm was given by Iliopoulos, Moore and Park in [23], running in $\mathcal{O}(n \log n)$ time.

A close and also fundamental problem is that of computing the shortest (resp. longest) cover of every prefix of a string. This gives rise to the so-called *minimal cover* (resp. *maximal cover*) *array*. An integer array C is the minimal cover (resp. maximal cover) array of y, if $\mathsf{C}[i]$ is the minimal (resp. maximal) length of covers of $y[0\,..\,i]$.

Apostolico and Breslauer, in [2, 5], gave an online linear-time algorithm for computing the minimal cover array of a string, using the algorithm by Knuth, Morris and Pratt, in [25], for computing the period of every prefix of a string in linear time. In addition, Li and Smyth, in [28], provided an algorithm, having the same characteristics, for computing the maximal cover array of a given string; this algorithm gives also all the covers for every prefix of the string.

A left seed of a string y of length n, firstly defined in [8], is a prefix of y that is a cover of a superstring of y. Similarly, a right seed of a string y, also firstly defined in [8], is a suffix of y that is a cover of a superstring of y. An integer array LS is the minimal left-seed (resp. maximal left-seed) array of y, if $\mathsf{LS}[i]$ is the minimal (resp. maximal) length of left seeds of $y[0\,..\,i]$. The minimal right-seed array (resp. maximal right-seed) RS is defined in a similar fashion. These structures, apart from their own theoretical interest, they might also prove useful in finding faster algorithms for problems on seeds.

In this report we summarise our previous results obtained for the left-seed and the right-seed arrays: a linear-time algorithm for computing the minimal left-seed array of y[8], a linear-time solution for computing the maximal left-seed array of y[8], an $\mathcal{O}(n \log n)$ time algorithm for computing the minimal right-seed array of y[7] and a linear-time solution for computing the maximal right-seed array of y[7]. It is important to note that all of the proposed algorithms use linear auxiliary space.

The rest of the section is structured as follows. In Subsection 2.2, we present the basic definitions used throughout the report, and we define the problems solved. In Subsection 2.3, we prove some useful properties of left seeds, right seeds, and covers, used later on for the design and analysis of the provided algorithms. In Subsection 2.4, we describe and analyse the proposed

algorithms. Finally, we give some future proposals and a brief conclusion in Section 4.

2.2. Definitions and Problems

Throughout this report we consider a string y of length $|y| = n$, $n > 0$, on a fixed alphabet. It is represented as $y[0\,..\,n-1]$. A string w is a *factor* of y if $y = uwv$ for two strings u and v. It is a *prefix* of y if u is empty and a *suffix* of y if v is empty. The concatenation of two strings x and y is the string of the letters of x followed by the letters of y. It is denoted by xy. A string u is a *border* of y if u is a both a prefix and a suffix of y. *The border* of y, denoted by $\mathrm{Border}(y)$, is the length of the longest border of y. The *border array* B of y is the array of integers $\mathsf{B}[0\,..\,n-1]$ for which $\mathsf{B}[i]$, $0 \leq i < n$, stores the length of the border of the prefix $y[0\,..\,i]$. A string u is a *period* of y if y is a prefix of u^k for some positive integer k, or equivalently if y is a prefix of uy. *The period* of y, denoted by $\mathrm{Per}(y)$, is the length of the shortest period of y. It is a known fact that, for any string y, $\mathrm{Per}(y) + \mathrm{Border}(y) = |y|$. The *period array* P of y is the array of integers $\mathsf{P}[0\,..\,n-1]$ for which $\mathsf{P}[i]$, $0 \leq i < n$, stores the length of the period of the prefix $y[0\,..\,i]$. For a string $u = u[0\,..\,m-1]$ such that u and v share a common part $u[m-\ell\,..\,m-1] = v[0\,..\,\ell-1]$ for some $1 \leq \ell \leq m$, the string $u[0\,..\,m-1]v[\ell\,..\,n-1] = u[0\,..\,m-\ell-1]v[0\,..\,n-1]$ is called a *superposition* of u and v with an *overlap* of length ℓ. A string x of length m is a *cover* of y if both $m < n$ and there exists a set of positions $P \subseteq \{0, \ldots, n-m\}$ that satisfies both $y[i\,..\,i+m-1] = x$ for all $i \in P$ and $\bigcup_{i \in P}\{i, \ldots, i+m-1\} = \{0, \ldots, n-1\}$. The *minimal cover array* C of y is the array of integers $\mathsf{C}[0\,..\,n-1]$ for which $\mathsf{C}[i]$, $0 \leq i < n$, stores the length of the shortest cover of the prefix $y[0\,..\,i]$. The *maximal cover array* C of y is the array of integers $\mathsf{C}[0\,..\,n-1]$ for which $\mathsf{C}^{\mathrm{M}}[i]$, $0 \leq i < n$, stores the length of the longest cover of the prefix $y[0\,..\,i]$ which is smaller than y (0 if none).

A string v is a *seed* of y, if it is a cover of a superstring of y. A left seed of a string y is a prefix of y that it is a cover of a superstring of y of the form yv, where v is a possibly empty string. The *minimal left seed* of y, denoted by $\mathrm{mls}(y)$, is the shortest prefix of y that is a cover of a superstring of y. The *minimal left-seed array* LS of y is the array of integers $\mathsf{LS}[0\,..\,n-1]$ for which $\mathsf{LS}[i]$, $0 \leq i < n$, stores the length of the minimal left seed of the prefix $y[0\,..\,i]$. The *maximal left seed* of y, denoted by $\mathrm{Mls}(y)$, is the longest prefix of y that is a cover of a superstring of y. The *maximal left-seed array* LS^{M} of y is the array of integers $\mathsf{LS}^{\mathrm{M}}[0\,..\,n-1]$ for which $\mathsf{LS}^{\mathrm{M}}[i]$, $0 \leq i \leq n-1$, stores the length of the maximal left seed of the prefix $y[0\,..\,i]$, which is smaller

than $i+1$ (0 if none). The *minimal right seed* of y, denoted by $\mathrm{mrs}(y)$, is the shortest suffix of y that is a cover of a superstring of y. The *minimal right-seed array* RS of y is the array of integers $\mathsf{RS}[0\,..\,n-1]$ for which $\mathsf{RS}[i]$, $0 \le i < n$, stores the length of the minimal right seed of the prefix $y[0\,..\,i]$. The *maximal right seed* of y, denoted by $\mathrm{Mrs}(y)$, is the longest suffix of y that is a cover of a superstring of y. The *maximal right-seed array* RS^{M} of y is the array of integers $\mathsf{RS}^{\mathrm{M}}[0\,..\,n-1]$ for which $\mathsf{RS}^{\mathrm{M}}[i]$, $0 \le i \le n-1$, stores the length of the maximal left seed of the prefix $y[0\,..\,i]$, which is smaller than $i+1$ (0 if none).

The following example shows B, P, C, C^{M}, LS, LS^{M}, RS, and RS^{M} for the string $y = \mathtt{abaababaabaabab}$ and all its left and right seeds.

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$y[i]$	a	b	a	a	b	a	b	a	a	b	a	a	b	a	b
$\mathsf{B}[i]$	0	0	1	1	2	3	2	3	4	5	6	4	5	6	7
$\mathsf{P}[i]$	1	2	2	3	3	3	5	5	5	5	5	8	8	8	8
$\mathsf{C}[i]$	1	2	3	4	5	3	7	3	9	5	3	12	5	3	15
$\mathsf{C}^{\mathrm{M}}[i]$	0	0	0	0	0	3	0	3	0	5	6	0	5	6	0
$\mathsf{LS}[i]$	1	2	2	3	3	3	3	3	3	3	3	3	3	3	13
$\mathsf{LS}^{\mathrm{M}}[i]$	0	0	2	3	4	5	6	7	8	9	10	11	12	13	14
$\mathsf{RS}[i]$	1	2	2	3	3	3	5	3	5	5	3	8	5	3	8
$\mathsf{RS}^{\mathrm{M}}[i]$	0	0	2	3	4	5	6	7	8	9	10	11	12	13	14

All left seeds of y: aba, abaaba, abaababa, abaababaa, abaababaab, abaababaaba, abaababaabaa, abaababaabaab, abaababaabaaba, abaababaa baabab

All right seeds of y: aabaabab, baabaabab, abaabaabab, babaabaabab, ababaabaabab,aababaabaabab, baababaabaabab, abaababaabaabab

We consider the following problems for a given string y.

Problem 1 (Computing all left seeds) *Compute all the left seeds of y.*

Problem 2 (Computing all right seeds) *Compute all the right seeds of y.*

Problem 3 (Computing the minimal left-seed array) *Compute the minimal left-seed array LS of y.*

Problem 4 (Computing the maximal left-seed array) *Compute the maximal left-seed array* LS^{M} *of* y.

Problem 5 (Computing the minimal right-seed array) *Compute the minimal right-seed array* RS *of* y.

Problem 6 (Computing the maximal right-seed array) *Compute the maximal right-seed array* RS^{M} *of* y.

2.3. Properties

In this subsection, we prove some useful properties of left seeds, of right seeds, and of covers of a given string y.

Lemma 2.1 ([8]) *A string z is a left seed of y iff it covers a prefix of y whose length is at least the period of y.*

Proof Direct: Suppose a string z covers a prefix of y, say uv, larger or equal to $\mathrm{Per}(y)$, where $|u| = \mathrm{Per}(y)$ and v is a possibly non empty string. Let k the smallest integer such that y is a prefix of u^k. Then z is a cover of $u^k v = ywv$, for some string w, possibly empty. Therefore z is a left seed of y.
Reverse: Let z a left seed of y.

- if $|z| \leq \mathrm{Border}(y)$. Then a suffix v of z, possibly empty, is a prefix of the border (consider the left seed that covers $y[\mathrm{Per}(y) - 1]$). Then z is a cover of uv, where u is the period of y.
- if $|z| > \mathrm{Border}(y)$. Let z not a cover of a prefix of y larger or equal to $\mathrm{Per}(y)$. Let v a border of y such that $|v| = \mathrm{Border}(y)$. Then v is a factor of z, such that $z = uvw$, where u and w are non empty words (consider the left seed that covers $y[\mathrm{Per}(y)-1]$). This gives uv a longest border for y, which is a contradiction. □

Lemma 2.2 ([7]) *A string z is a right seed of y iff it covers a suffix of y whose length is at least the period of y.*

Proof Direct consequence of Lemma 2.1. □

Lemma 2.3 ([32]) *Let u be a proper cover of x, in our case some prefix $y[0\,..\,i]$ of y, and let $z \neq u$ be a substring of x such that $|z| \leq |u|$. Then z is a cover of x if and only if z is a cover of u.*

Proof Clearly if z is a cover of u and u a cover of x then z is a cover of x. Suppose now that both z and u cover x. Then z is a border of x and hence of u ($|z| \leq |u|$); thus z must also be a cover of u. □

Lemma 2.4 ([8]) *Let y a string of length n and $\mathrm{Per}(y) = n$, then the minimal left seed of y is y.*

Proof By definition of the minimal left seed: $|\mathrm{mls}(y)| \leq n$. Let $|\mathrm{mls}(y)| < n$. Then, in order to cover y, a non-empty prefix of $\mathrm{mls}(y)$, say w, is a suffix of y (consider the left seed that covers $y[n-1]$). Then $n - |w|$ gives a shorter period for y, which is a contradiction. □

Lemma 2.5 ([8]) *Let $\mathsf{LS}[i] = |\mathrm{mls}(y[0\,..\,i])|$, for all $0 \leq i < n$, then $\mathsf{LS}[i] \leq \mathsf{LS}[i+1]$, for all $0 \leq i < n-1$.*

Proof Let $\mathsf{LS}[i] > \mathsf{LS}[i+1]$. By definition of the minimal left seed: $\mathsf{LS}[i]$ covers some superstring $y[0\,..\,i]u$, where u is a possibly empty string, and $\mathsf{LS}[i+1]$ covers some superstring $y[0\,..\,i+1]v$, where v is a possibly empty string. In other words $\mathsf{LS}[i{+}1]$ covers $y[0\,..\,i]y[i{+}1]v$. Therefore, by definition of the minimal left seed, $\mathsf{LS}[i+1]$ is a minimal left seed of $y[0\,..\,i]$. But then we get a shorter left seed for $y[0\,..\,i]$, which is a contradiction. □

Lemma 2.6 ([8]) *Let y a string of length n and $\mathrm{Per}(y) = k$, then*

- *if $k = n$, then there is no maximal left seed for y*
- *if $k < n$, then the maximal left seed of y is $y[0\,..\,n-2]$*

Proof

- if $k = n$, then, by definition of the maximal left seed, it holds that $|\mathrm{mls}(y)| < n$. Let $y[0\,..\,j]$ be the maximal left seed of y, with $0 \leq j < n$. Then, in order to cover y, a non-empty prefix of $y[0\,..\,j]$, say w, is a suffix of y (consider the maximal left seed that covers $y[n-1]$). Then $n - |w|$ gives a shorter period for y, which is a contradiction.
- if $k < n$, then the maximal left seed of y is $y[0..n-2]$, as it is a cover of the superstring $yy[\mathrm{Border}(y)\,..\,n-2]$ of y, and it has the maximum length allowed, which is $n-1$. □

Lemma 2.7 ([7]) *Let y a string of length n and $\mathrm{Per}(y) = k$, then*

- *if $k = n$, then there is no maximal right seed for y.*
- *if $k < n$, then the maximal right seed of y is $y[1..n-1]$*

Proof Similar to the proof of Lemma 2.6. □

2.4. The algorithms

In this subsection, we describe our algorithms for solving Problems 1-6 for a string y of length n.

2.4.1. Computing the minimal and the maximal left-seed array

In this subsection, we describe our algorithms for solving Problem 3 and Problem 4. For solving Problem 3, we use the algorithm for computing the minimal cover array by Apostolico and Breslauer [2, 5] in linear time, and Lemma 3.1, which gives the necessary and sufficient condition for a prefix of a string to be a left seed of that string. Problem 4 reduces to detecting border-free prefixes of the given string.

Algorithm R-ARRAY computes an array R, which stores the length of the longest prefix of y that is covered by $y[0\,..\,i]$, 0 if none. It takes as input the minimal cover array C.

ALGORITHM R-ARRAY(C, n)
1: **for** $i \leftarrow 0$ **to** $n-1$ **do** R$[i] \leftarrow 0$;
2: **for** $i \leftarrow 0$ **to** $n-1$ **do** R[C$[i]-1] \leftarrow i+1$;
3: **return** R$[0\,..\,n-1]$;

Algorithm MINIMAL-LEFT-SEED-ARRAY computes an array LS, which stores the length of the minimal left seed of $y[0\,..\,i]$. It takes as input the array R and the period array P.

ALGORITHM MINIMAL-LEFT-SEED-ARRAY(y, n, P,R)
1: **for** $i \leftarrow 0$ **to** $n-1$ **do**
2: **if** P$[i] \leftarrow i+1$ **do** LS$[i] \leftarrow i+1$;
3: **else**
4: $ls \leftarrow$ LS$[i-1]$;
5: **while** R$[ls-1] <$ P$[i]$ **do** $ls \leftarrow ls+1$;
6: LS$[i] \leftarrow ls$;
7: **return** LS$[0\,..\,n-1]$;

Then, we proceed to compute the array LS. There are two cases:

- if P$[i] = i+1$, then by Lemma 2.4, LS$[i] = i+1$

- if $\mathsf{P}[i] < i+1$, then by Lemma 2.5, we must look for $\mathsf{LS}[i]$ in the set $\{\mathsf{LS}[i-1], ..., n\}$

Lemma 2.1 gives the necessary and sufficient condition ($\mathsf{R}[ls-1] \geq \mathsf{P}[i]$), i.e. the length of the longest prefix of y that is covered by $y[0\,..\,ls-1]$ to be greater or equal than the period of $y[0\,..\,i]$, where ls is the candidate length of the minimal left seed of $y[0\,..\,i]$.

Theorem 2.8 *Computing the minimal left-seed array of a string y of length n can be done in linear time.*

Proof Arrays C, P and R can be computed in linear time as of [11, 25, 2, 5]. Then, we only have to prove that the process to compute array LS runs in linear time, as well. Let c_i be the number of comparisons done in the second loop at step i. Then clearly,
T=Time needed
$= 2n + 2 + \sum_{i=0}^{n-1}(1 + I(\mathsf{P}[i] = i+1) + I(\mathsf{P}[i] \neq i+1) * (2 + 2c_i - 1))$
$= 2n + 2 + \sum_{i=0}^{n-1}(2 + I(\mathsf{P}[i] \neq i+1) * 2c_i)$
$= 2n + 2 + 2n + I(\mathsf{P}[i] \neq i+1) \sum_{i=0}^{n-1} 2c_i$
Therefore

$$T \leq 4n + 2 + 2\sum_{i=0}^{n-1} c_i \tag{1}$$

Let s_i be the set of values checked in the second loop at step i. Clearly $s_i \cap s_{i+1} = 1$ given by the 1st value to be checked in s_{i+1}, which is the same as the last value in s_i. Then

$$c_i + c_{i+1} = |s_i| + |s_{i+1}| = |s_i \cup s_{i+1}| + |s_i \cap s_{i+1}| \tag{2}$$

$$c_i + c_{i+1} = |s_i \cup s_{i+1}| + 1 \tag{3}$$

By Equations 1 and 3:

$$T \leq 4n + 2 + 2(n - 1 + |\bigcup_{i=0}^{n-1} s_i|) \tag{4}$$

As $\bigcup_{i=0}^{n-1} s_i$ is at most $\{0, ..., n-1\}$ then $|\bigcup_{n-1}^{i=0} s_i| \leq n$, and so $T \leq 8n$. Therefore $T = \mathcal{O}(n)$. □

For solving Problem 4, we can use Lemma 2.6 to obtain the following two cases.

- if $\mathsf{P}[i] = i + 1$, then $\mathsf{LS}^{\mathrm{M}}[i] = 0$
- if $\mathsf{P}[i] < i + 1$, then $\mathsf{LS}^{\mathrm{M}}[i] = i$

Hence, we obtain the following result.

Theorem 2.9 *Computing the maximal left-seed array of a string y of length n can be done in linear time.*

2.4.2. Computing the minimal and the maximal right-seed array

In this subsection, we describe our algorithms for solving Problem 5 and Problem 6. For solving Problem 5, we use a variant of the partitioning algorithm in [10, 23] to find the sets of ending positions of all the occurrences of each factor in the string. We are then able to find which suffix of each prefix of the string is covered by that factor, and check for right seeds using Lemma 2.2. Problem 6 reduces to detecting border-free prefixes of the given string.

In the following lines, we give a brief description of the partitioning algorithm used for solving Problem 5.

For a factor w in y, the set of end positions of all the occurrences of w in y, gives us the *end set* of w. We define an equivalence relation $\approx_\ell$ on positions of y such that $i \approx_\ell j$ iff $y[i-\ell+1\,..\,i] = y[j-\ell+1\,..\,j]$. Therefore, depending on the length of the factor, we get equivalence classes for each length ℓ, $1 \leq \ell \leq n$. Equivalence classes for $\ell = 1$ are found by going over y once, and keeping the occurrences of each letter in separate sets. For larger ℓ, we consider classes of the previous level to make a refinement on them, and calculate the classes of that level. So on level ℓ, $1 < \ell \leq n$, we refine a class C by a class D, by splitting C in classes $\{i \in C / i-1 \in D\}, \{i \in C / i-1 \notin D\}$. In order to achieve a good running time, we do not use all classes for refinement; only classes of the previous level, which were split two levels before, are used. From those, we can omit the largest siblings of each family, and use only the small ones for the computation. We terminate the algorithm when all classes reach a singleton stage.

In Fig. 5, the classes of $\approx_\ell$, $\ell = 1, 2, \ldots, 8$, for the string $y =$ `abaababaa baabab` are illustrated. The partitioning algorithm creates two types of sets at each round of equivalent classes; the old and the new ones (see Fig. 5). A

set is called *old*, if it has been created by deletion of elements from its parent set, i.e. it consists of remaining elements only. A set is called *new*, if it is composed from deleted elements from its parent set. Hence, the partitioning algorithm on y will give us distinct end sets E_i with their corresponding factor length range $(\ell_{\min_i}, \ell_{\max_i})$, as shown in the example below.

Example Let us consider $y =$ `abaababaabaabab`, the example string from Fig. 5. Then, in level order, E_3 is the set $\{3, 8, 11\}$ with corresponding factor length range $(2, 4)$. □

Proposition 2.10 ([23]) *The number of distinct end sets in the partitioning for a string of length n is $\mathcal{O}(n)$.*

Proposition 2.11 ([10]) *The complexity of the partitioning algorithm for a string of length n is $\mathcal{O}$(number of elements of new sets), where the number of elements of new sets in the partitioning is bounded above by $n \log n$.*

While executing the partitioning algorithm, we also maintain the following data structures:

- For each end set $E_i = \{a_1, a_2, .., a_k\}$, with corresponding factor length ℓ, of y, such that $k > 1$.
    1. $Gap_{i,\ell}$ list
    $$Gap_{i,\ell}(a_j) = a_{j+1} - a_j, j \in \{1, 2, \ldots, k-1\} \tag{5}$$
    2. $cover_{i,\ell}$ value
    An end set E_i, with corresponding factor length ℓ, is said to have $cover_{i,\ell}$ value equal to the first element of the set ,say a_j, such that $Gap_{i,\ell}(a_j) > \ell$, otherwise a value equal to the last element of the set. This value gives the last element coverable in each set by its equivalent factor, starting from the first element of the set.

- For each distinct end set $E_i = \{a_1, a_2, .., a_k\}$, with corresponding factor length range $(\ell_{\min_i}, \ell_{\max_i})$, of y, such that $k > 1$.
    1. CV_i array
    An array CV_i, such that $\mathsf{CV}_i[j]$ is the j^{th} $cover_{i,\ell}$ value, for some $\ell_{\min_i} \leq \ell \leq \ell_{\max_i}$, appeared in E_i in ascending order.

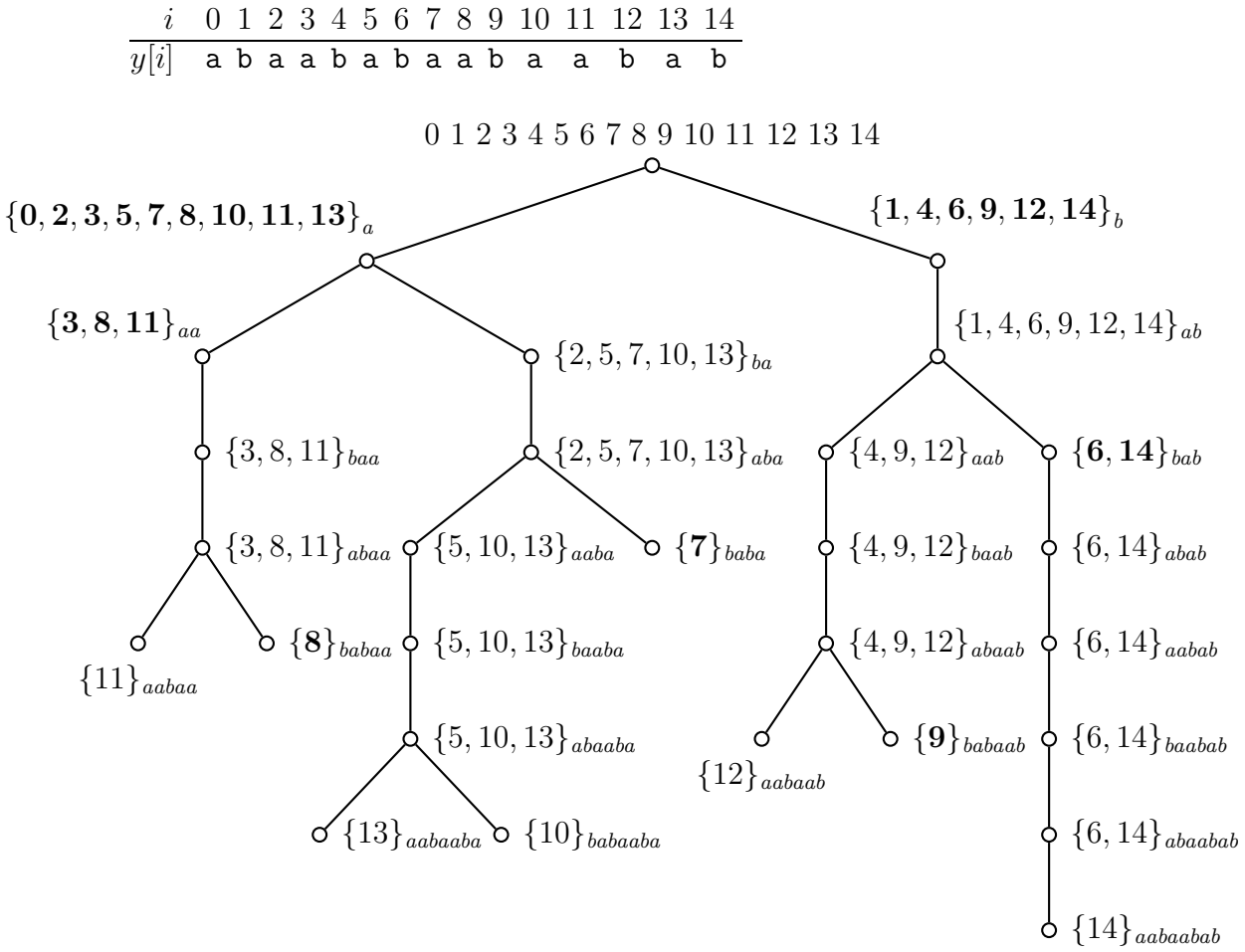


Figure 1: Classes of equivalence and their refinements for string $y =$ abaababaabaabab. The sets considered for the computation of the partitioning are shown in bold.

2. FL_i array
   An array FL_i, such that $\mathsf{FL}_i[j] = \ell$, where ℓ is the factor length of E_i when the j^{th} $cover_{i,\ell}$ value, for some $\ell_{\min_i} \leq \ell \leq \ell_{\max_i}$, first appears.
3. $first_i$ value
   A value $first_i$, which gives the first element that has not been

assigned a minimal right seed in the set or any of its ancestors.

Below, we show how to update the above mentioned data structures in time $\mathcal{O}(n \log n)$.

- **In new sets:**
  When a new set of size m is created, we can update all relevant data structures in $\mathcal{O}(m)$ time. By running over the elements of the set in order, we can easily update the *cover* value, *Gap* list, and the CV and FL arrays.

- **In old sets:**
  An old set is created from its parent set, by deletion of d elements from it. When an element a_j is deleted from its parent set, we can easily update the *Gap* values of its neighbors. All these operations take time $\mathcal{O}(d)$, where d is the number of elements of new sibling sets of the old set. Updating the *cover* value is more difficult, as new data structures have to be created, but is still under control, as of Lemma 2.12, below. The CV and FL arrays are updated when a new *cover* value is found.

Lemma 2.12 *In a chain of old sets with a new set of size m as a root, the the cover values are calculated in the whole chain in $\mathcal{O}(m)$ time.*

Proof We create a queue Q that initially includes the *Gap* list of the first old set in the chain. When a new gap is created by deletion of an element in subsequent old sets, two gaps are merged. Then, we stack the new gap on top of the second gap, that is used to form it in Q, and so on (those gaps have the same element on their right edge). We also keep pointers on the new gaps, formed in the corresponding *First-appear*(ℓ) list (those lists are kept in a different queue), where ℓ is the length of the corresponding factor of the set. Pointers for deleted gaps are added in the corresponding *Delete*(ℓ) list (those lists are kept in a different queue, as well).

We then go over the *First-appear* and *Delete* queues, which mark the beginning of a distinct old set. Gaps in the corresponding *First-appear*(ℓ) list are moved to the right of the element at their bottom, and gaps that are in the corresponding *Delete*(ℓ) list are deleted, maintaining the structure of Q. We then search for *cover* value in Q, by popping out gaps from Q, until the first gap in Q is greater than ℓ (*cover* value is the element on the left edge of last gap considered), or Q becomes empty (*cover* value is the element on

the right edge of the last gap considered). Whenever a gap, that has stacked elements on it, is popped out of Q, its stacked elements are passed to an element out of the queue called *start*, whose right element is taken to be the first gap in Q . If *cover* value found is smaller than next length to be encountered in *First-appear* queue, we check for more *cover* value changes as before.

It is easy to observe that Q has at most $2s - 1$ elements, where s is the size of the *Gap* of the first old set in the chain. Thus, we can create the lists *First-appear*, *Delete* and queue Q in $\mathcal{O}(m)$ time, as $s < m$. We check for *cover* value when a distinct old set is encountered for first time (at most m such sets). While inside that set, *cover* value changes are made iff the gap after a *cover* value gets equal to the factor length (at most $|Q|$ such cases). Failed attempts are made at most once for each element of Q, therefore at most $|Q|$ failed attempts. □

Therefore, the maintenance of the above data structures takes time $\mathcal{O}(number\ of\ elements\ of\ new\ sets)$, where the number of elements of new sets in the partitioning is bounded above by $n \log n$, as of Proposition 2.11.

Example Consider the example string from Fig. 5. Starting with the set for factor `a`, and continuing just with old sets (actually the first set in not exactly an old set here), we get the data structures in Fig. 2.

$\ell = 1$	0	2	3	5	7	8	10	11	13
$\ell = 2$		2		5	7		10		13
$\ell = 4$				5			10		13
$\ell = 7$									13

(a) End sets

$\ell = 1$	2	1	2	2	1	2	1	2
$\ell = 2$			3	2		3		3
$\ell = 4$						5		3

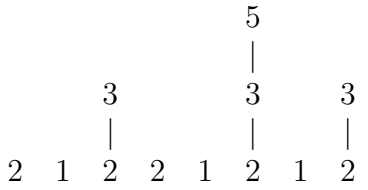
(b) *Gap* lists

Figure 2: End sets and corresponding *Gap* lists

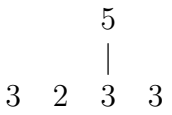
In Fig. 3(a), we show how queue Q will look like for $\ell = 1$. *First-appear*(1) keeps pointers at the bottom of the queue. *First-appear*(2) keeps pointers at the elements stacked a level above them. *First-appear*(4) keeps pointers at the elements stacked two levels above them. In Fig. 3(b) and 3(c), we show how queue Q will look like for $\ell = 2$ and for $\ell = 4$, respectively.

We are now in a position to calculate *cover* values. □

Before proceeding with calculating the minimal right-seed array of y, we also prove the following auxiliary lemma.



(a) Queue Q for $\ell = 1$



(b) Queue Q for $\ell = 2$

(c) Queue Q for $\ell = 4$

Figure 3: Queue Q

Lemma 2.13 *Let an end set $E_i = \{a_1, a_2, \ldots, a_k\}$ of y with corresponding factor z, $|z| = \ell$. Then z is a right seed for some set $\{y[0\,..\,a_1], y[0\,..\,a_2], \ldots, y[0\,..\,a_s]\}$, where $0 \leq s \leq k$ and $s = 0$ gives us the empty set. There is no other prefix of y with its end position in E_i having z as a right seed.*

Proof Let a_t the first element of E_i such that z is not a right seed of $y[0\,..\,a_t]$. If there exists no such element, then $s = k$ and z is a right seed for $\{y[0\,..\,a_1], y[0\,..\,a_2], .. y[0\,..\,a_k]\}$. If there exists no element $a_q \in E_i$ greater than a_t, such that z is a right seed of $y[0\,..\,a_q]$, then $s = t - 1$, and the required set S is:

$$S = \begin{cases} \emptyset, & s = 0 \\ \{y[0\,..\,a_1], y[0\,..\,a_2], .. y[0\,..\,a_{t-1}]\}, & \text{otherwise} \end{cases} \tag{6}$$

Suppose that there exists an element $a_q \in E_i$ greater than a_t such that z is a right seed of $y[0\,..\,a_q]$, i.e. it covers a superstring u, where $u = vy[0\,..\,a_q]$ and v is a possibly empty string. Therefore, there exists an occurrence of z in u ending in some $p \in \{y[a_t - \ell], \ldots, y[a_t - 1]\}$. Thus z is a cover of $vy[0\,..\,p]$. But there also exists an occurrence of z in y ending in $y[a_t]$. This shows that z is a cover of $vy[0\,..\,a_t]$, and hence a right seed of $y[0\,..\,a_t]$, which gives a contradiction. □

We are now in a position to calculate the minimal right-seed array of y by

operating on the distinct end sets, while running over them in a level order traversal of the partition tree. The value *first* is passed from a parent set to the child set. If *first* value is in the set, we do not need to update the value. If *first* value is not in a new child set, it can be easily updated by searching for the smallest element which is greater or equal to *first* value in the new set; running over the elements of the set (Lemma 2.13), takes time $\mathcal{O}(\textit{size of the new set})$. If *first* value is not in the old child set, we need to find the value just after it (Lemma 2.13), by searching in the elements of the parent set after *first* value; this takes time $\mathcal{O}(k)$, where k is the number of elements of new sibling sets of the old set. Keeping *first* value increases the time requirements for the partitioning algorithm only by a constant factor.

The period of $y[0\,..\,i]$, $\mathsf{P}[i]$, gives also the minimal right seed that can occur only once (that is why, in the next lines, we exclude distinct sets of size one; if they have not been assigned a right seed yet, then $\mathsf{P}[i]$ gives the length of their minimal right seed).

Let δ denote the first element of each distinct end set E_i, with corresponding factor length range $(\ell_{\min_i}, \ell_{\max_i})$, then the following hold.

- If $\textit{first}_i \leq \textit{cover}_{i,\ell_{\min_i}}$, by Lemma 2.2, the length of the minimal right seed for $y[0\,..\,\textit{first}_i]$ is

$$max\{\ell_{\min_i}, \mathsf{P}[\textit{first}_i] - (\textit{first}_i - \delta)\} \tag{7}$$

  If there are no such lengths in the factor length range of E_i, we stop operations in that set (as a consequence of Lemma 2.13).

- If $\textit{first}_i > \textit{cover}_{i,\ell_{\min_i}}$, we move to the smallest factor length, denoted by γ, such that $\textit{first}_i \leq \textit{cover}_{i,\gamma}$. This is easily found using the corresponding arrays CV_i and FL_i. By Lemma 2.2, the length of the minimal right seed for $y[0\,..\,\textit{first}_i]$ is

$$max\{\gamma, \mathsf{P}[\textit{first}_i] - (\textit{first}_i - \delta)\} \tag{8}$$

  If there are no such lengths in the factor length range of E_i, we stop operations in that set (as a consequence of Lemma 2.13).

If the minimal right seed of $y[0\,..\,\textit{first}_i]$ is found, we assign the smallest element of E_i which is greater than $\textit{first}_i$, as a new value for $\textit{first}_i$, and continue searching from E_i with corresponding factor length the last length assigned as a minimal right seed length (as a consequence of Lemma 2.13).

Reporting minimal right seeds takes time $\Theta(n)$, i.e. one report for each position, as constant time is needed for each report. Failed attempts are made:

- at most one per report (when after the report, the next element of the set does not give a minimal right seed)
- at most two per distinct set (one at the start of searching in the set, and one on a failure finding a suitable class for a future minimal right seed)

Therefore failed attempts are of $\mathcal{O}(n)$, and as constant time is needed for each report, the overall time needed for failed attempts is $\mathcal{O}(n)$. Also going over the *cover* changes takes time proportional to reporting *cover* values, which is $\mathcal{O}(n \log n)$.

Theorem 2.14 *Computing the minimal right-seed array of a given string y of length n can be done in $\mathcal{O}(n \log n)$ time.*

Proof By Proposition 2.11, executing the partitioning algorithm takes time $\mathcal{O}(n \log n)$. Maintaining the *Gap* list and the value *cover* increases the time requirements for the partitioning algorithm by a constant factor. Maintaining the array CV and the array FL is of no extra cost to maintaining the value *cover*. Maintaining the value *first* also increases the time requirements for the partitioning algorithm by a constant factor. The KMP algorithm, used for computing the period array P, runs in linear time [25]. Reporting the minimal right seeds requires $\Theta(n)$ time, i.e. one report for each position. Failed attempts are at most $\mathcal{O}(n)$. Going over the *cover* changes, while searching for minimal right seeds, takes time proportional to reporting *cover* values, which is $\mathcal{O}(n \log n)$. Hence, in overall, the described algorithm runs in $\mathcal{O}(n \log n)$ time.

Example An overview of the algorithm for computing the minimal right-seed array of string $y = \texttt{abaababaabaabab}$ is shown in Fig. 4. □

It is easy to see that Problem 6 is solved by solving Problem 4 (observe the similarity of Lemma 2.6 and Lemma 2.7), and so we obtain the following result.

Theorem 2.15 *Computing the maximal right-seed array of a given string y of length n can be done in linear time.*

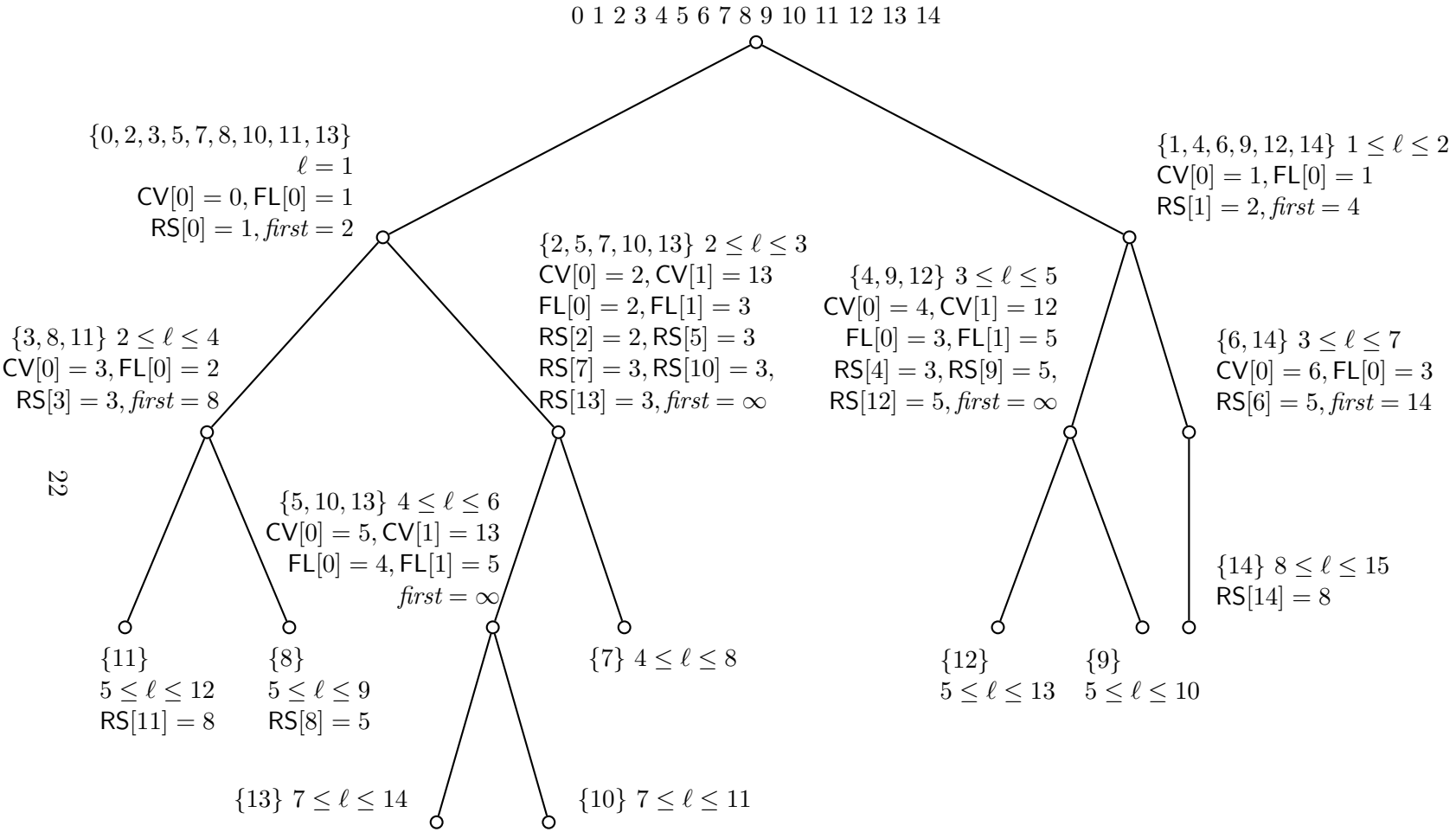


Figure 4: Computing the minimal right-seed array of string $y = \texttt{abaababaabaabab}$ (see also Fig. 5)

3. Subtree repeats

Tree pattern matching has been intensively studied over the past decades because of its various applications, among others, in mechanical theorem proving, term-rewriting, instruction selection, and non-procedural programming languages [21, 14, 26]. In addition, tree pattern matching has direct applications in computational biology, e.g. glycan classification [27], exact and approximate pattern matching and discovery in RNA secondary structure [31].

In many applications, it is essential to extract the repeated patterns in a tree within a matchematical structure [13, 16, 24]. In particular, the *common subtrees* problem consists of finding all of the subtrees having the same structure and the same labels on the corresponding nodes of two ordered labelled unranked trees [19]. This problem of equivalence, which is strictly related to the *common subexpression* problem [13, 16], arises, for instance, in the code optimisation phase of compiler design, or in saving storage for symbolic computations [1, 13, 16].

In this report, we consider a slightly different problem, and provide a completely different solution to what has be done so far. We focus on finding all the repeating subtrees – the subtrees occurring more than once – in a tree structure. Notice that this problem can be solved by the algorithm presented in [19]. However, that solution requires the construction of a suffix tree, which is expensive in practical terms. This probem is analogous to the well-known problem of finding all the repetitions in a given word [10]. Apart from its pleasing theoretical features, finding all the repeating subtrees, can be directly and effectively applied as an alternative solution, on the *maximum agreement subtree* (MAST) problem for trees representing the evolutionary history of a set of species [20]. That is, given a set of evolutionary leaf-labelled trees on the same set of taxa, the MAST problem consists of finding a subtree homeomorphically included in all input trees, and with the largest number of taxa.

The proposed algorithm is divided into two phases: the preprocessing phase and the phase where all the repeating subtrees are computed. The preprocessing phase tranforms the given tree to a string representing its postfix notation, and then computes arrays that store the height of each node of the tree, the parent of each node, and an indicator showing whether a node is the first child of its parent. The second phase, for computing all the repeating subtrees, is done in a bottom-up manner, using a partitioning procedure.

The importance of the proposed algorithm is underlined by the fact that it can be applied in both unlabelled and labelled ordered ranked trees. Its linear runtime, as well as the use of linear auxiliary space, are important parts of its quality.

The rest of the section is structured as follows. Subsection 3.1 presents all the notations and definitions used throughout this section, and formally defines the probelms solved. In Subsection 3.2, we give some useful properties of trees in postfix notation. Subsection 3.3 describes the algorithm for finding all the repeating subtrees in unlabelled and labelled ordered ranked trees. Finally, in Section 3.4 we present some experimental results.

3.1. Preliminaries

In this section we define some more notations and definitions that we will use throughout the report.

We denote the set of natural numbers by $\mathbb{N}$. An *alphabet* Σ is a finite, non-empty set of symbols. A string is a succession of zero or more symbols from an alphabet Σ. The string with zero symbols is denoted by ε. The set of all strings i over the alphabet Σ is denoted by Σ^*. A string x of length m is represented by $x_1x_2 \dots x_m$, where $x_i \in \Sigma$ for $1 \leq i \leq m$. The length of a string x is denoted by $|x|$. A string w is a *factor* (or substring) of x if $x = uwv$ for $u, v \in \Sigma^*$, and is represented as $w = x_i \dots x_j, 1 \leq i \leq j \leq |x|$.

A *ranked alphabet* is a couple $\mathcal{A} = (\Sigma, \varphi)$, where Σ is a finite, non-empty set of symbols and φ is the mapping $\varphi : \Sigma \mapsto \mathbb{N}$.

An *ordered directed graph* G is a pair (N, R), where N is a set of nodes and R is a set of linearly ordered lists of edges such that each element of R is of the form $((f, g_1), (f, g_2), \dots, (f, g_n))$, where $f, g_1, g_2, \dots, g_n \in N$, $n \geq 0$. This element would indicate that, for node f, there are n edges leaving f, the first entering node g_1, the second entering node g_2, and so on.

A sequence of nodes $(f_0, f_1, \dots, f_n)$, $n \geq 1$, is a *path* of length n from node f_0 to node f_n if there is an edge which leaves node f_{i-1} and enters node f_i for $1 \leq i \leq n$. A *cycle* is a path $(f_0, f_1, \dots, f_n)$, where $f_0 = f_n$. An ordered *dag* (dag stands for Directed Acyclic Graph) is an ordered directed graph that has no cycle. *Labelling* of an ordered graph $G = (A, R)$ is a mapping of A into a set of labels drawn from a finite alphabet.

Given a node f, its *out-degree* is the number of distinct pairs $(f, g) \in R$, where $g \in A$. By analogy, *in-degree* of node f is the number of distinct pairs $(g, f) \in R$, where $g \in A$.

An ordered *tree* t is an ordered dag $t = (N, R)$ with a special node $r \in A$ called the *root* such that
(1) r has in-degree 0,
(2) all other nodes of t have in-degree 1,
(3) there is just one path from the root r to every $f \in N$, where $f \neq r$.

A tree t is *unordered* if no ordering is given on the edge lists of its nodes.

A tree t is *labelled* if every node $f \in N$ is labelled by a symbol $a \in \Sigma$, Σ a finite alphabet.

A tree t is ranked if for every node $f \in N$ its out-degree of is given.

The number of nodes of a tree t is denoted by $|t|$.

The height of a tree t, denoted by *Height(t)*, is defined as the maximal length of a path from the root of t to a leaf of t.

Let a list of edges of a node f of a tree t be $((f, g_1), (f, g_2), \ldots, (f, g_n))$, where $f, g_1, g_2, \ldots, g_n \in N$, $n \geq 0$. Then $\{g_1, g_2, \ldots, g_n\}$ are siblings between them, they are children of f and f is their parent.

The *postfix notation post*(t) of a labeled, ordered, ranked tree t is obtained by applying the following *Step* recursively, beginning at the root of t:
Step: Let this application of *Step* be node v. If v is a leaf, list v and halt. If v is an internal node having descendants $v_1, v_2, \ldots, v_{\varphi(v)}$, apply *Step* to $v_1, v_2, \ldots, v_{\varphi(v)}$ in that order and then list v.

Example An illustration of the tree t, having postfix notation $post(t) = a_0\, a_0\, a_0\;\; a_1\, a_2\, a_0\, a_1\, a_3\, a_0\, a_1\, a_1\, a_1\, a_0\, a_0\, a_1\, a_2\, a_2\, a_0\, a_0\, a_2\, a_0\, a_0\, a_1\, a_2\, a_4$ is presented in Fig. 5.

A subtree p *matches* an object tree T at node n if p is equal to the subtree of t rooted at n.

A *subtree repeat* in a subject tree T, represented by its postfix notation $x = post(T)$, is a tuple:

$$M_{x,u} = (p; i_1, i_2, \ldots, i_r), r \geq 2$$

where $i_1 < i_2 < \ldots < i_r$ and

$$u = x_{i_1} \ldots x_{i_1+|p|-1} = x_{i_2} \ldots x_{i_2+|p|-1} = \ldots = x_{i_r} \ldots x_{i_r+|p|-1}$$

and $u = post(p)$. If the tuple includes all the occurrences of u in x, then $M_{x,u}$ is said to be complete and is written $M^*_{x,u}$.

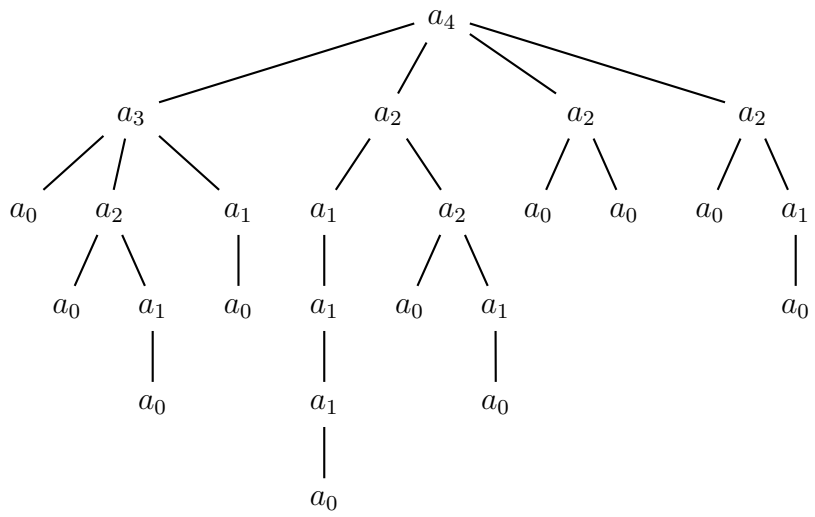


Figure 5: Tree t from Example 3.1 having postfix notation $post(t) = a_0\, a_0\, a_0\, a_1\, a_2\, a_0\, a_1\, a_3\, a_0\, a_1\, a_1\, a_1\, a_0\, a_0\, a_1\, a_2\, a_2\, a_0\, a_0\, a_2\, a_0\, a_0\, a_1\, a_2\, a_4$

We consider the following problems:

Problem 7 (All subtree repeats of an unlabelled ordered ranked tree) *Find the complete subtree repeats of an unlabelled ordered ranked tree t consisted of n nodes.*

Problem 8 (All subtree repeats of a labelled ordered ranked tree) *Find the complete subtree repeats of a labelled ordered ranked tree t consisted of n nodes.*

3.2. Properties of trees in postfix notation

In this subsection we present and prove some basic properties of trees in their postfix notation.

Lemma 3.1 *Given a tree t and its postfix notation $post(t)$, the postfix notations of all subtrees of t are substrings of $post(t)$.*

Proof By induction on the height of the subtree:

1. If a subtree t' consists of only one node a ($|t'| = 1$ and $h(t') = 0$), where $\varphi(a) = 0$, then $post(t') = a$ and the claim holds for that subtree.

2. Assume the claim holds for subtrees $t_1, t_2, \ldots, t_p$, where $p \geq 1$ and $h(t_i) \leq m$, $1 \leq i \leq p$, $m \geq 0$. We must prove that the claim holds also for each subtree $t' = t_1 t_2 \ldots t_p a$, where $\varphi(a) = p$, $h(t') = m + 1$:
As $post(t') = post(t_1)\ post(t_2) \ldots post(t_p)\ a$, the claim holds for subtree t'. □

Lemma 3.2 *Given a ranked tree t of size n and its postfix notation $post(t) = x_1 x_2 \ldots x_n$, the sum of arities of all nodes of t is $\sum_{i=1}^{n} \varphi(x_i) = n - 1$.*

Proof The sum of arities of all nodes is the sum of the number of children of all nodes which is the sum of the edges hanging from each node. Each parent does not share children with any other node therefore each edge is counted only once. It is easy to see that there are $n - 1$ edges in the tree (each node is hanging from an edge except the root of the tree). □

However, not every substring of a tree in postfix notation is a subtree in postfix notation. This is obvious due to the fact that for a given tree with n nodes in postfix notation, there can be $\mathcal{O}(n^2)$ distinct substrings but there are just n subtrees, each node of the tree is the root of one subtree. Just those substrings which themselves are trees in postfix notation are subtrees in postfix notation. This property is formalised by the following definition and lemma.

Definition Let $w = a_1 a_2 \ldots a_m$, $m \geq 1$, be a string over a ranked alphabet $\mathcal{A}$. Then, the *arity checksum* $ac(w) = \varphi(a_1) + \varphi(a_2) + \ldots + \varphi(a_m) - m + 1 = \sum_{i=1}^{m} \varphi(a_i) - m + 1$.

Lemma 3.3 *Let $post(t)$ and w be a tree t in postfix notation and a substring of $post(t)$, respectively. Then, w is the postfix notation of a subtree of t, if and only if $ac(w) = 0$, and $ac(w_1) \geq 1$ for each w_1, where $w = xw_1$, $x \neq \varepsilon$.*

Proof For any two subtrees t_1 and t_2 it holds that $post(t_1)$ and $post(t_2)$ are either two different strings or one is a substring of the other. The former case occurs if the subtrees t_1 and t_2 are two different trees with no shared part and the latter case occurs if one tree is a subtree of the other tree. No partial overlapping of subtrees is possible in ranked ordered trees. Moreover, for any two adjacent subtrees it holds that their postfix notations are two adjancent substrings.

- *If:* By induction on the height of subtree t, where $w = post(t)$:
  1. Assume that $h(t) = 1$, which means we consider the case $w = a$, where $\varphi(a) = 0$. Then, $ac(w) = 0$ and the claim holds for $h(t) = 1$.
  2. Assume that the claim holds for subtrees $t_1, t_2, \ldots, t_p$ where $p \geq 1$, $h(t_1) \leq m$, $h(t_2) \leq m$, ..., $h(t_p) \leq m$, $ac(post(t_1)) = 0$, $ac(post(t_2)) = 0$, ..., $ac(post(t_p)) = 0$. We are to prove that it holds also for a subtree of height $m+1$. Assume $w = post(t_1)post(t_2) \ldots post(t_p)a$, where $\varphi(a) = p$. Then $ac(w) = ac(post(t_1)) + ac(post(t_2)) + \ldots + ac(post(t_p)) + p - (p+1) + 1 = 0$ and $ac(w_1) \geq 1$ for each w_1 , where $w = xw_1$, $x \neq \varepsilon$.
     Thus, the claim holds for the case $h(t) = m + 1$.

- *Only if*: Assume $ac(w) = 0$, and $w = a_1 a_2 \ldots a_k$, where $k \geq 1$, $Arity(a_k) = p$. Since $ac(w_1) \geq 1$ for each w_1, where $w = xw_1$, $x \neq \varepsilon$, none of the substrings w_1 can be a subtree in postfix notation. This means that the only possibility for $ac(w) = 0$ is that w is of the form $w = post(t_1)\ post(t_2) \ldots post(t_p)\ a$, where $p \geq 0$, and $t_1, t_2 \ldots t_p$ are adjacent subtrees. In such case, $ac(w) = 0 + p - (p + 1) + 1 = 0$. No other possibility of the form of w for $ac(w) = 0$ is possible. Thus, the claim holds. □

For the validity of our algorithm we also prove the following condition for a substring in postfix notation to be a subtree.

Lemma 3.4 *Let post(t) and w be a tree t in postfix notation and a substring of post(t), respectively. Then $w = w_1 \ldots w_{|w|}$, $w_i \in \mathcal{A}$, is the postfix notation of a subtree of t iff:*

- *w is composed by one leaf, or*
- *$ac(w) = 0$, w_1 corresponds to a node which is both a leaf and a first child, $|w| \geq 2$*

Proof First part is obvious.Now for the second part we need to prove:
Direct: Let w be the postfix notation of a subtree of t. Then w_1 corresponds to both a leaf and a first child as (due to postorder traversal of the tree). Also $ac(w) = 0$ as of Lemma 3.2.
Reverse:Let w be a substring of $post(t)$, such that $ac(w) = 0$, w_1 is both a

leaf and a first child, and $|w| \geq 2$. Substrings of $post(t)$ starting from w_1 might end:

- on a node z whose subtree has w_1 as a leftmost leaf
  Then that substring is the postfix notation of the subtree rooted at z (we consider postorder traversal of the tree) and by Lemma 3.2 the arity checksum of that subtree is 0.

- on a node z whose subtree has a leftmost leaf found before w_1 in $post(t)$
  Then that substring is the postfix notation of the subtree rooted at z, which contains the largest subtree, say s, having w_1 as a leftmost leaf. However the siblings of s are missing and it is obvious to see that the sum of arities is greater that the size of the substring minus 1 (more children than the number of nodes minus 1) and so the arity checksum of that substring is greater than 0.

- on a node z whose subtree has a leftmost leaf found after w_1 in $post(t)$
  Then that substring contains a subtree, say s, having w_1 as a leftmost leaf and a collection of leaves and subtrees found later in the postorder traversal of the tree. It is obvious to see that the sum of arities is less that the size of the substring minus 1 (less children than the number of nodes minus 1) and so the arity checksum of that substring is less than 0. □

3.3. Algorithms

In this section, we present algorithms solving Problems 7 and 8. The algorithms are divided in two phases: the preprocessing phase and the searching phase.

We divide the rest of this section in three subsections: first, we present the preprocessing part of the algorithm; then, we show a method for solving Problem 7 which we then extend to solve Problem 8.

3.3.1. Preprocessing

Given a tree t, its postfix notation is first computed using a simple postorder traversal of the tree.

While not necessary in general, a new identifier can be encoded for each node of the subject tree, based on its label and rank. These identifiers, along with the arity of the respective nodes, form the ranked alphabet. In this way,

the case that the tree consists of nodes having the same label but different arity, can be easily handled.

Let $post(t) = x_1 x_2 \dots x_n$ be the postfix notation of tree t. The preprocessing phase completes by computing 3 auxiliary arrays, which will be used during the searching phase:

1. The height of each subtree of t, having node x_i as its root, is stored in array H, for $1 \leq i \leq n$.
2. An array P of n elements, with the *i-th* element having the value p if x_p is the parent of x_i.
3. A binary array FC consisting of 1s and 0s, where the *i-th* element is set to 1 in case x_i is the first (leftmost) child of its parent node $x_{P[i]}$.

3.3.2. Finding subtree repeats

ALGORITHM SUBTREE-REPEATS($post(t) = x_1 x_2 \dots x_n$ over ranked alphabet $\mathcal{A} = (\Sigma, \varphi)$)

```
1: sc ← 1;
2: for i ← 1 to n do
3:    if φ(x_i) = 0 then
4:       S ← S ⋃ {i};
5:       T[i] ← sc;
6:       TL[i] ← 1;
7:    else
8:       T[i] ← 0;
9:       TL[i] ← 0;
10: for i ← 1 to H[n] do
11:    while LA[i] is not empty do
12:       PARTITION(DEQUEUE(LA[i]))
13: return Sets of starting positions of substrings of post(t) and their
    lengths, representing subtrees from t;
```

We are now in a position to present Algorithm SUBTREE-REPEATS, solving Problem 7. The computation is based on a bottom-up traversal of the input tree t, described by its postfix notation $post(t) = x_1 x_2 \dots x_n$. The algorithm, at each step (level) i, locates and outputs all repeating subtrees of height i. We also introduce an auxiliary array called *level array* (LA), which keeps track of queues of triplets. These triplets describe factors of $post(t)$,

and are of the form (S, ℓ, ac), where S is a set containing the starting positions of the occurences of the factor, ℓ is the factor's length, and ac is its arity checksum, which, in case is 0, indicates that the factor corresponds to some subtree of t.

The algorithm starts by constructing a triplet $(S, 1, 0)$, representing all leaves, i.e. subtrees of height 0 (lines 3-6), with its set S containing all positions of the unary symbol in $post(t)$. The triplet is then passed to the function Assign-Level, which splits the elements of S in several subsets, according to the height of the subtree specified by the parent (stored in $H[P[root]]$) of each element in S (line 4). Elements which do not correspond to subtrees being leftmost children (first children) of their parent nodes are discarded. The resulting subsets are then wrapped into triplets and appended in the appropriates queues of LA according to $H[P[root]]$. Note, that function Assign-Level takes as input only triplets describing factors that correspond to subtrees.

At each step i of the algorithm, the triplets in $LA[i]$ are passed to the function Partition, which partitions a triplet according to the next factor, starting at position r (line 2), that is to be concatenated with the factor described by the triplet. The next factor either represents a subtree of t in postfix notation, in case its first symbol marks the beginning of a subtree (lines 3-5), or a single symbol (lines 6-8). Triplets are recursively partitioned (line 18) until they describe factors representing subtrees in postfix notation (lines 10-16). When a triplet finally describes a factor representing a subtree, starting positions of those factors are assigned an index (stored in array T), and their lengths are stored in an array TL (lines 14-15). This is to indicate that the factor starting at position i i having length $TL[i]$ was found to correspond to a subtree. Those triplets are then processed by Assign-Level, which partitions them in levels and stores them in the appropriate queues of LA. As stated before, only the elements i of the set S of a triplet $E = (S, \ell, 0)$, such that the subtree corresponding to the factor $x_i \dots x_{i+\ell}$ is the first children of the subtree specified by its parent node, are considered (Lemma 3.4). The algorithm terminates when the level array LA is empty.

Theorem 3.5 *The algorithm* Subtree-Repeats *computes all subtree repeats of a given tree t in $\Theta(n)$ time, where $|t| = n$.*

Proof The preprocessing phase, i.e. the computation of $post(t)$, arrays P,H and FC, is done in $\Theta(n)$ time. During the expansion of the subtrees performed in the function Partition, the algorithm does not read a symbol

ALGORITHM Assign-Level($E = (S, \ell, ac)$)

1: **for** $i \in S$ **do**
2: $root = i + TL[i] - 1$;
3: **if** $FC[root] = 1$ **then**
4: $S_{H[P[root]]} \leftarrow S_{H[P[root]]} \cup \{i\}$;
5: $L \leftarrow L \cup H[P[root]]$;
6: **for** $i \in L$ **do**
7: Enqueue($LA[i], (S_i, \ell, 0)$);
8: **return** Partitioning of E in levels;

ALGORITHM Partition($E = (S, \ell, ac)$, $post(t) = x_1 x_2 \ldots x_n$)

1: **for** $i \in S$ **do**
2: $r = i + \ell$;
3: **if** $T[r] \neq 0$ **then**
4: $E_{T[r]} \leftarrow (S_{T[r]} \cup \{i\}, \ell + TL[r], ac - 1)$;
5: $L \leftarrow L \cup \{E_{T[r]}\}$;
6: **else**
7: $E_{x_r} \leftarrow (S_{x_r} \cup \{i\}, \ell + 1, ac - 1 + \varphi(x_r))$;
8: $L \leftarrow L \cup \{E_{x_r}\}$;
9: **for** $E_i = (S_i, \ell_i, ac_i) \in L$ **do**
10: **if** $ac_i = 0$ **then**
11: Output(S_i, ℓ_i);
12: $sc = sc + 1$;
13: **for** $j \in S_i$ **do**
14: $T[j] \leftarrow sc$;
15: $TL[j] \leftarrow \ell_i$;
16: Assign-Level(E_i);
17: **else**
18: Partition(E_i);
19: **return** Partitioning of E in classes corresponding to next element to be considered;

more than once, but rather reads the previously expanded subtrees. Merging the subtrees is done in $n-1$ operations (number of children of the tree). □

3.3.3. Solving the problem for labelled ordered ranked trees

The algorithm described in the previous section can be adapted to solve Problem 8 using a slight modification, which we present in this section.

During initialization we should pay attention to form different sets of leaves according to their label. Additionally, partitioning should be done according to the label, rank and subtree of the next element that is to be considered. In that way we will require a $(\Sigma+1)n \times 1$ array of sets to be formed in order to obtain the partitioning. However we can do better than this by separating the partitioning in the following cases:

- When we partition with respect to subtree starting from next element to be considered we use an auxiliary $n \times 1$ array of sets.
- When we partition with respect to next element to be considered we first use an auxiliary $n \times 1$ array of sets to partition with respect to the arity of the element and then an auxiliary $\Sigma \times 1$ array of queues to partition with respect to the label of the element. We can restrict to the different labels that appear in the tree, say k (clearly $k \leq n$), thus using an auxiliary $k \times 1$ array of queues at the second step of the partitioning, thus using $\Theta(n)$ space during the execution of the algorithm.

The procedures that differ from the previous algorithm are given in the Appendix.

3.4. Experiments

We have conducted an experiment to verify the linear runtime of the proposed algorithm computing subtree repeats in unlabelled ordered ranked trees, in practice. For the experiment we have used randomly generated trees with their size ranging from 100 to 10000 nodes with a step of 100 nodes and a bounded alphabet with the arity of the symbols ranging between 0 and 5. Figure 6 shows the number of operations carried out by our implementation of the algorithm against the number of nodes of the tree instances. The resulting graph clearly indicates the linear relationship between the runtime and the number of nodes of tree instances.

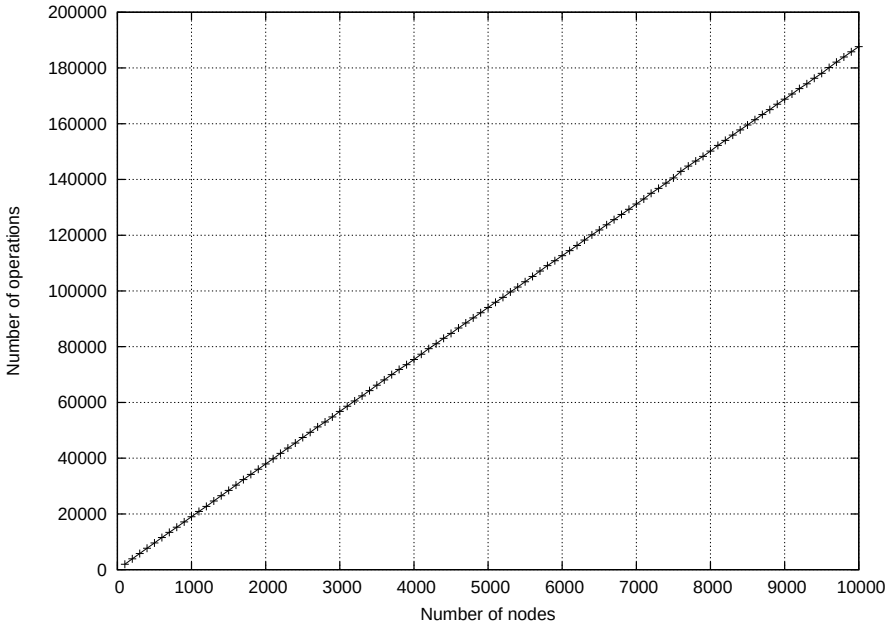


Figure 6: Number of operations performed by Algorithm SUBTREE-REPEATS, according to the size of the input tree

4. Conclusion and Further Work

In this report, we have provided a linear-time algorithm for computing all left seeds and all right seeds of a string, a linear-time algorithm for computing the mimimum left-seed array of a string, a linear-time solution for computing the maximum left-seed array of a string, an $\mathcal{O}(n \log n)$-time algorithm for the computing the minimum right-seed array of a string, and a linear-time solution for computing the maximum right-seed array of a string.

Recently, Crochemore, Iliopoulos, Pissis and Tischler in [12], provided linear-time algorithms for checking the validity of minimal and maximal cover arrays and inferring strings from valid minimal and maximal cover arrays. Their result completed the series of algorithmic characterizations of data structures that store fundamental features of strings. They concern Border arrays [15, 17] and Prefix arrays [9] that store periods of all the prefixes of a string, as well as the element of Suffix arrays [4, 18] that memorizes the list of positions of lexicographically sorted suffixes of the string. The algorithms may be regarded as reverse engineering processes and, beyond their obvious theoretical interest, they are useful to test the validity of some constructions.

Hence, further work can be done on the following relevant problems as well.

Problem 9 (Validity problem) *Let A be an integer array of length n. Decide if A is the minimal left (resp. right) seed array of some string.*

Problem 10 (Construction Problem) *Let A be an integer array of length n. When A is a valid minimal left-seed (resp. right-seed) array, exhibit a string over an unbounded alphabet whose minimal left-seed (resp. right-seed) array is A.*

Furthermore we have formally defined the problem of computing all the subtree repeats in ordered ranked trees and presented a new solution based on the bottom-up technique. The proposed algorithm is divided into two phases: the preprocessing phase and the phase where all the repeating subtrees are computed. The preprocessing phase tranforms the given tree to a string representing its postfix notation, and then computes and stores some useful properties of the tree. The second phase, for computing all the repeating subtrees, is done in a bottom-up manner, using a partitioning procedure.

It is important to mention that the proposed algorithm for computing all the subtree repeats in ordered ranked trees runs in linear time and space. This problem is analogous to the well-known problem of finding all the repetitions in a given word. The significance of the proposed algorithm is underlined by the fact that it can be applied in both unlabelled and labelled ordered ranked trees. The presented experimental results demonstrate the efficiency of the proposed algorithm in practice.

The goal of our future research is to modify and apply the proposed algorithm, as an alternative solution, on the maximum agreement subtree problem for trees representing the evolutionary history of a set of species.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2 edition, Aug. 2006.

[2] A. Apostolico and D. Breslauer. Of periods, quasiperiods, repetitions and covers. In *Structures in Logic and Computer Science*, pages 236–248, 1997.

[3] A. Apostolico and A. Ehrenfeucht. Efficient detection of quasiperiodicities in strings. *Theor. Comput. Sci.*, 119(2):247–265, 1993.

[4] H. Bannai, S. Inenaga, A. Shinohara, and M. Takeda. Inferring strings from graphs and arrays. In B. Rovan and P. Vojtás, editors, *Mathematical Foundations of Computer Science*, volume 2747 of *Lecture Notes in Computer Science*, pages 208–217. Springer, 2003.

[5] D. Breslauer. An on-line string superprimitivity test. *Inf. Process. Lett.*, 44(6):345–347, 1992.

[6] M. Christou, M. Crochemore, T. Flouri, C. S. Iliopoulos, B. Melichar, and S. P. Pissis. Computing all the subtree repeats in ordered ranked trees. In *Proceedings of the 18th International Symposium on String Processing and Information Retrieval*, Lecture Notes in Computer Science. Springer, 2011. (accepted).

[7] M. Christou, M. Crochemore, O. Guth, C. S. Iliopoulos, and S. P. Pissis. On the right-seed array of a string. In *Proceedings of the 17th Annual International Computing and Combinatorics Conference (COCOON'11)*, Lecture Notes in Computer Science. Springer, 2011. (accepted).

[8] M. Christou, M. Crochemore, C. S. Iliopoulos, M. Kubica, S. P. Pissis, J. Radoszewski, W. Rytter, B. Szreder, and T. Walen. Efficient seed computation revisited. In *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM 2011)*, Lecture Notes in Computer Science. Springer, 2011. (accepted).

[9] J. Clement, M. Crochemore, and G. Rindone. Reverse engineering prefix tables. In S. Albers and J.-Y. Marion, editors, *26th International Symposium on Theoretical Aspects of Computer Science (STACS 2009)*, pages 289–300, Dagstuhl, Germany,

2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany. http://drops.dagstuhl.de/opus/volltexte/2009/1825.

[10] M. Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.*, 12(5):244–250, 1981.

[11] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, New York, NY, USA, 2007.

[12] M. Crochemore, C. Iliopoulos, S. Pissis, and G. Tischler. Cover array string reconstruction. In A. Amir and L. Parida, editors, *Combinatorial Pattern Matching*, volume 6129 of *Lecture Notes in Computer Science*, pages 251–259. Springer Berlin / Heidelberg, 2010.

[13] P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *J. ACM*, 27:758–771, October 1980.

[14] M. Dubiner, Z. Galil, and E. Magen. Faster tree pattern matching. *J. ACM*, 41:205–213, March 1994.

[15] J.-P. Duval, T. Lecroq, and A. Lefebvre. Border array on bounded alphabet. *Journal of Automata, Languages and Combinatorics*, 10(1):51–60, 2005.

[16] P. Flajolet, P. Sipala, and J.-M. Steyaert. Analytic variations on the common subexpression problem. In *Proceedings of the seventeenth international colloquium on Automata, languages and programming*, pages 220–234, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[17] F. Franek, S. Gao, W. Lu, P. J. Ryan, W. F. Smyth, Y. Sun, and L. Yang. Verifying a Border array in linear time. *J. Combinatorial Math. and Combinatorial Computing*, 42:223–236, 2002.

[18] F. Franek and W. F. Smyth. Reconstructing a Suffix Array. *J. Foundations of Computer Sci.*, 17(6):1281–1295, 2006.

[19] R. Grossi. On finding commong subtrees. *Theor. Comput. Sci.*, 108(2):345–356, 1993.

[20] S. Guillemot and F. Nicolas. Solving the maximum agreement subtree and the maximum compatible tree problems on many bounded degree trees. *CoRR*, abs/0802.0024, 2008.

[21] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *J. ACM*, 29:68–95, January 1982.

[22] C. Iliopoulos and K. Park. A work-time optimal algorithm for computing all string covers. *Theoretical Computer Science*, 164(1-2):299–310, 1996.

[23] C. S. Iliopoulos, D. W. G. Moore, and K. Park. Covering a string. *Algorithmica*, 16:289–297, Sept. 1996.

[24] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, STOC '72, pages 125–136, New York, NY, USA, 1972. ACM.

[25] D. E. Knuth, J. H. M. Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[26] S. R. Kosaraju. Efficient tree pattern matching. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 178–183, Washington, DC, USA, 1989. IEEE Computer Society.

[27] T. Kuboyama. *Matching and Learning in Trees*. PhD thesis, University of Tokyo, 2007.

[28] Y. Li and W. F. Smyth. Computing the cover array in linear time. *Algorithmica*, 32(1):95–106, 2002.

[29] M. Lothaire, editor. *Algebraic Combinatorics on Words*. Cambridge University Press, 2001.

[30] M. Lothaire, editor. *Appplied Combinatorics on Words*. Cambridge University Press, 2005.

[31] G. Mauri and G. Pavesi. Algorithms for pattern matching and discovery in rna secondary structure. *Theor. Comput. Sci.*, 335:29–51, May 2005.

[32] D. Moore and W. Smyth. An optimal algorithm to compute all the covers of a string. *Information Processing Letters*, 50(5):239–246, 1994.

Appendix A. Additional details for the Subtree repeats problem

In this section more details on the solution of the Subtree repeats problem are given. In particular we show the procedures of the preprocessing phase, an example for the unlabelled case and the procedures for solving the labelled version of the problem.

Appendix A.1. Preprocessing phase

In this subsection, the procedures of the preprocessing phase are presented.

ALGORITHM COMPUTE-SUBTREE-HEIGHT(φ, n)

```
 1: S ← ∅;
 2: for i ← 1 to n do
 3:    if φ[i] = 0 then
 4:       Push(S, 0);
 5:       H[i] ← 0;
 6:    else
 7:       r ← 0;
 8:       for j ← 1 to φ[i] do
 9:          r ← max(r, Pop(S));
10:       H[i] ← r + 1;
11:       Push(S, r + 1);
12: return Array H[1 ... n];
```

ALGORITHM COMPUTE-NODE-PARENTS(φ, n)

```
 1: S ← ∅;
 2: for i ← 1 to n do
 3:    for j ← 1 to φ[i] do
 4:       r ← Pop(S);
 5:       P[r] ← i;
 6:    Push(S, i);
 7: return Array P[1 ... n − 1];
```

ALGORITHM FIRST-CHILD(φ, n)

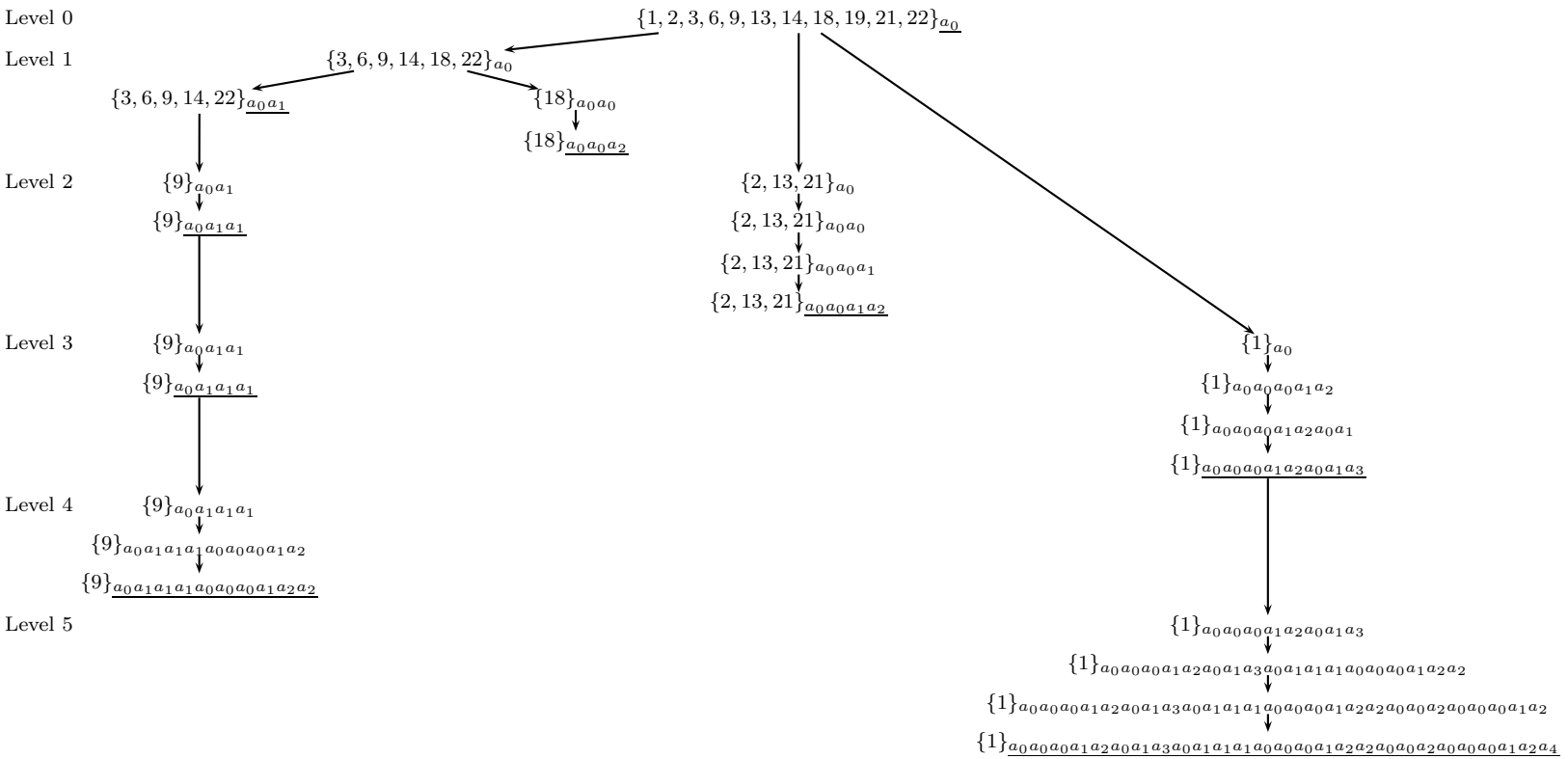
```
1: S ← ∅;
2: for i ← 1 to n do
3:    if φ[i] = 0 then
4:       Push(S, i);
5:    else
6:       for j ← 1 to φ[i] − 1 do
7:          r ← Pop(S);
8:          FC[r] ← 0;
9:       r ← Pop(S);
10:      FC[r] ← 1;
11:      Push(S, i);
12: FC[n] ← 1
13: return Array FC[1...n];
```

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
$post(t)$	a_0	a_0	a_0	a_1	a_2	a_0	a_1	a_3	a_0	a_1	a_1	a_1	a_0	a_0	a_1	a_2	a_2	a_0	a_0	a_2	a_0	a_0	a_1	a_2	a_4
P	8	5	4	5	8	7	8	25	10	11	12	17	16	15	16	17	25	20	20	25	24	23	24	25	-
H	0	0	0	1	2	0	1	3	0	1	2	3	0	0	1	2	4	0	0	1	0	0	1	2	5
FC	1	1	1	0	0	1	0	1	1	1	1	1	1	1	0	0	0	1	0	0	1	1	0	0	-

Table A.1: Preprocessing output

Appendix A.2. Example

In this subsection, we show how the proposed algorithm computes all the subtree repeats of the tree in Fig. 5.


Level 0
Level 1
Level 2
Level 3
Level 4
Level 5
$\{1, 2, 3, 6, 9, 13, 14, 18, 19, 21, 22\}_{a_0}$
$\{3, 6, 9, 14, 18, 22\}_{a_0}$
$\{3, 6, 9, 14, 22\}_{a_0a_1}$
$\{18\}_{a_0a_0}$
$\{18\}_{a_0a_0a_2}$
$\{9\}_{a_0a_1}$
$\{9\}_{a_0a_1a_1}$
$\{9\}_{a_0a_1a_1}$
$\{9\}_{a_0a_1a_1a_1}$
$\{9\}_{a_0a_1a_1a_1}$
$\{9\}_{a_0a_1a_1a_1a_0a_0a_0a_1a_2}$
$\{9\}_{a_0a_1a_1a_1a_0a_0a_0a_1a_2a_2}$
$\{2, 13, 21\}_{a_0}$
$\{2, 13, 21\}_{a_0a_0}$
$\{2, 13, 21\}_{a_0a_0a_1}$
$\{2, 13, 21\}_{a_0a_0a_1a_2}$
$\{1\}_{a_0}$
$\{1\}_{a_0a_0a_0a_1a_2}$
$\{1\}_{a_0a_0a_0a_1a_2a_0a_1}$
$\{1\}_{a_0a_0a_0a_1a_2a_0a_1a_3}$
$\{1\}_{a_0a_0a_0a_1a_2a_0a_1a_3}$
$\{1\}_{a_0a_0a_0a_1a_2a_0a_1a_3a_0a_1a_1a_1a_0a_0a_0a_1a_2a_2}$
$\{1\}_{a_0a_0a_0a_1a_2a_0a_1a_3a_0a_1a_1a_1a_0a_0a_0a_1a_2a_2a_0a_0a_2a_0a_0a_0a_1a_2}$
$\{1\}_{a_0a_0a_0a_1a_2a_0a_1a_3a_0a_1a_1a_1a_0a_0a_0a_1a_2a_2a_0a_0a_2a_0a_0a_0a_1a_2a_4}$

index	Sets
0	$\{1,2,3,6,9,13,14,18,19,21,22\}_1$
1	$\{3,6,9,14,18,22\}_1$
2	$\{9\}_2, \{2,13,21\}_1$
3	$\{9\}_3, \{1\}_1$
4	$\{9\}_4$
5	$\{1\}_8$

Table A.2: Level array

Appendix A.3. Procedures for labelled ordered ranked trees

In this subsection, the procedures for computing all the subtree repeats of a given labelled ordered rank tree are given.

index	Factors of subtrees
1	a_0
2	$a_0a_0a_2$
3	a_0a_1
4	$a_0a_0a_1a_2$
5	$a_0a_1a_1$
6	$a_0a_0a_0a_1a_2a_0a_1a_3$
7	$a_0a_1a_1a_1$
8	$a_0a_1a_1a_1a_0a_0a_0a_1a_2a_2$
9	$a_0a_0a_0a_1a_2a_0a_1a_3a_0a_1a_1a_1a_0a_0a_0a_1a_2a_2a_0a_0a_2a_0a_0a_0a_1a_2a_4$

Table A.3: Indexing of subtrees

ALGORITHM PARTITIONING-L(E)

1: **for** $i \in S$ **do**
2: $next = i + \ell_E$;
3: **if** $T[next] \neq 0$ **then**
4: $E_{T[next]} \leftarrow (S_{T[next]} \bigcup \{i\}, \ell_E + TL[next], ac_E - 1)$;
5: **else**
6: $E_{\varphi(post(t)[next])} \leftarrow (S_{\varphi(post(t)[next])} \bigcup \{i\}, \ell_E + 1, ac_E - 1 + \varphi[next])$;
7: **for** every *class* created in the second step of the above loop **do**
8: **for** $i \in S_{class}$ **do**
9: $next = i + \ell_{E_{class}} - 1$;
10: $E_{label(post(t)[next]),class} \leftarrow (S_{label(post(t)[next]),class} \bigcup \{i\}, \ell_{E_{class}}, ac_{E_{class}})$;
11: **for** every *class* considered **do**
12: **if** $ac_{class} = 0$ **then**
13: Output S_{class}, ℓ_{class};
14: $sc = sc + 1$;
15: **for** $i \in S_{class}$ **do**
16: $T[i] \leftarrow sc$;
17: $TL[i] \leftarrow \ell[E_{class}]$;
18: Assign level(E_{class});
19: **else**
20: Partitioning(E_{class});
21: **return** Partitioning of E in classes corresponding to next element to be considered;

ALGORITHM SUBTREE-REPEATS-L($post(t), FC, P, H, n, \varphi$)

1: $sc \leftarrow 1$;
2: **for** $i \leftarrow 1$ to n **do**
3: **if** $\varphi(post(t)[i]) = 0$ **then**
4: $E_{label(post(t)[i])} \leftarrow (S_{label(post(t)[i])} \bigcup \{i\}, 1, 0)$;
5: $T[i] \leftarrow sc$;
6: $TL[i] \leftarrow 1$;
7: **else**
8: $T[i] \leftarrow 0$;
9: $TL[i] \leftarrow 0$;
10: **for** every $class$ considered **do**
11: Output S_{class}, 1;
12: $sc = sc + 1$;
13: Assign level(E_{class});
14: **for** $i \leftarrow 1$ to $H[n]$ **do**
15: **while** LA[i] non empty **do**
16: $E \leftarrow pop(LA[i])$;
17: Partition E;
18: **return** Sets of starting positions of Subtrees in post(t) together with their length;