Linear-size suffix tries

Maxime Crochemore, Chiara Epifanio, Roberto Grossi, Filippo Mignosi

Linear-Size Suffix Tries

Maxime Crochemore* Chiara Epifanio† Roberto Grossi‡ Filippo Mignosi§

Abstract

Suffix trees are highly regarded data structures for text indexing and string algorithms [MCreight 76, Weiner 73]. For any given string w of length $n = |w|$, a suffix tree for w takes $O(n)$ nodes and links. It is often presented as a compacted version of a suffix trie for w, where the latter is the trie (or digital search tree) built on the suffixes of w. Here the compaction process replaces each maximal chain of unary nodes with a single arc. For this, the suffix tree requires that the labels of its arcs are substrings encoded as pointers to w (or equivalent information). On the contrary, the arcs of the suffix trie are labeled by single symbols but there can be $\Theta(n^2)$ nodes and links for suffix tries in the worst case because of their unary nodes. It is an interesting question if the suffix trie can be stored using $O(n)$ nodes. We present the *linear-size suffix trie*, which guarantees $O(n)$ nodes. We use a new technique for reducing the number of unary nodes to $O(n)$, that stems from some results on antidictionaries. For instance, by using the linear-size suffix trie, we are able to check whether a pattern p of length $m = |p|$ occurs in w in $O(m \log |\Sigma|)$ time and we can find the longest common substring of two strings w_1 and w_2 in $O((|w_1| + |w_2|) \log |\Sigma|)$ time for an alphabet Σ.

Keywords: factor and suffix automata, pattern matching, suffix tree, text indexing.

1 Introduction

One of the seminal results in pattern matching is that the size of the minimal automaton accepting the substrings of a string, called directed acyclic word graph (DAWG), is linear according to the length of the string, and it can be built in linear time [2]. This result is surprising since the maximal number of substrings that may occur in a string is quadratic, and DAWG's arcs are labeled with single symbols. Suffix trees also contain a linear number of nodes, but they need to store substrings in the arcs (e.g. pointers to the input string).

Further work has been aimed at this direction. For example, a compact version of the DAWG and a direct algorithm to construct it is given in [9]. An algorithm for the online construction of DAWGs is described in [15] and [16]. Space-efficient implementations of compact DAWGs is presented in [13] and [14]. When compared to the well-known suffix trees, Blumer et al. [3] proved that, given an arbitrary string w of length $n = |w|$ and its reversal $\tilde{w}$, the following holds

$$n < e_{ST}(w) = e_{DAWG}(\tilde{w}) \leq 2n,$$

*King's College London, UK and Université Paris-Est, France (Maxime.Crochemore@kcl.ac.uk).

†Dipartimento di Matematica e Informatica, Università di Palermo, Italy (epifanio@math.unipa.it).

‡Dipartimento di Informatica, Università di Pisa, Italy (grossi@di.unipi.it). Partially supported by Project 2012C4E3KT AMANDA (Algorithmics for MAssive and Networked DAta).

§Dipartimento di Matematica e Applicazioni, Università degli Studi dell'Aquila, Italy (mignosi@di.univaq.it).

where $e_{ST}(w)$ and $e_{DAWG}(\tilde{w})$ represent, respectively, the number of nodes of the suffix tree for w and those of the DAWG for $\tilde{w}$. Moreover, considering the size of the two structures with respect to the same string w, they prove that

$$e_{ST}(w) \simeq e_{DAWG}(w)$$

where $\simeq$ indicates a property that holds on the average, as $\sum_{w\in\Sigma^n} e_{DAWG}(w) = \sum_{w\in\Sigma^n} e_{DAWG}(\tilde{w})$. For comparisons and results on this subject, we refer the reader to [4, 5].

Most of the research on suffix trees took instead different directions, for example, by compacting chains of single-child nodes as arcs labeled by substrings. As far as we know, the natural question of having suffix tries with $o(n^2)$ nodes in the worst case has not been investigated in the literature. Indeed it is not difficult to build some examples of input strings that require $\Theta(n^2)$ nodes for the suffix trie; however, this does not imply that all of these nodes are strictly necessary for efficient pattern searching. Patricias [22] posed this issue but required to know the length of the substring to be skipped during searching. Some related questions have been addressed for other variants of suffix trees, such as position heaps [10, 23], where the ideas proposed are completely different from those in this paper as the underlying structure is a heap rather than a trie. Also, the trick of encoding the input string as a path and its substrings as subpaths[1] does not answer the question as conceptually the arcs of the suffix tree are still (implicitly) labeled with substrings.

In this paper we present a new technique for reducing the number of nodes in suffix tries, which provides a totally different alternative to that of encoding substrings in suffix trees. We call the resulting data structure a *linear-size suffix trie*, shortly LST, for an input string w of n symbols. Denoting by $e_{LST}(w)$ the number of nodes of the LST of w, we prove that

$$e_{ST}(w) \leq e_{LST}(w) \leq e_{ST}(w) + n.$$

The LST can be built in $O(n)$ time from the suffix tree, and checking whether a pattern p of length $m = |p|$ occurs as a substring in w takes $O(m)$ time, by traversing a small part of the LST.[2] Since the arcs in the LST are labeled with single symbols and only a subset of the nodes are kept, we must recover the symbols which are lacking somehow without accessing the text w. We exploit suffix links on nodes and some of the arcs for this purpose. We think that the LST is as powerful as the suffix tree in most applications because it retains its topological structure. For example, we can find the longest common substring of two strings w_1 and w_2 in $O(|w_1| + |w_2|)$ time. We show some applications in Section 3 and give further discussion in the last section.

We remark that our contributions are at the moment mainly of theoretical nature. Anyhow, they may have some practical applications. First of all, a quick space analysis in the last section shows that LSTs can be made space-efficient. Second, the ideas and applications of LSTs can be extended to compact DAWGs. This is interesting as the size of compact DAWGs can be exponentially smaller than the size of the text for families of highly compressible strings.

This paper is organized as follows. Section 2 presents our data structure, the linear-size suffix trie. Section 3 contains our new search algorithms for locating a pattern with the compact tries and related automata and shows hot to use LST in order to find the longest common substring of two strings. Section 4 draws open problems and conclusions.

[1] Specfically, the input string w is encoded as path π of $n+1$ nodes with arcs labeled by single symbols, where the first node is the root of the suffix tree and the last node is the leaf corresponding to the suffix w. When encoding arcs by substrings $w[j, i]$, they are not represented by integer pairs (i, j); rahter, pointer pairs (p_i, p_j) can be employed, so that p_k points to the kth node in π for $k = i, j$.

[2] We assume that the alphabet of the symbols in w is constant-size, although the result can be extended to any alphabet Σ by multiplying the time complexity by a factor of $\log |\Sigma|$, as it is customary in the literature.

2 Linear-Size Suffix Trie (LST)

Given a string $w = a_1 a_2 \dots a_n$ of n symbols drawn from an alphabet Σ, we denote by $w[j, j+k-1]$ the substring of w of length k that appears at position j in w. The length of w is denoted by $|w| = n$. The empty string ϵ is a substring of any string. A substring of the form $w[j, |w|]$ (resp. $w[1, j]$) is called a suffix (resp. prefix) of w. We adopt the notation for strings defined in [20].

The suffix trie $ST(w)$ of a string w is a trie where the set of leaves is the set of suffixes of w that do not appear previously as substrings in w. The suffix tree $S(w)$ is the compacted version of $ST(w)$ where each maximal chain of unary nodes is replaced by a single arc whose label is the substring of w obtained by concatenating the symbols on the chain. It is often required in the suffix tree definition that the last symbol of w is a terminator not appearing elsewhere; here we do not need this requirement, assuming that for any suffix $w[j, |w|]$ of w there is a node v storing $w[j, |w|]$, even if $w[j, |w|]$ has appeared as substring in another position. We assume that the reader is familiar with suffix trees and related notions, such as suffix links [21]. In the following, we identify the nodes u with the substrings u of w that they represent, and we use suffix links for the nodes v in $S(w)$ or $ST(w)$: namely, $v = c \cdot s(v)$ where $s(v)$ is the node at the end of the suffix link for v and c is a symbol of the alphabet Σ.

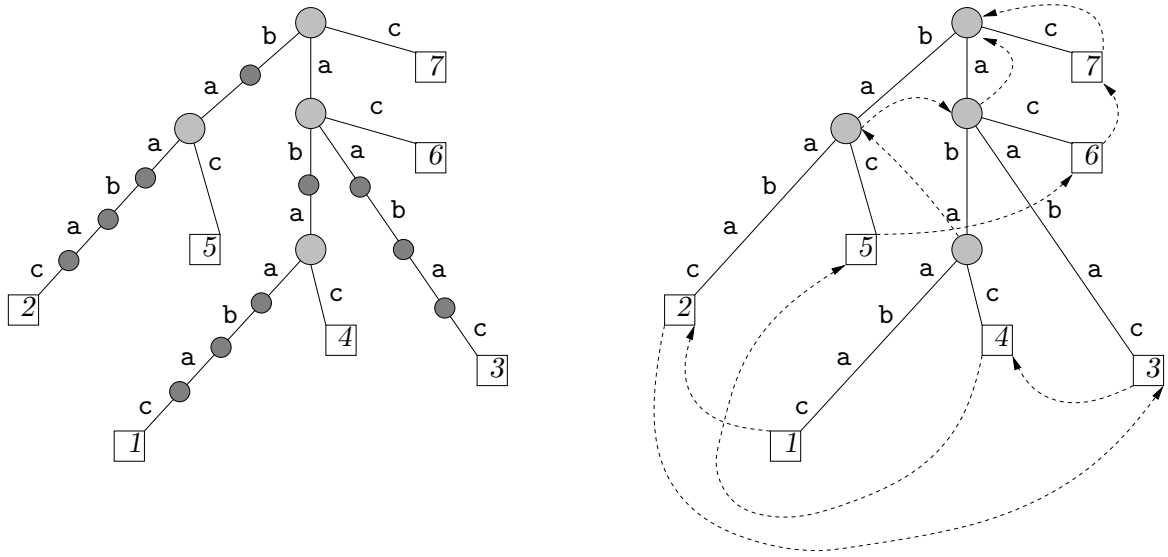

Figure 1: The suffix trie $ST(w)$(on the left) for string $w =$ `abaabac`, and its suffix tree $S(w)$ (on the right), where the suffix links are shown as dashed arrows and the substring labels are represented as pairs of integers (not shown).

Basic definitions. We now introduce our linear-size suffix trie $LST(w)$. Basically, we use the same approach of the suffix tree $S(w)$, but we define an alternative way of compacting. Only the symbols labeling the arcs of $LST(w)$ will suffice. As mentioned before, the interesting property is that $LST(w)$ uses $O(n)$ nodes, arcs, suffix links, and symbols in total.

The set of nodes of $LST(w)$ is made up of all the nodes that appear also in the suffix tree $S(w)$, plus some nodes that appear only in the suffix trie $ST(w)$. Namely, these nodes are selected in the suffix trie $ST(w)$ according to the following criterion.

1. The nodes that appear also in the suffix tree $S(w)$, and

2. the nodes v such that their suffix link $s(v)$ is a node appearing also in $S(w)$.

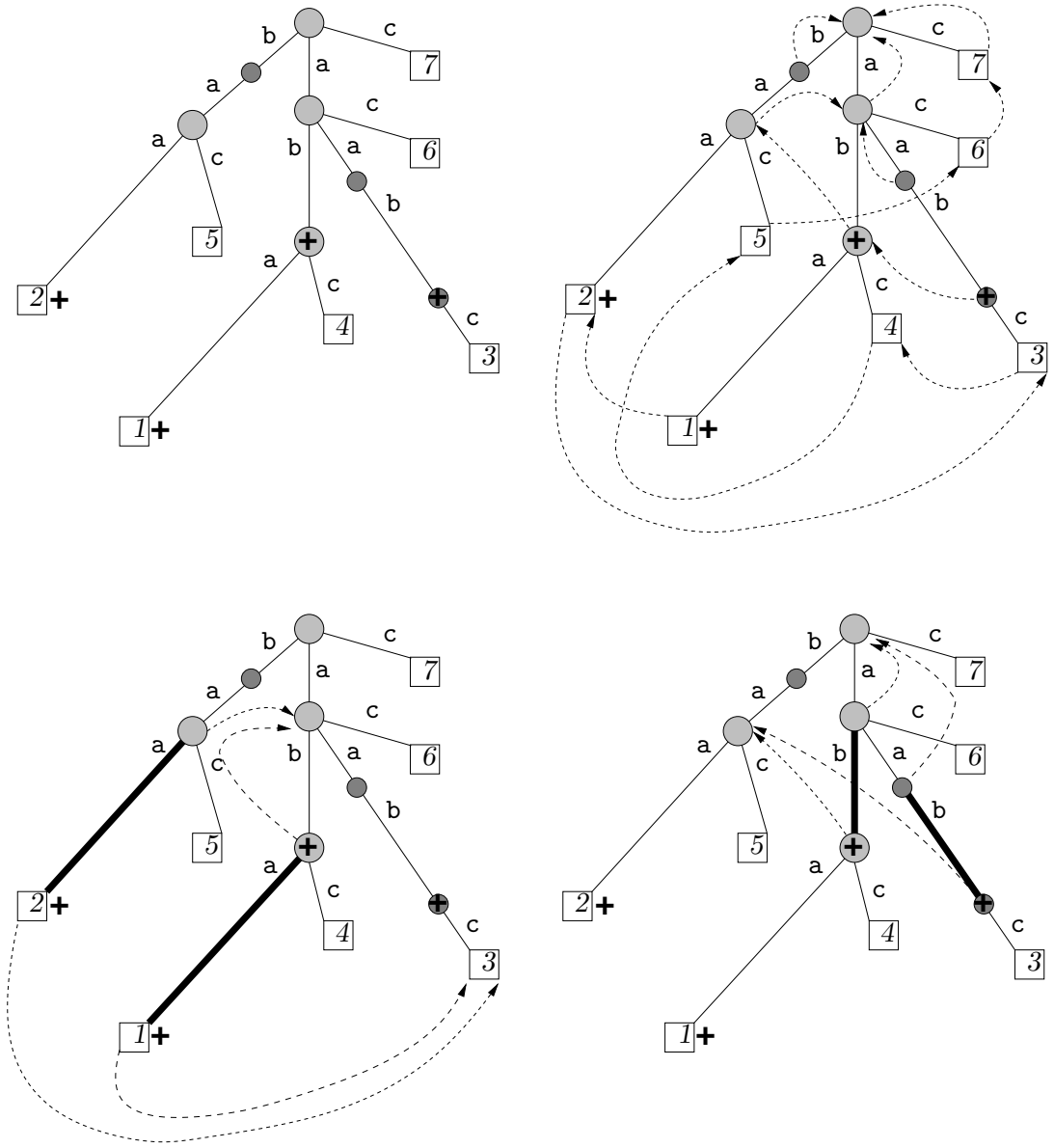

Figure 2: The linear-size suffix trie $LST(w)$ for $w =$ `abaabac`. Top left: the basic structure with nodes of type 1 (big bullets) and type 2 (small bullets), and + signs. Top right: the suffix links for its nodes represented as dashed arrows. Bottom left and right: the suffix links for the arcs of length greater than one (one pair per arc), where the "source" arcs are represented as boldline.

Example 1. Let $w =$ `abaabac` be a string. Figure 1 shows the suffix trie $ST(w)$ and the suffix tree $S(w)$, while Figure 2 shows the linear-size suffix trie $LST(w)$, with nodes of type 1 (big bullets), of type 2 (small bullets), and suffix links (dashed arrows). As explained below, the + signs denote nodes whose parent arc appears also in $ST(w)$ with a substring of length greater than 1 as a label.

Remark 1. The children of the root of the suffix trie $ST(w)$ survive and can be of both types. If not of type 1, they are of type 2 as $s(v) = \epsilon$ for each such node v. Also, the nodes of $LST(w)$ are clearly a subset of the nodes of $ST(w)$.

We now define the arcs of $LST(w)$ as follow. For each node $v \in LST(w)$ we assign the same number of outgoing arcs as its appearance $v \in ST(w)$, each of them labeled by its distinct symbol. In particular, for any arc of the form (v, v') with label $a \in \Sigma$ in $ST(w)$, we have an arc (v, x) with label a in $LST(w)$, where x is the closest descendant of v' in $ST(w)$ such that x is also in $LST(w)$ (hence $x = v'$ when v' is also in $LST(w)$).

We observe that the arc (v, x) in $LST(w)$ represents the whole path from v to x in $ST(w)$. When $x = v'$, we say that this arc has length one; when $x \neq v'$, we say that this arc has length greater than one and add a + sign to node x, in order to record this information. We observe that the children of the root of $LST(w)$ have no a + sign as their path is of length one (see Remark 1). In general, we are only missing symbols for arcs of length greater than one, and, in this case, we add the + sign (see Figure 2).

We retain the definition of a suffix link over the nodes of $LST(w)$ as it is well defined. Indeed, for any node v of $LST(w)$, the node pointed to by its suffix link $s(v)$ is always a node appearing also in the suffix tree $S(w)$, and thus $s(v) \in LST(w)$.

We introduce suffix links *also* for some arcs, namely, those of length greater than one. Consider an arc (u, v) where v is marked with a + (note that u cannot be the root). We define its suffix link $s(u, v)$ as the pair of vertices $s^k(u)$ and $s^k(v)$, where $s^k(x)$ indicates the traversal of k suffix links starting from x, and k is the smallest integer such that $s^k(u)$ is no more the parent of $s^k(v)$. The bottom part of Figure 2 reports some examples of these suffix links.

Note that the suffix links for arcs (u, v) are well defined as well. In the worst case, $s^k(u)$ is the root, and all outgoing arcs from the root have length one, and thus there is a node between $s^k(u)$ and $s^k(v)$, where $k \leq |u|$. We will use suffix links for arcs to speed up the search in $LST(w)$ (see Section 3). Also, depending on the choice of the alphabet Σ, the children of each node can be stored in unsorted fashion when Σ is constant-size or hashable, or in sorted fashion when Σ is comparison-based and arbitrarily large.

Properties. We now show that the size of $LST(w)$ is linear in the length $n = |w|$ of w, independently of the alphabet size.

We first give a linear upper bound on number of nodes of $LST(w)$. The nodes of type 1 appear also in the suffix tree, thus they are at most $2n$ in number [21, 25]. We have to prove an upper bound for the number of nodes of type 2. A crude bound is $O(n|\Sigma|)$, as there can be $O(|\Sigma|)$ nodes of the suffix trie having a suffix link to the same node appearing in the suffix tree. As it can be $|\Sigma| = O(n)$, this gives $O(n^2)$ nodes in the worst case. Fortunately, there are at most overall $O(n)$ nodes of type 2 in $LST(w)$ as shown next.

Lemma 1. *Let Σ be any alphabet and w be a string over Σ of length $n = |w|$. Then, the number of nodes of type 2 in the linear-size suffix trie $LST(w)$ is smaller than the string length n.*

Proof. Let $w = a_1 a_2 \dots a_n$, with $a_i \in \Sigma$. For any integer i, $1 \leq i \leq |w| - 1$, we claim that there is at most one node v of the suffix trie $ST(w)$ such that

- v does not belong to the suffix tree $S(w)$,
- $s(v)$ is a node of the suffix tree $S(w)$, and
- v ends in position i, i.e., $v = w[j, i]$ for some $j < i$.

Since ϵ, which is the root, and w itself are nodes of the suffix tree $S(w)$, then the proof of this lemma follows by the claim. Now, suppose by contradiction that there exist two distinct such nodes, $v = w[j, i]$ and $v' = w[j', i]$ that both end in position i. Suppose that v' is longer than v,

i.e., $1 \leq j' < j < i$ (as the other way is analogous). Hence v is a suffix of v' and, consequently, is a suffix of $s(v')$ as well.

Now, the fact that v and v' are not nodes of $S(w)$ implies that they have only one child in the suffix trie $ST(w)$, and neither of them are suffixes of w. This, in turn, implies that there is a unique symbol a that follows *any* occurrence of both v and v' in w, where $a = w_{i+1}$. The fact that $s(v')$ is a node of the suffix tree means that either $s(v')$ is a suffix of w, or there is another occurrence of $s(v')$ not ending in position i and followed by a symbol $b \neq a$. Since v is a suffix of $s(v')$, the same condition holds for v, which is a contradiction. □

Putting it together, the number of arcs and their labeling symbols, as well as the number of suffix links for nodes, is asymptotically bound above by the number of nodes in $LST(w)$. Also, the number of suffix links for arcs is bound above by the number of arcs. We can therefore conclude that together they are $O(n)$ in total.

Theorem 1. *Let Σ be any alphabet and w be a string over Σ of length $n = |w|$. The total size of $LST(w)$ is $O(n)$ space.*

Concerning space requirements, a quick space analysis shows that LSTs are in practice slightly space-saving with respect to suffix trees, even considering that in some applications there is the need of some additional space such as for other new suffix links, or the space for storing the lengths of arcs needed for solving LCS. Indeed in LSTs there is a number smaller than n of new nodes (see Theorem 1) and for each node there is exactly one new arc. On the other side in LSTs the two pointers (length and position in the text) associated with any arc (where the total number of arcs is bounded by $2n - 2$) are replaced with one symbol. Anyhow, the LSTs cannot compete for instance, in terms of space efficiency, with some relatively new data structures, such as compressed suffix trees [24] and position heaps [10].

The construction of $LST(w)$ can be easily done in linear time by a simple post-processing after the corresponding suffix tree construction is given. Indeed, one can make a bottom-up traversal of $S(w)$, and for each arc (u, v) consider the path (not necessarily a single arc) in the suffix tree between $s(u)$ and $s(v)$. With a standard skip-and-count technique on this path, one can then infer which are the nodes of type 2 between u and v. If required in some applications, during this construction we can also equip any arc of $LST(w)$ with the corresponding length. Furthermore, the time bounds stated in this paper are valid under the uniform cost model and the hypothesis that the size of the alphabet is constant. Otherwise, we have a $\log |\Sigma|$ multiplicative term. These assumptions are classical in the theoretical approach that we are using in this paper.

Lemma 2. *Let Σ be any alphabet and w be a string over Σ of length $n = |w|$. Given the suffix tree $S(w)$ for w, the linear-size suffix trie $LST(w)$ can be built in $O(n \log |\Sigma|)$ time using $O(n)$ space.*

3 Traversing Linear-Size Tries with Applications

For the linear-size suffix trie $LST(w)$ of a string w, we describe the basic task of reconstructing the missing symbols on an arc of length greater than one. Using this task, we show how to check whether a pattern p occurs as a substring in w. Note that the arcs in $LST(w)$ are labeled with single symbols instead of substrings, so we must recover the missing symbols somehow. Recall that the outgoing arcs from the root are of length one. Thus we can focus on those arcs leaving from nodes other than the root.

First of all, we begin by using only the suffix links for the nodes. We introduce a recursive algorithm that takes as input an arc (u, v) in the linear-size suffix trie $LST(w)$, where u is different

from the root, and returns the string β associated with the corresponding path between the two nodes u and v in the suffix trie $ST(w)$. We first check if there is no + sign in v, i.e., if the arc is of length one. In this case (lines 3 and 4), we directly return the associated symbol a as β. Otherwise (lines 6–11), we know that a is the first symbol of β, and use the suffix links for u and v to recursively find the remaining symbols between $z = s(u)$ and z', where (z, z') is the arc leaving from z and labeled with a, and between z' and $s(v)$. Here the while loop replaces the second recursive call (for z' and $s(v)$), as it is a tail recursion.

```
DECOMPACT (arc (u, v)) for the linear-size suffix trie LST(w):
 1. β ← ϵ;
 2. a ← label of (u, v);
 3. if v does not have a + sign then
 4.       β ← a;
 5. else
 6.       z ← s(u);
 7.       while z ≠ s(v)
 8.             β ← β · DECOMPACT((z, z')), where (z, z') is labeled with a;
 9.             z ← z';
10.             if z has only one child x then
11.                   a ← label of (z, x);
12. return β
```

The proof of correctness for algorithm DECOMPACT is established by induction using the following simple property.

Proposition 1. *Let (u, v) be an arc in the linear-size suffix trie $LST(w)$. Then, if the path from $s(u)$ to $s(v)$ contains intermediate nodes, they are unary (i.e. not branching).*

Proof. Suppose by contradiction that there exists a node y in the path from $s(u)$ to $s(v)$ that branches. By definition, node y is of type 1 as it appears also in the suffix tree $S(w)$. Now, let c be the first symbol of u and v, which must be the same as u is a proper prefix of v. Also, observe that $s(u)$ is a proper prefix of y, and y is a proper prefix of $s(v)$. Consequently, u is a proper prefix of $c \cdot y$, and $c \cdot y$ is a proper prefix of v. Since $s(c \cdot y) = y$, the node $c \cdot y$ must be of type 2, lying between u and v in $LST(w)$, thus contradicting the hypothesis that (u, v) is an arc of $LST(w)$. Hence, the branching node y cannot exist. □

Lemma 3. *Algorithm* DECOMPACT *correctly recovers the substring β corresponding to the arc (u, v).*

Proof. The proof is by induction on the length ℓ of the arc (u, v). If $\ell = 1$, then there is no + sign in v, and lines 3 and 4 correctly return the symbol a. If $\ell > 1$, we observe that eventually there will be an arc of length one between $s^k(u)$ and $s^k(v)$ in $LST(w)$ for a value of $k \leq |u|$. Indeed, in the worst case, $s^k(u)$ becomes the root, and all arcs leaving from the root have length one. This is why algorithm DECOMPACT recursively examines $s(u)$ and $s(v)$ in lines 6–11. By Proposition 1, we know that the path between $s(u)$ and $s(v)$ in $LST(w)$ has no branching nodes. Therefore, it is a sequence of arcs. Line 9 of algorithm DECOMPACT allows the shortening of the length of the path in the case that $z \neq s(v)$. If the check at line 10 fails, it implies that $z = s(v)$, and $s(v)$ branches, and thus the while loop terminates. This completes the induction. □

An easy variation of algorithm DECOMPACT can be used to check whether a pattern $p = p[1, m]$ occurs in w. More precisely, algorithm SEARCH takes as input a node u and a pattern p, where initially u is the root of $LST(w)$. If there is no outgoing arc from u labeled $p[1]$, it returns `null`. If such an arc exists, say (u, v), it "decompacts" (u, v) and proceeds with the rest of the matching task. Here, p is seen as a *global* pointer variable to the pattern (as in the C language) so that an assignment $p \leftarrow p[2, m]$ in line 9 has a global effect through the recursive calls.

```
SEARCH (node u, pattern p) for the linear-size suffix trie LST(w):
 1. if p = ε then
 2.     return(u)
 3. else
 4.     if ∄ arc (u, v) labeled with p[1] then
 5.         return null
 6.     else
 7.         let (u, v) be the arc labeled with p[1];
 8.         if v does not have a + sign then
 9.             p ← p[2, m];
10.             SEARCH(v, p)
11.         else
12.             z ← s(u);
13.             while z ≠ s(v)
14.                 let (z, z') be the arc with label p[1];
15.                 SEARCH(z,p)
16.                 z ← z'
```

The proof of the correctness of algorithm SEARCH follows by Proposition 1.

Lemma 4. *Algorithm* SEARCH *correctly finds if p occurs in w using $LST(w)$.*

As for the time complexity, the drawback is that SEARCH may require more than $O(m)$ time to find p. More precisely, it could be that $s(u)$ and $s(v)$ are still an arc: it costs $O(1)$ time, but we do not "decompact" any further symbols. So this cannot be amortized to guarantee an $O(m)$ cost.

To circumvent this drawback, we need to employ the suffix links for the arcs of greater than unit length. To see how these can help us in our goal, consider the task of decompacting the arc (u, v) in order to reconstruct its corresponding substring β. We could potentially waste $O(k) = O(m)$ time by repeatedly considering $(s(u), s(v))$, $(s^2(u), s^2(v))$, and so on, until reaching the first k such that $s^k(u)$ and $s^k(v)$ are no longer directly connected by an arc in $LST(w)$: only when reaching $s^k(u)$ and $s^k(v)$ we can discover one further symbol from β. Instead, using the suffix link $s(u, v)$ for the arc, we can jump directly to $s^k(u)$ and $s^k(v)$, and discover one further symbol. We can do this in amortized time, since we pay $O(1)$ time, instead of $O(k)$ time, for any discovered symbol. In such a way, we obtain a new algorithm FASTDECOMPACT, where lines 6–8 of DECOMPACT change to exploit the suffix link $s(u, v)$. Algorithm FASTDECOMPACT works in linear time as shown below.

```
FastDecompact (arc (u, v)) for the linear-size suffix trie LST(w):
 1. β ← ϵ;
 2. a ← label of (u, v);
 3. if v does not have a + then
 4.     β ← a;
 5. else
 6.     let s^k(u) and s^k(v) be the suffix link s(u, v) for the arc (u, v);
 7.     z ← s^k(u);
 8.     while z ≠ s^k(v)
 9.         β ← β · Decompact((z, z')), where (z, z') is labeled with a;
10.         z ← z';
11.         if z has only one child x then
12.             a ← label of (z, x);
13. return β
```

As a result of the linearity of FastDecompact, we have the following result, form which obtaining FastSearch is immediate.

Theorem 2. *Given the linear-size suffix trie $LST(w)$, reconstructing the substring of an arc of length ℓ takes $O(\ell)$ time and, consequently, finding if a pattern p of length m occurs in w takes $O(m)$ time. For an arbitrary alphabet Σ, this cost is multiplied by a factor of* $\log|\Sigma|$.

3.1 Longest Common Substrings and Generalized Linear-Size Suffix Tries

A natural question is whether the linear-size suffix trie has limitations in the application domains when compared to the suffix tree. A short answer is that the topology of the two trees are intimately related as we show next for the longest common substring problem.

We consider the case of two strings, but the interested reader can easily extend the techniques described here to the general case. We recall that the problem of finding the *longest common substring* (LCS) is a classic problem in string algorithms, where a set $\mathcal{S}$ of $q \geq 2$ strings and an integer $2 \leq d \leq k$ are given as input, and the length of the longest string that appears in at least d them is to be determined [11]. An almost immediate solution for $k = 2$ using the suffix tree of two concatenated strings separated by a symbol was available since 1973, when Weiner published his famous linear-time construction algorithm for suffix trees [25].

Concerning the LCS problem in its full generality, we recall that in 1992, Hui [17] gave a linear-time solution using a famous constant-time solution for the lowest common ancestor (LCA) in trees coupled with the generalized suffix tree. The journal version of this result appeared some years later in [18]. It seems that the natural notion of generalized suffix tree for a set of strings was firstly introduced in 1992 by Gusfield and others in [12] and used in the same year by Hui in order to settle the LCS problem [7]. This notion thus appeared 22 years after the Weiner's suffix tree construction algorithm, even if it was implicit in Weiner's paper [25] in the case of a set of two strings.

We just define the generalized linear-size suffix trie of two strings and leave to the reader its obvious extension to the general case of q strings. We are given two strings $w_1\$_1$ and $w_2\$_2$, where the two symbols $\$_1$ and $\$_2$ never appear elsewhere in the concatenated string w_1w_2. Analogously as done in the case of one string, the set of nodes of the generalized linear-size suffix trie $GLST(w)$ includes the nodes of the generalized suffix tree $GS(w)$ and adds some extra unary nodes from the generalized suffix trie $GST(w)$. Namely, it contains the following nodes of $GST(w)$, where we adopt an analogous definition of suffix links.

1. The nodes that appear also in the generalized suffix tree $GS(w)$, and

2. the nodes v of such that $s(v)$ appears in the generalized suffix tree $S(w)$, where $s(v)$ is the ending node of the suffix link.

The arcs are defined analogously as in the case of a single string. Another way to see the generalized linear-size suffix trie, following one approach described in [11] is to consider the linear-size suffix trie of the single string $w_1\$_1 w_2\$_2$ that is the concatenation of $w_1\$_1$ and of $w_2\$_2$, and then to eliminate the "artificial" suffixes that follows the symbol $\$_1$. This second approach guarantees the linearity in space of the generalized linear-size suffix trie.

In order to solve the LCS problem between $w_1\$_1$ and $w_2\$_2$, let us suppose that we have the string lengths of the arcs of length greater than one. We do not need the suffix links on these arcs. Then we proceed exactly in analogous way than in the case of the generalized suffix trees. This comes at no surprise at all. With a standard traversal we find a node that have the greatest string depth (starting from the root), such that the leaves in its subtree represent suffixes of both strings. The path label from root to this node gives a solution to the problem.

4 Conclusions, Open Problems and Further work

In this paper we presented a new technique for obtaining suffix tries of linear size. Our techniques can also be extended to tries and automata, as discussed in previous work [6]. The results in this paper are an extension of some ideas originally described in [6, 8].

Linear-size suffix tries can be employed, analogously to suffix trees, to solve other problems different from pattern matching as the substrings labels and their lengths can be recovered. We think that linear-size suffix tries have the potential to be used in the place of standard suffix trees in all applications because their underlying trees structures and the suffix links are close enough. We find somehow surprising that a simple data structure like LST has not been discovered earlier. We further think that for large enough text generated by a memoryless source, the search for patterns generated by the same source can also be done in average linear time without using suffix links on the arcs whose ending node has a + sign. We leave as an open problem to give an average analysis for this case.

The ideas developed in this paper, as long as the applications we presented, may be extended to CDAWGs and this will be object of further work. In this setting, the size of a CDAWG can be exponentially smaller than the one of the text for families of highly compressible strings, as in the case of CDAWGs of Sturmian words.

Acknowledgments

We are deeply grateful to Jackie Daykin for her comments on improving the paper.

References

[1] A. Apostolico, C. Guerra *The Longest Common Subsequence Problem Revisited* Algoritmica, **2**, pp. 315–336 (1987).

[2] A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, J. Seiferas. *The Smallest Automaton Recognizing the Subwords of a Text*, Theoretical Computer Science, **40(1)**, pp. 31–55 (1985).

[3] A. Blumer, A. Ehrenfeucht, and D. Haussler *Average sizes of Suffix Trees and DAWGs*, Discrete Applied Mathematics, **24(1–3)**, pp. 37–45 (1989).

[4] D. Breslauer, G. F. Italiano. *Near real-time Suffix Tree construction via the fringe marked ancestor problem*, Journal of Discrete Algorithms, **18**, pp. 32–48 (2013).

[5] M. Crochemore, *Reducing space for index implementation*, Theoretical Computer Science, **292(1)**, pp. 185–197 (2003).

[6] M. Crochemore, C. Epifanio, R. Grossi, F. Mignosi, *A Trie-Based Approach for Compacting Automata*, Proceedings of CPM04, LNCS **3109**, pp. 145-154 (2004).

[7] M. Crochemore, C.S. Iliopoulos, A. Langiu, F. Mignosi, *The longest common substring problem*, Mathematical Structures in Computer Science, in press, 2015.

[8] M. Crochemore, F. Mignosi, A. Restivo, S. Salemi, *Data compression using antidictonaries*, in Special issue *Lossless data compression*, J. Storer ed., Proceedings of the IEEE, **88(11)**, pp. 1756–1768 (2000).

[9] M. Crochemore, R. Vérin, *Direct Construction of Compact Directed Acyclic Word Graphs*, Proceedings of CPM97, LNCS **1264**, Springer-Verlag, pp. 116–129 (1997).

[10] A. Ehrenfeucht, R.M. McConnell, N. Osheim, S.-W. Woo *Position heaps: A simple and dynamic text indexing data structure*, Journal of Discrete Algorithms **9(1)**, pp. 100–121 (2011).

[11] D. Gusfield. *Algorithms on Strings, trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press (1997).

[12] D. Gusfield, G. M. Landau, B. Schieber *An Efficient Algorithm for the All Pairs suffix-Prefix Problem*, Inf. Process. Lett. **41(4)**, pp. 181–185 (1992).

[13] J. Holub. *Personal Communication* (1999).

[14] J. Holub, M. Crochemore. *On the Implementation of Compact DAWG's*, Proceedings of CIAA02, LNCS **2608**, Springer-Verlag, pp. 289–294 (2003).

[15] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, G. Pavesi. *On-Line Construction of Compact Directed Acyclic Word Graphs*, Discrete Applied Mathematics **146(2)**, pp. 156–179 (2005).

[16] S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, G. Pavesi. *On-Line Construction of Compact Directed Acyclic Word Graphs*, Proceedings of CPM01, LNCS **2089**, Springer-Verlag, pp. 169–180 (2001).

[17] L. C. K. Hui *Color Set Size Problem with Application to String Matching*, Proceedings of CPM92, LNCS **644**, Springer-Verlag, pp. 230–243 (1992).

[18] L. C. K. Hui *A practical algorithm to find longest common substring in linear time*, Int. J. of Comput. Syst. Sci. & Eng. **15(2)**, pp. 73–76 (2000).

[19] D. E. Knuth, J. H. Morris Jr., V. R. Pratt. *Fast Pattern Matching in Strings*, SIAM J. Comput. **6(2)**, pp. 323–350 (1977).

[20] M. Lothaire. *Algebraic Combinatorics on Words.* Encyclopedia of Mathematics and its Applications, **90**, Cambridge University Press (2002).

[21] E. M. McCreight. *A Space-Economical Suffix Tree Construction Algorithm*, Journal of the ACM, **23(2)**, pp. 262–272 (1976).

[22] Donald R. Morrison. *PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric*, Journal of the ACM, **4(15)**, pp. 514–534 (1968).

[23] Y. Nakashima, T. I, S. Inenaga, H. Bannai, M. Takeda. *The Position Heap of a Trie*, Proceedings of SPIRE2012, LNCS **7608**, Springer-Verlag, pp. 360–371 (2012).

[24] L.M.S. Russo, G. Navarro, A.L. Oliveira. *Fully compressed suffix trees*, ACM Transactions on Algorithms (TALG) **7(4)**, pp. 53:1–53:34 (2011).

[25] P. Weiner. *Linear pattern matching algorithms*, Proceedings of SWAT73, IEEE Computer Society Washington, pp. 1–11 (1973).