Reverse Engineering of Compact Suffix Trees and Links: a Novel Algorithm*

Bastien Cazaux,† Eric Rivals†
†L.I.R.M.M. & Institut Biologie Computationnelle
Université de Montpellier II, CNRS U.M.R. 5506
161 rue Ada, F-34392 Montpellier Cedex 5, France
cazaux@lirmm.fr, rivals@lirmm.fr

5th July 2014

Abstract

Invented in the 70's, the Suffix Tree (ST) is a data structure that indexes all substrings of a text in linear space. Although more space demanding than other indexes, the ST remains an inspiring index likely because it represents substrings in a hierarchical tree structure. Along time, STs have acquired a central position in text algorithmics with myriad of algorithms and applications to for instance motif discovery, biological sequence comparison, or text compression. It is well known that different words can lead to the same suffix tree structure with different labels. Moreover, the properties of STs prevent all tree structures from being STs. Even the suffix links, which play a key role in efficient construction algorithms and many applications, are not sufficient to discriminate the suffix trees of distinct words. The question of recognising which trees can be STs has been raised and termed Reverse Engineering on STs. For the case where a tree is given with potential suffix links, a seminal work provides a linear time solution only for binary alphabets. Here, we also investigate the Reverse Engineering problem on ST with links and exhibit a novel approach and algorithm. Hopefully, this new suffix tree characterisation makes up a valuable step towards a better understanding of suffix tree combinatorics.

*This work is supported by ANR Colib'read (ANR-12-BS02-0008) and Défi MASTODONS SePhHaDe from CNRS. This article is published in J. Discrete Algorithms in 2014, see [2] http://dx.doi.org/10.1016/j.jda.2014.07.002.

†

1 Introduction

Forty years after its invention, the Suffix Tree (ST) remains a ubiquitous data structure for indexing all substrings of a text [3] and still inspires many researchers. Given a word w, the ST of w can be build in linear time and space using well-known construction algorithms [9, 6, 8, 1, 3]. To achieve linear time construction, all internal nodes of a ST are equipped with an edge of a different type called a *suffix link*. Once built, the ST is kept in memory and serves for matching exactly any given pattern in a time linear in the pattern length rather than in the text length [3]. However, approximate pattern matching or overlap matching can also be achieved with STs, as well as many other applications, for instance in bioinformatics [3].

To any word corresponds a suffix tree. As illustrated in Figure 1, two distinct words can give rise to the same suffix tree (in terms of structure, not in terms of labels). Moreover, not all tree structures can be a suffix tree. Hence, the reverse engineering problem (REST), which, given a tree asks for a word whose suffix tree is isomorphic to the input tree, is a natural, but non trivial question. How does one characterise a tree that is a suffix tree? Which strings do generate the same ST? How many distinct STs exist for strings of a given length? All these questions remain open, not mentioning enumeration or random sampling. To our knowledge, the problem REST has been studied in a master thesis [7][1], and in a special case in one article, namely when the tree and its potential suffix links[2] are given as inputs [4, 5]. We call this version the *SLI-REST* (for Suffix Links on Internal nodes - REST). Both reverse engineering problems are fundamental for enumerating, counting, and even uniformly sampling suffix trees at random, in other words for understanding their combinatorics. The reverse engineering problem has been raised and addressed for other data structures like border arrays, suffix arrays, etc (see references in [5]).

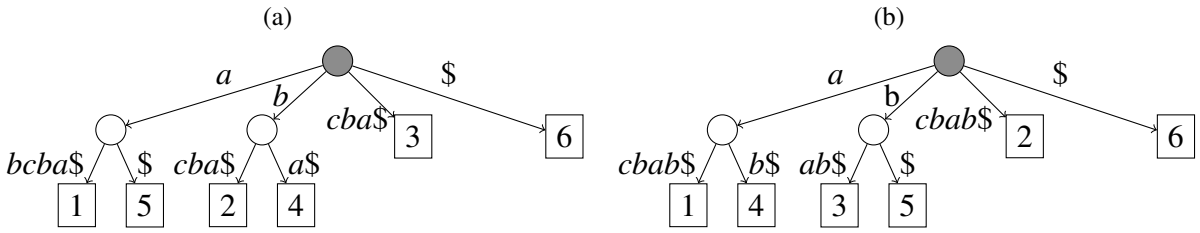


Figure 1: Two different strings with an identical suffix tree structure: in (a) the string *abcba*\$ and in (b) the string *acbab*\$. Note that in both trees the suffix links (not represented) of internal nodes are identical (they go to the root). Almost all labels from internal nodes to leaves are distinct between the two suffix trees, even their lengths differ. No permutation of the alphabet symbols can transform *abcba*\$ into *acbab*\$.

The SLI-REST problem is defined as follows: given a tree T and a tree of additional links F (on the same nodes as T), find a string whose ST and Suffix Links (SL) are isomorphic to (T,F). In [4, 5], the authors first define SLI-REST, but then investigate a restricted version of it where T is

[1] A study that did not come up with a characterisation.

[2] We qualify those links of potential to distinguish them from true suffix links.

an ordered tree and is equipped with a *labelling function* giving for each edge the first symbol of its label. They propose an algorithm that builds the Suffix Tour Graph on the nodes of T, and checks whether it contains a Eulerian cycle including the root and all leaves of T. On a binary alphabet, the authors show with combinatorial arguments that the number of labelling functions is bounded by a constant, and hence their algorithm runs in linear time. Since it explores all labelling functions, the same algorithm can also output all strings solving SLI-REST. However, the combinatorial explosion of possible labelling functions for a larger alphabet makes it inappropriate. Here, we propose a different approach for the general version of SLI-REST (where T is neither ordered, nor labelled) and organise the paper as follows. Notation, definitions, and known properties are written below for strings, suffix trees, and other concepts. Section 2 explains how to compute the labels for all internal edges, Section 3 details the case where all nodes are equipped with potential suffix links, while Section 4 explains how to find the potential suffix links of leaves and proposes an algorithm for solving SLI-REST. Finally, Section 5 summarises the differences and improvements between [5] approach and ours, and Section 6 concludes.

1.1 Notation on strings, trees and graphs.

An *alphabet* Σ is a finite set of *letters*. A *word* or *string* over Σ is a finite sequence of elements of Σ. The set of all words over Σ is denoted by $\Sigma^\star$, and ε denotes the empty word. For a word x, $|x|$ denotes the *length* of x. Given two words x and y, we denote by xy the *concatenation* of x and y. For every $1 \leq i \leq j \leq |x|$, $x[i]$ denotes the i-th letter of x, and $x[i\,;\,j]$ denotes the *substring* $x[i]x[i+1]\dots x[j]$. The suffix of x starting at position i is denoted $\mathrm{suff}_i(w)$. We denote by $\#(\Lambda)$ the cardinality of any set Λ.

A directed graph $G := (V_G, E_G)$ consists of a set of vertices V_G and a set of directed edges E_G that is a subset of $V_G \times V_G$. A *Eulerian cycle* in a graph G is a simple path that visits each edge exactly once (with the same start and end vertices). It can be seen as a cyclic permutation of E_G. A *Eulerian route* is a traversal of a Eulerian cycle starting at a given vertex. The $\#(E_G)$ possible routes of a Eulerian cycle are all cyclic shifts one of another. A tree is a graph in which any two vertices can be connected by a unique simple path. In a rooted tree T, the *root* is denoted by $\bot_T$, the vertices of V_T are partitioned into *internal* nodes and *leaves*; so, $V_T := V_T^{in} \cup V_T^{leaf}$. For any node $v \in V_T$, $f_T(v)$ denotes its unique *father*, while $Children_T(v)$ is its set of *children*. A leaf has no children and the root has no father. Moreover, a child of v that is a leaf is termed a *chleaf*; thus, children of v are partitioned into chleaves and non-leaf children (also called *chin*), which we denote by $Chleaves_T(v) := Children_T(v) \cap V_T^{leaf}$ and $Children_T(v) := Chleaves_T(v) \cup Chin_T(v)$. In addition, $V_T(v)$ denotes the set of nodes of the subtree of T rooted in v.

An *alphabetisation* of a tree T on an alphabet Σ is an injective mapping from the set of edges going out $\bot_T$ to Σ. A *labelling* l of a tree T on an alphabet Σ is a mapping from E_T onto Σ^*: it maps an edge e to a word $l(e)$ of Σ^*. Let $u, v \in V_T$ such that $v \in V_T(u)$, $\tilde{l}(u,v)$ denotes the word formed by the concatenation of edge labels on the (unique) path joining u to v in T, and $l(v) := l(f_T(v), v)$. The (unique) *word represented by node* v with labelling l is $\tilde{l}(v) := \tilde{l}(\bot_T, v)$.

Note that the notion of labelling function of [5] assigns the first letter to the labels of all edges; so it does not coincide with our notion of labelling.

1.2 Suffix trees

The *suffix tree (ST)* of a string w, denoted by ST_w, is a labelled rooted tree that encodes all the suffixes of w. We assume that any string ends with a special symbol $\$ \notin \Sigma$, which implies that any two suffixes cannot be a prefix of each other. The definition of a suffix tree requires that

C1: an internal node has at least two children,

C2: the labels of edges going from a node to its children start with distinct symbols,

C3: each suffix is represented by a leaf and conversely; leaves are numbered by the starting position of their suffix,

C4: each edge is labelled by a non empty substring of w such that the concatenation of the labels found on the path from a root to a leaf spells out the corresponding suffix.

Let Σ_w denote the alphabet of w; let T_w and l_w denote respectively the rooted tree of ST_w and the labelling induced by w on T_w. In other words, ST_w is a pair (T_w, l_w).

From the above characterisation of ST_w, several well-known properties follow:

P1: Its number of leaves equals the length of w ($\#(V_{T_w}^{leaf}) = |w|$); by C1 its number of internal nodes satisfies $\#(V_{T_w}^{in}) \leq |w| - 1$, and thus its total number of nodes (and of edges) satisfies $\#(V_{T_w}) \leq 2|w| - 1$, which means altogether that the size of T_w is linear in $|w|$. Note that the size of l_w can be quadratic in $|w|$.

P2: By C1 and C2, the number of children of any internal node is smaller than the cardinality of Σ_w (*i.e.* $\#(Children_{T_w}(v)) \leq \#(\Sigma_w)$). Moreover, the root $\perp_{T_w}$ has exactly $\#(\Sigma_w)$ children and exactly one of its leaves is labelled by \$ (by hypothesis).

P3: An internal node v represents a maximal prefix, namely $\tilde{l_w}(v)$, shared by all suffixes located at the leaves of its subtree.

P4: Let λ be a bijective mapping from Σ_w onto any other alphabet, say Γ. Clearly, substituting the letters of w using λ yields a word $\lambda(w)$, whose ST is isomorphic to T_w, and whose labelling results from applying λ on l_w.

For linear time construction algorithms and many applications [9, 6, 8, 1, 3], internal nodes of STs are equipped with a *suffix link* defined as follows. We **extend this notion to leaves**; so, the following definition applies to all nodes of V_T except the root (instead of V_T^{in}). Let SL_w be the mapping from $V_{T_w} \setminus \{\perp_{T_w}\}$ to V_{T_w} that maps a node u to a node v iff $\text{suff}_2(\tilde{l_w}(u)) = \tilde{l_w}(v)$. We define the directed graph F_w such that $V_{F_w} := V_{T_w}$ and (u,v) belongs to E_{F_w} iff $SL_w(u) = v$. The edge (u,v) is the *suffix link* of u. It follows that F_w is a tree rooted in $\perp_{T_w}$. We call F_w the *complete suffixing tree* of ST_w.

Classically, SL equip only all internal nodes of a ST (except the root). By well-known property of SL [3], each SL points to another internal node, with those of the children of the root pointing to the root (which represents the empty string). Hence, the SL of internal nodes forms a tree rooted

in the root of ST_w. For $0 < i \leq |w|$, the SL of leaf i will point to leaf $(i+1)$ (for $\text{suff}_2(\text{suff}_i(w)) = \text{suff}_{(i+1)}(w)$). Hence, the SL of the leaves build a *chain* going from leaf 1 to leaf $|w|$ (which represents the suffix \$) and finally to the root of T_w. We termed the graph formed by F_w minus this chain, the *internal suffixing tree* of ST_w. We thus summarise this in Proposition 1.

Proposition 1 (Suffix links) *Let w be a string and ST_w denotes its suffix tree. Then*

1. *the suffix links of internal nodes form a tree rooted in the root of ST_w,*
2. *the suffix links of the leaves form a chain going through all suffixes in decreasing order of length, that is from the longest suffix of w until its root.*

Hence, F_w, the set of all suffix links, is a tree rooted in the root of ST_w.

As a corollary, only one leaf of ST_w, the one representing w itself, is a leaf in F_w. Note that the edges of F_w are directed towards the root of T.

2 Reverse engineering: internal labelling

Let us refine the problem SLI-REST. Let k be the cardinality of the alphabet Σ. Let (T,F) be an instance of SLI-REST. If any node in T has more than k children, then (T,F) cannot be the suffix tree of a word written on alphabet Σ. Otherwise, by property P4 we can arbitrarily choose the letters that belong to the alphabet and set $\Sigma := \{a_1, \ldots, a_k\}$. Once done, we can associate with each outgoing edge of $\perp_T$, that is with each of its children, distinct letters of Σ as the first letter of their label. By hypothesis, Σ contains enough letters and without loss of generality, we can assume that $\perp_T$ has exactly k children. Now the question becomes: Find a word on alphabet Σ such that its suffix tree and links are isomorphic to (T,F). For it is easy to choose a large enough k in linear time in the size of (T,F), solving this question allows us to solve SLI-REST. Figure 2 shows our running example with the input (T,F), that is the tree T in solid edges and the potential suffix links in dashed edges, and the arbitrary choice of first letters labelling the edges going out of the root.

Now, we define a first necessary condition of ST, called the *kinship condition* (Definition 1), and state an important result (Proposition 2) regarding the labels of all internal edges of T: If (T,F) satisfies the kinship condition, the labels of all internal edges are uniquely determined. We will see later how to compute them in linear time. Conversely, once the edges are labelled, one can find where the SL of each internal node should point to. We can formally define the tree formed by this set of SL (Definition 2). Clearly, any suffix tree satisfies the kinship condition.

Definition 1 (Kinship condition (Figure 3a)) *We say that (T,F) satisfies the* kinship condition *if and only if for any $v \in V_F \setminus (\{\perp_F\} \cup Children_T(\perp_T))$, one has $f_F(v) \in V_T(f_F(f_T(v)))$.*

Verifying the kinship condition means checking for each node v whether $f_F(f_T(v))$ is an ancestor of $f_F(v)$ in T. Let l be a labelling of T. In the same way as F_w is defined from T_w and l_w, we define a rooted tree F_l from T and l.

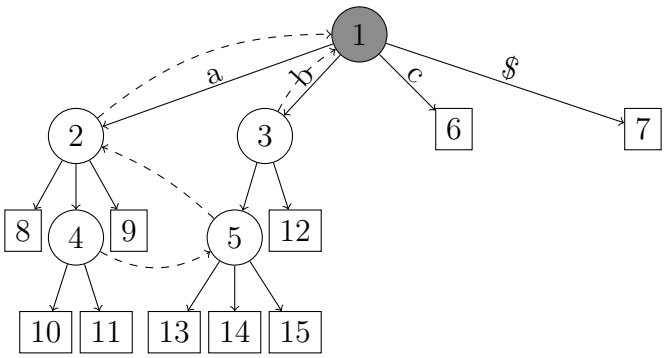


Figure 2: Running example: An input as a pair (T,F) with T a rooted tree (with solid edges) and F defined on the internal nodes of T (with dashed edges), which gives the potential suffix links of internal nodes. The same pair (T,F) is used a running example throughout this work. As one searches a word on an alphabet of size 3, one sets $\Sigma \cup \{\$\} = \{a,b,c,\$\}$ and as suggested in the text, one associates arbitrarily one letter to each edge linking the root of T to its children. These symbols are also referred to as the first letter of the branch since for each node in the subtree of that branch, the label in a suffix tree must start with this letter.

Definition 2 (F_l) *Let l be a labelling of T. F_l is a rooted tree whose set of vertices is V_T and for any two nodes u,v distinct from $\perp_T$, (u,v) is an edge of F_l if and only if $\text{suff}_2(\tilde{l}(u)) = \tilde{l}(v)$.*

Proposition 2 *Let F a potential, complete suffixing tree of T. Then (T,F) satisfies the kinship condition if and only if there exists a unique labelling l such that the tree of suffix links implied by l equals F (iff $F_l = F$).*

Throughout the paper, we denote by l_F the unique labelling as implied by Proposition 2.

Proof " $\Rightarrow$ " Suppose that (T,F) satisfies the kinship condition and let l be a labelling of T such that $F_l = F$. Among internal nodes minus the root, we consider two cases: either v is a child of the root or it is not. Let us show that for all $v \in V_F \setminus (\{\perp_F\} \cup \mathit{Children}_T(\perp_T))$,

$$l(v) = \tilde{l}(f_F(f_T(v)), f_F(v)).$$

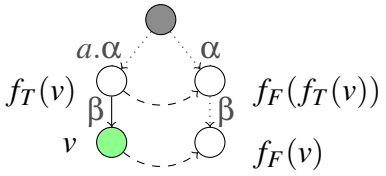
By the definitions of $\tilde{l}$ and F_l, we know that $\tilde{l}(v) = \tilde{l}(f_T(v))l(v)$ and $\text{suff}_2(\tilde{l}(f_T(v))) = \tilde{l}(f_F(f_T(v)))$. As $\tilde{l}(f_F(v)) = \tilde{l}(f_F(f_T(v)))\tilde{l}(f_F(f_T(v)), f_F(v))$ we obtain that $l(v) = \tilde{l}(f_F(f_T(v)), f_F(v))$.

Moreover, for any labelling l on T satisfying $F_l = F$, if the node v is a child of the $\perp_T$, there exists $i \in [1,k]$ such that $l(v) = a_i\tilde{l}(f_F(v))$. By construction, using a top-down traversal of F, we have a unique labelling l such that $F_l = F$.

" $\Leftarrow$ " Let l be the unique labelling such that $F_l = F$. By contraposition, assume that (T,F) do not satisfy the kinship condition. It means that there exists a node v such that $f_F(v) \notin V_T(f_F(f_T(v)))$. As $F_l = F$, $\tilde{l}(f_F(f_T(v)))$ is a prefix of $\tilde{l}(f_F(v))$, which leads to a contradiction and closes the proof. □

The proof of Proposition 2 contains a procedure to compute the complete labels of all internal edges of T; we call it the *internal labelling procedure*. As the kinship condition can be verified

(a) Kinship condition



(b) Separation condition (Def. 4 p. 8)

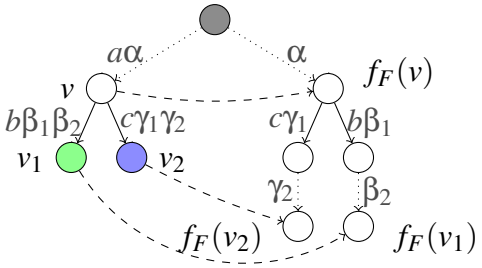


Figure 3: Illustration of two necessary conditions that nodes of a tree must satisfy in order to be a suffix tree. In (a) the kinship condition: the dashed edges from v (green) to $f_F(v)$ is required since $f_T(v)$ has a link to $f_F(f_T(v))$. In (b) the separation condition: the dashed edges going out of v_1 and v_2 must end up in different branches under $f_F(v)$.

in linear time, one can filter out any instance not satisfying it. On a remaining instance, we can determine the labels of all nodes having a potential SL, that is an edge of F starting from them. If F contains potential suffix links only for internal nodes, we must find the labels of edges going to the leaves and check whether these build a proper word. If F contains potential suffix links for all nodes (leaves included), we get all labels from Proposition 2, and we only need to check whether they form a word. We investigate this second case in the next section.

3 Recognising a ST with a complete suffixing tree

In this section, we assume that all nodes are equipped with a suffix link. In other words, in the input (T,F), we suppose F is a potential complete suffixing tree. We consider this case for two reasons. First, it is not trivial especially when the input is invalid, and we will demonstrate this can be solved efficiently by only looking at the structures of T and F without checking equality of the labels. Second, once one knows the chain of suffix links that lists all leaves of T (hence all putative suffixes), one can check the structure and then derive a word for (T,F) (see the algorithm below). Thus, to solve SLI-REST in the general case one needs to find this chain, and Section 4 is devoted to this question.

From Proposition 1, we know two structural properties that the set of suffix links must satisfy. This translates into a necessary condition on F. It is easy to see that checking the structure condition takes linear time in the size of (T,F).

Definition 3 (Structure condition) *We say that (T,F) satisfies the* structure *condition if and only if 1/ the set of internal nodes of T forms a subtree of F rooted in $\perp_T$, and 2/ the set of leaves T forms a subtree of F rooted in $\perp_T$ that is isomorphic to a chain.*

Algorithm for computing a word w realising (T,F) By hypothesis, we know the first symbol of the label for each child of the root of T (Section 1). For any node v (leaves included), this symbol is the first letter of the word represented by v. With a simple traversal of T, one can store in each node the corresponding first symbol of its branch. Since (T,F) satisfies the structure condition, we

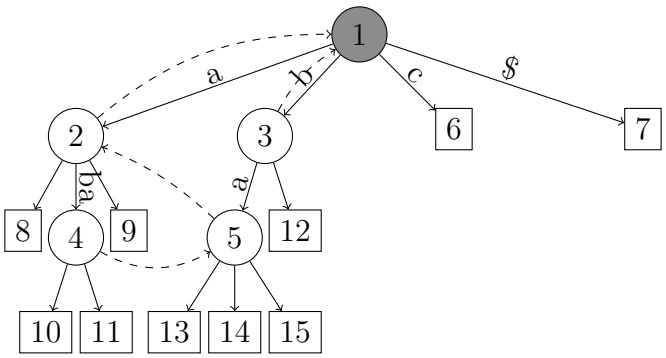


Figure 4: Running example: same tree with the labels on all edges going to internal nodes of T. The labels are computed for any node that has a dashed edge, (*i.e.* a potential suffix link) starting from it, as explained in the text. For instance, as the dashed edge link node 2 to node 1 one knows that the label between these nodes has length one, as one is the number of suffix links between the starting node and the root minus the same number for its father. Thus, this label equals a. The label between nodes 2 and 4 must be of length $3-1=2$. Moreover, it must be equal to the concatenation of labels between the nodes pointing to by the suffix links of 2 and of 4, that is equal to the labels read between the root and node 5. Thus the label between node 2 and 4 equals ba.

know that suffix links of the leaves form a chain that lists all suffixes in order of decreasing length. For any $1 \leq i < |w|$, going from leaf i to the next one (*i.e.* to leaf $(i+1)$) in this chain by following one suffix link, means going from the suffix starting at position i in w to the one starting at position $i+1$. The letter that is eliminated by doing so is $w[i]$ and is thus the first letter of the branch of leaf i. Hence, traversing the chain of leaves using their suffix links and printing at each step the first letter of the branch of the current leaf outputs w.

If (T,F) satisfies the structure condition, then the only leaf of T that is a leaf of F, represents the longest suffix, *i.e.*, a candidate word that realises (T,F). One could compute this word, call it z, as explained above, build its suffix tree with suffix links and labels, check whether these are isomorphic with (T,F), and verify the equality between the labels. However, building all labels and checking their equality for all edges can take quadratic time in the length of z. We want to avoid this procedure. For this sake, we introduce another necessary condition (Definition 4) and show that if (T,F) satisfies both the structure and separation conditions, then it forms a valid suffix tree (Theorem 6). One can check these in linear time (Proposition 7) and compute a word realising it also in linear time.

Definition 4 (Separation condition (Figure 3b)) *We say that (T,F) satisfies the* separation *condition if and only if for all $v \in V_F^{in} \setminus \{\perp_F\}$, for all $v_1, v_2 \in Children_T(v)$ with $v_1 \neq v_2$, $f_F(v_1)$ and $f_F(v_2)$ belong to distinct branches of the T-subtree rooted in $f_F(v)$.*

One sees quite easily that the separation condition implies the kinship condition.

Proposition 3 *If (T,F) satisfies the separation condition, it also satisfies the kinship condition.*

The next proposition means that (T,F) labelled with l_F satisfies condition C2.

Proposition 4 *If (T,F) satisfies the separation condition, then for all $v \in V_F^{in}$, for all $v_1, v_2 \in Children_T(v)$:*

$$v_1 \neq v_2 \;\Rightarrow\; l_F(v_1)[1] \neq l_F(v_2)[1].$$

Lemma 5 (Relation between conditions) *(T,F) satisfies the separation condition if and only if it satisfies the kinship condition and if the labelling l_F satisfies condition C2 on (T,F).*

Proof The propositions 4 and 3 yield the result in one direction. For the other direction, let u, v, w be nodes of T such that v, w are children of u and $u \neq \bot_T$. By hypothesis, l_F satisfies condition C2, hence: $l_F(v)[1] \neq l_F(w)[1]$. By the kinship condition, we have for both v and w that $\tilde{l_F}(f_F(u), f_F(v)) = l_F(v)$ and $\tilde{l_F}(f_F(u), f_F(w)) = l_F(w)$. From which we deduce that the labels $\tilde{l_F}(f_F(u), f_F(v))$ and $\tilde{l_F}(f_F(u), f_F(w))$ start with distinct first symbols, and hence, the nodes $f_F(v)$ and $f_F(w)$ lie on different branches below $f_F(u)$. □

Theorem 6 *(T,F) is realised if and only if (T,F) satisfies both the separation and the structure conditions.*

Proof " $\Rightarrow$ " From the properties of Suffix Trees, we get that (T,F) is realised, then it satisfies the separation and the structure conditions.
" $\Leftarrow$ " Assume (T,F) satisfies both the separation and the structure conditions. By Proposition 3, there exists a unique labelling l_F such that $F_{l_F} = F$. To prove the realisation of (T,F), we must find a string w satisfying the four conditions of a suffix tree characterisation (see C1-4 on p. 4). C1 is satisfied by hypothesis since we filtered out any input tree T violating it. One deduces C2 from Proposition 4 since (T,F) satisfies the separation condition.

From the structure condition, V_T^{leaf} forms a subtree of F rooted in $\bot_T$ that is isomorphic to a chain graph. The leaf at the extremity of that chain, denote it v, is the only one in $V_T^{leaf} \cap V_F^{leaf}$. If we set $w := \tilde{l_F}(v)$, thanks to the chain, we can map bijectively each suffix of w to a leaf in T, which gives condition C3. By construction, for any internal node u, there exists at least a leaf $v \in V_T(u)$. Hence, by the kinship condition, $\tilde{l_F}(u)$ is a prefix of $\tilde{l_F}(v)$, which is a suffix of w. Thus, $\tilde{l_F}(u)$ is a substring of w, which gives us condition C4 and concludes the proof. □

Proposition 7 *Verifying that (T,F) satisfies the separation condition takes linear time in $\#(V_T^{leaf})$.*

Proof To check that (T,F) satisfies the separation condition, we use Algorithm 1, which traverses a subtree of F in a depth-first search manner. Let v be an internal node of T and $v_1, v_2, \ldots, v_k$ be all its children in T. Let w (respectively $w_1, w_2, \ldots, w_k$) be the father of v (respectively of $v_1, v_2, \ldots, v_k$) in F. We need to show that for any $1 \leq i < j \leq k$, w_i and w_j belong to distinct branches of the subtree rooted in w, *i.e.* that their lowest common ancestor is w.

If the algorithm returns False at iteration i, then two cases arise. Either z is the root of T (line 7), which means that z is not in the subtree of w and thus (T,F) violates the separation condition.

Algorithm 1: Separation$(w, w_1, w_2, \ldots, w_k)$.

Input : A set of nodes $w, w_1, \ldots, w_k$ coloured white such that $Children_T(w) = \{w_1, \ldots, w_k\}$
Output: Returns True if for any $1 \leq i < j \leq k$ the lowest common ancestor in F of w_i et w_j is w and False otherwise.

```
1 for i ← 1 to k do
2     z ← w_i;
3     while z ≠ w do
4         if z is black then return False;
5         else colour z black;
6         z ← f_T(z);
7         if z is the root of T then return False;
8     return True;
```

In the alternative case, z is already black (line 4), which occurs when z has already been visited at an earlier iteration, say $j < i$, with child w_j. Then z is the lowest common ancestor of w_i and w_j. As $z \neq w$ (for we are in the while loop - see line 3), (T, F) violates the separation condition. It follows that the algorithm returns True if for all $1 \leq i < j \leq k$, w is the lowest common ancestor of w_i and w_j.

For each internal node v of T, we call Algorithm 1 with $\{f_F(v)\} \cup \{\bigcup_{v_i \in Children_T(v)} f_F(v_i)\}$. We get that (T, F) satisfies the separation condition if and only if it returns True for all nodes of V_T^{in}. As soon as Algorithm 1 returns False for some node, we stop the overall verification and return False. Note that if we proceed branch by branch and make the calls successively for the internal nodes of a given branch, we ensure that all tested nodes locally satisfy the kinship condition (Def. 1). It follows that amortised complexity of the whole verification is linear in the size of T. □

Thanks to Propositions 2, 7, and to Theorem 6, we can both decide if (T, F) is realised, find the labelling l_F, and compute a string w realising (T, F) in linear time.

An example of an invalid input satisfying the structure and kinship condition Figure 5 displays a non trivial example to show that checking the separation condition is compulsory to discriminate valid from invalid inputs. The input tree and the first letter of the edges' label violate the separation condition.

4 Recognising a ST with an internal suffixing tree

From Section 3, we know how to solve SLI-REST in linear time when all nodes are equipped with a potential suffix link (*i.e.* have an out-going edge of F starting from it). However, in the general case, only internal nodes have potential suffix links. In other words, the input F is a tree on the internal nodes of T (if it has not been filtered at an earlier step because it violated simple necessary conditions). In this section, we will see how to extend F such that each leaf also has a potential

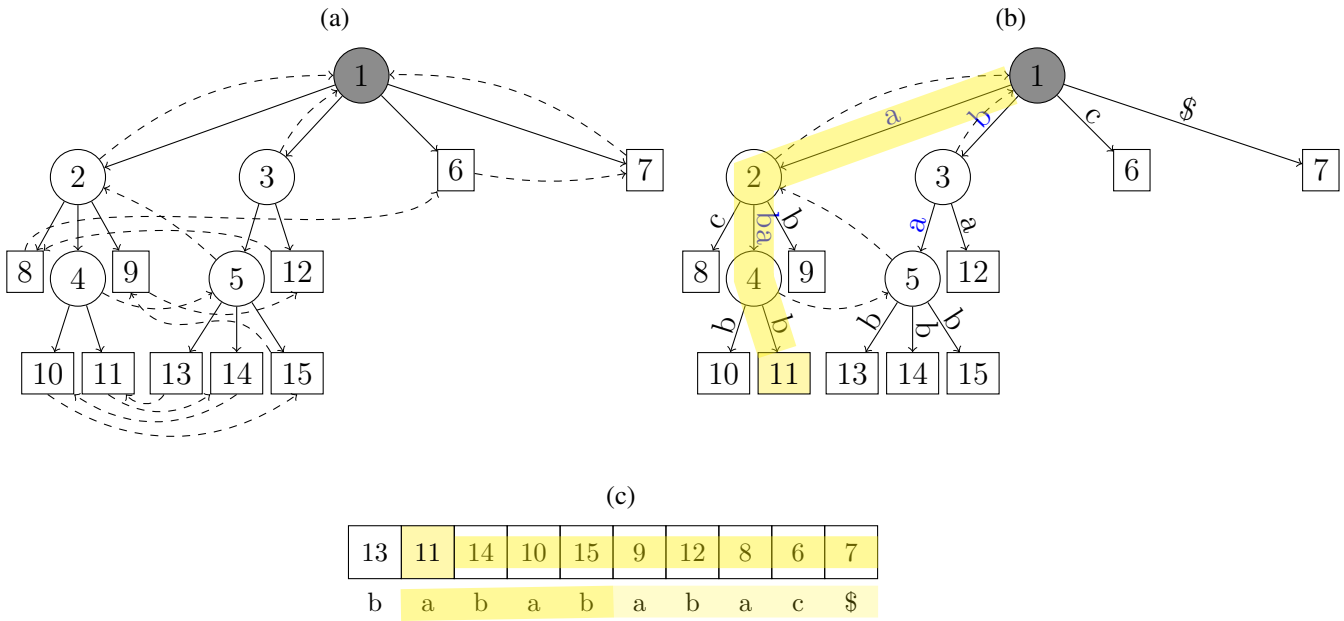


Figure 5: An invalid input (T, F) satisfying the kinship and structure conditions, but not the separation condition. T is identical to the tree in our running example, but F is not. (a) the input tree T and its complete suffixing tree F; (b) T with the labels of internal branches, the edges of F only for internal nodes, and the first symbols of the labels of edges going to leaves. (c) The word obtained with the algorithm described on p.7, with in yellow the substring corresponding to the word represented by leaf 11. Obviously, the labelling of the leaves does not comply with condition C2 of suffix trees.

suffix link and those links form a chain. This is equivalent to finding a permutation of the leaves. The description is divided into three parts:

1. how to reduce the combinatorics of possible solutions? We explain why one can explore solutions by considering only *multi-permutations* of a subset of internal nodes,

2. defining a graph on this subset of nodes,

3. defining the properties that a route which traverses this subset of nodes shall satisfy such that the (T, F) can be realised.

Then, we will conclude by giving an algorithm that explores all potential valid routes and outputs all distinct words realising (T, F).

4.1 From a chain of leaves to a multi-permutation of some internal nodes

We seek how to extend F by adding potential suffix links to the leaves. In Section 2, we showed that a complete suffixing tree must satisfy the separation condition. One can see that from its definition (Def. 4), this necessary condition must be inherited by the subtree of internal nodes of

a complete suffixing tree. Thus, we deduce that the input F (of this section) must also satisfy the separation condition.

Now, the Structure condition (Def. 3) requires that the potential suffix links of the leaves form a chain on all leaves ending at the root of T, and such a chain is equivalent to a permutation of the leaves. Clearly, one cannot explore the complete set of permutations in polynomial time; thus we need to reduce the search space. First, remember that strings are ended by a terminating symbol occurring only once, the dollar. From property P2 (see p. 2), $\perp_T$ has a chleaf labelled by this symbol, and it must represent the suffix of length 1 of a realising string. Hence, this leaf must be the one that precedes the root in the chain. So now we seek a permutation of the set of leaves minus that leaf of the root. From now on we only consider this subset of the leaves. We get that the set of extensions of F such that (T,F') satisfies the structure condition (where F' denotes an extension of F) is in one-to-one correspondence with the set of permutations of the leaves. Another property helps us to restrict the combinatorics: in a suffix tree, if two leaves have the same father, then exchanging their edge labels does not change the word, but only the starting position of the corresponding suffixes (see Figure 6 on page 13). In other words, if (T,F') is realised by a word w, then for any extension F'' obtained by exchanging two such leaves in the chain, we have that (T,F'') is also realised by w, and conversely. This is an important property for, from now on, we can "replace" in the permutation chleaves of the same node by their father. For simplicity, we insert a dummy internal father node between any chleaf of $\perp_T$ and $\perp_T$ (Otherwise, all these chleaves would be replaced by the root and we would lose the first letter of their labels; moreover, with this transformation, the root has no chleaf anymore - see below). We obtain the following property (see also Figure 6).

Proposition 8 *Let σ be a leaf permutation such that (T,F) is realised by a word w, and u,v be two leaves sharing the same father node. The permutation σ' in which u and v have been swapped is also realised by w.*

Let us call V_c the subset of internal nodes of T that have at least one chleaf. For any $v \in V_c$, let us denote by $n(v)$ its number of chleaves. Hence, what we seek is a *multi-permutation* of the nodes of V_c, where each element of V_c appears exactly $n(v) \geq 1$ times (each occurrence represents one of its chleaves in the permutation of the leaves). We coined the term *multi-permutation* as an equivalent of the term multi-set, but for a permutation. To transform a leaf permutation into a multi-permutation of V_c one replaces each leaf by its father; hence, many leaf permutations map to the same multi-permutation of V_c. Conversely, from a multi-permutation of V_c, one creates a leaf permutation by replacing each node of V_c by one of its chleaves. We term this an *induced leaf permutation* (see Figure 6). Thus, all possible relative orderings of the chleaves of a node exist in distinct induced leaf permutations. Note that if two internal nodes are consecutive in a multi-permutation, then two among their chleaves must be consecutive in an induced leaf permutation (see Figures 6 and 7).

Proposition 9 *Let σ_i be a multi-permutation of V_c and σ_f be an* induced leaf permutation *of σ_i. Let v_1, v_2 be two consecutive nodes of σ_i. Then, the edge of F starting in v_1 (*i.e. *the potential suffix link of v_1) points to v_2 or to an ancestor of v_2 in T if and only if the extension made of F union σ_f satisfies the kinship condition.*

Proof Assume (T,F) satisfies the separation condition; by Proposition 3 it satisfies the kinship condition. As for any leaf u, its link in σ_f points in the subtree of the node pointed to by the link of its father, one gets that $(T,F\cup\sigma_f)$ also satisfies the kinship condition. The proof in the other direction follows from properties of suffix trees. □

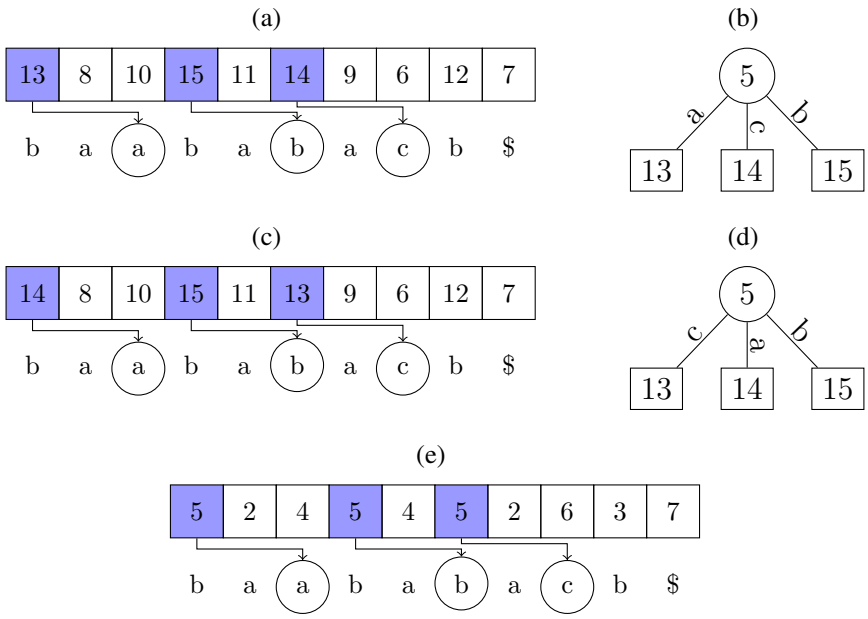


Figure 6: Running example: different leaf permutations that exchange the order of chleaves of the same node (here chleaves $13,14,15$ of node 5 - see (a), (c)) are equivalent to exchanging the labels of the edges leading to these chleaves (see (b), (d)). Hence, using a multi-permutation of nodes of V_c (see (e)), as a representative of all its induced leaf permutations simplifies the algorithm. As node 5 represents the word ba, the label corresponding to each chleaf is located two positions after the chleaf number in table (a-c). See the node numbering on Figure 4.

We call an *acceptable multi-permutation* (AMP) any multi-permutation of V_c verifying the condition of Proposition 9 and such that the last node is a dummy child of $\perp_T$. An extension of an AMP is the extension of F obtained by taking the union of F with an induced leaf permutation of this AMP. Two remarks regarding the extensions of an AMP. First, by the condition of Proposition 9, any extension satisfies the kinship condition. Second, as an induced permutation of the leaves is a chain of the leaves, the extension also satisfies the structure condition.

We say that a multi-permutation is *realised* if the extension of F built by an induced leaf permutation is realised. We can deduce the next proposition, which allows to reduce drastically the elements considered during the exploration.

Proposition 10 *Let σ_i be a multi-permutation of V_c and let w a word. If σ_i is realised by w, then any* induced leaf permutation *is realised by w.*

4.2 The Graph of Internal Nodes (GIN)

We are now ready to define the Graph of Internal Nodes (GIN), in which a route will be an acceptable multi-permutation. The GIN is not a simple graph: several identical edges (in other words, a multiedge) can link two vertices. Edges of the GIN are partitioned in two sets, denoted R and B (Red and Blue), and thus the GIN is said to be bi-coloured. An edge of R is an edge of F: it links u and the node of T pointed by its potential suffix link (*i.e.*, $f_F(u)$) with a multiplicity of $n(u)$ (the number of chleaves of node u). Now for a node v of T, we define $c(v)$ as the number of edges in R leaving the subtree of T rooted in v minus those pointing into this subtree. Because (T,F) satisfies the kinship condition, $c(v) \geq 0$ for any node v otherwise (T,F) cannot be a positive instance. It is easy to filter out such instances. An edge of B is an edge of T linking the father of v to v with multiplicity $c(v)$. The notation $(u,v)^j$ is a shortcut for j identical edges (u,v).

Definition 5 (GIN, Figure 8) *The* Graph of Internal Nodes (GIN) *of* (T,F)*, denoted* $GIN(T,F) := (V,R,B)$ *is a bi-coloured directed graph such that*

$$V = V_T \setminus U$$

$$R = \{(u,v)^{n(u)} |\ f_F(u) = v\}$$

$$B = \{(u,v)^{c(v)} |\ f_T(v) = u\}$$

where $(u,v)^j$ *denotes a* j*-multiedge from* u *to* v *and*

$$U := \{v \in V_T^{in} \setminus \{\bot_T\} | v \textit{ has no edge neither in } R \text{ nor } B\}.$$

We will look for routes that visit all edges of the GIN. By the definitions of R and B, one gets that the numbers of in-going and out-going edges of any node $v \in V$ are equal. The GIN is Eulerian only if it is connected. In this case, we can search for Eulerian routes through the GIN that end by an edge between a dummy child of $\bot_T$ and $\bot_T$. We call such a route a *primary route*. The next lemma shows that there exists a bijection between the sets of AMP and of primary routes of the GIN.

Lemma 11 *An acceptable multi-permutation of* V_c *is bijectively associated to a primary route of the GIN.*

Proof Let p be a primary route of the GIN. We define the multi-permutation of V_c associated with p as follows: the multi-permutation, denoted M_p, keeps all starting nodes of each edge of R in their order of appearance in p. Let u,v be two consecutive nodes in M_p. Since u and v are adjacent in M_p, there exists only one edge of R between them in p. Thus, the remaining edges of the path between u and v belong to B. Then, the outgoing edge of F starting in u points to an ancestor of v in T. Moreover, as p ends on $\bot_T$, its last edge necessarily belongs to R, since only such edges can go up in T. We obtain that M_p is an acceptable multi-permutation.

Now, let M be an AMP on V_c and let u,v be two consecutive nodes in M. As M is acceptable, there exists a path between u and v in the GIN that starts with an edge of R and continues with

edges of B. By definition of the GIN, this path is unique. Thus, by choosing this path for any two consecutive nodes u, v of M, we build a unique route associated with M. Recall that in an AMP, each node v occurs exactly $n(v)$ times, with $n(v)$ being both its number of chleaves and its number of outgoing edges that belong to R. As the AMP satisfies Proposition 9, the definition of $c(v)$ implies that there exists a route traversing each edge of R exactly once. Assume that some edges of B are not visited by this route, as the GIN is Eulerian, then these edges form a set of cycles. However, this contradicts the fact edges in B belongs to the tree T. Hence, this route traverses all edges of the GIN exactly once. Moreover, since M ends on a child of $\perp_T$, it is primary, which concludes the proof. □

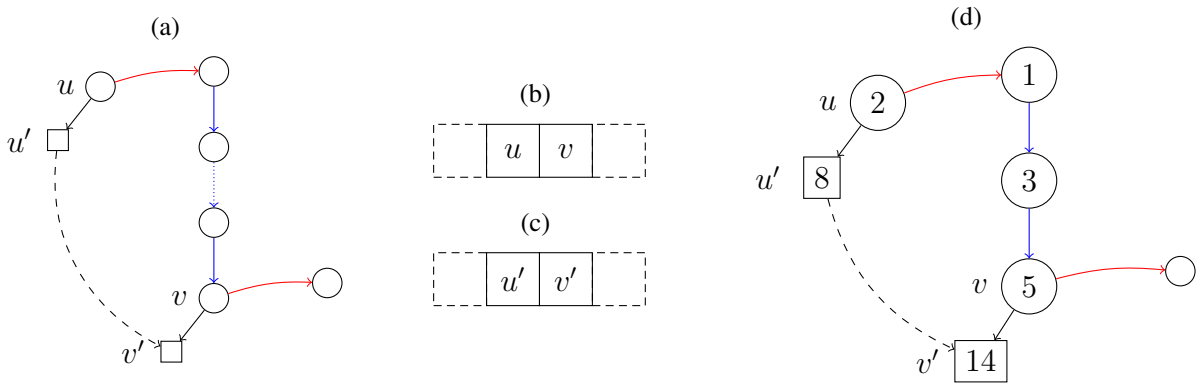


Figure 7: Link between primary routes, multi-permutations on V_c, and leaf permutations. (a) Window over a primary route of the GIN showing the path between two nodes u, v of V_c, which are by definition followed by an edge of R (in red). All other edges between u and v belong to B (in blue). Thus, v follows u in the corresponding acceptable multi-permutation of internal nodes (shown in (b)), and then u', a chleaf of u, follows v', a chleaf of v, in an induced leaf permutation (shown in (c)). Figure (d) shows an example of such path between internal nodes 2 and 5 of our running example, which will put leaf 14 after leaf 8 in the permutation.

As a consequence of Lemma 11, if the GIN is disconnected, no primary route exists and thus no acceptable multi-permutation of V_c. From now, we assume that instances with disconnected GIN are tested and discarded (in linear time in the size of (T, F)). For finding efficiently primary routes, we create bi-coloured labels on the edges of the GIN.

4.3 Labelling the GIN

The aim of the labels is two-fold: first, they will give us a word realising (T, F) if the instance is positive, second they allow us to show that the GIN is linear in the size of a solution. We create distinct labels l^R and l^B for the edges of R and of B, respectively. Basically, edges of R are the potential suffix links, and as SL link one suffix to the next, traversing one is equivalent to loosing the first symbol of the suffix of which the starting node is a prefix. Hence, we set the label of

an edge (u,v) to be the first symbol of the string represented by u (*i.e.* $\tilde{l_F}(u)[1]$). Edges of B are internal edges of T (those not ending at a leaf) and all these are labelled in T (see p. 6). Hence, l^B keeps the same label for the corresponding edge in the GIN. Formal definitions of l^R and l^B follow and are illustrated on Figures 8 and 9.

$$l^R(u,v) := \begin{cases} \tilde{l_F}(u)[1] & \text{if } (u,v) \in R \\ \varepsilon & \text{if } (u,v) \in B \end{cases} \quad (1) \qquad l^B(u,v) := \begin{cases} \varepsilon & \text{if } (u,v) \in R \\ l_F(u,v) & \text{if } (u,v) \in B. \end{cases} \quad (2)$$

We extend the labelling l^R and l^B to any route in the GIN, and define it be the concatenation of the labels of its edges (Figure 8). By the properties of ST and of SL, we obtain for any primary route p that $l^R(p) = l^B(p)$. If (T,F) is realised, then $l^R(p)$ will be a word realising it. From the equality $l^R(p) = l^B(p)$ it follows that the cardinality of B is at most that of R, and that $GIN(T,F)$ has a size linear in that of T.

Let us give the sketch of a proof for the equality $l^R(p) = l^B(p)$. For any node u, $\tilde{l}(u)$ is a prefix of $l^R(p[u..])$ and a suffix of $l^B(p[..u])$, where $p[u..]$ denotes any subroute of p starting at node u, and $p[..u]$ any subroute ending at node u. Let v be the last node of p before u having an outgoing edge that belongs to B. Then, we have $\left|l^B(p[v..u])\right| \leq \left|\tilde{l}(u)\right|$, where $p[v..u]$ is the subroute of p between v and u. Let w be the first node of p after u having an ingoing edge that belongs to R. Then, $\left|l^R(p[u..w])\right| \leq \left|\tilde{l}(u)\right|$. Now recall that B-edges go down the tree, while R-edges can only go up T till the root. As these inequalities are true for any node u, combining them with the previous properties shows the equality.

A primary route gives us an acceptable multi-permutation of V_c, which yields a permutation of the leaves. But we still need to find the first symbol of the label in T of each edge going to a leaf such that the separation condition is satisfied. Let us explain how this is done for a primary route p. While traversing p, we can keep for each node the record of the first symbols already associated with its children (see below the definitions of $i^0(v,u)$ and $i_p(e_i)$) and set those for its chleaves. During a traversal, we will collect in the set $T(p)$ the edges for which we are bound to choose a symbol that belongs to the edge's set of forbidden letters (because there are not enough letters left in the alphabet); see Figure 9. In other words, $T(p)$ collects the violations to the separation condition. Hence, we will see later that a necessary and sufficient condition on p such that (T,F) is realised, is: $T(p) = \emptyset$.

Figure 7 illustrates the relationship between a primary route and an acceptable multi-permutation of V_c. In a primary route p, the nodes of V_c are directly followed by an edge of R (coloured in red). Hence, all edges but one on the path linking two consecutive nodes, u, v, of V_c belong to B (in blue). By construction, this path is unique because of the kinship condition. Finally, if v follows u in the route, then a chleaf of v will follow a chleaf of u in the associated induced leaf permutation.

For any primary route p, we define $T(p) := \{e_i \in B \mid e_{i-1} \in R \text{ and } l^B(e_i)[1] \in i_p(e_{i-1})\}$, where for each $e_i = (v,u)$ in R,

$$i^0(v,u) := \cup_{w \in Chin_T(v)} \{\, l_F(v,w)[1] \,\} \qquad \text{initialise the set of forbidden symbols}$$

and, if for any $e \in R$, we denote the first B-edge coming after e in p by $\text{next}_p(e)$,

$$i_p(e_i) := i^0(e_i) \quad \bigcup \quad \{\, l^B(\text{next}_p(e_j))[1] \mid e_j \in R,\ e_j = (v,u),\ \textit{and}\ j < i \,\}.$$

Examples of i_p are shown on Figure 9. The goal of finding a primary route without violations is to assign distinct first symbols to the edges of any pair of sister leaves.

Theorem 12 *(T,F) is realised if and only if there exists a primary route p of $GIN(T,F)$ satisfying $T(p) = \emptyset$.*

Proof We know that (T,F) is realised if and only if there exists an AMP having an extension F' of F such that the labelling $l_{F'}$ satisfies condition C2. In other words, such that the labelling implied by F' assigns distinct first symbols to the edges of any pair of sister leaves. We obtain that a realised AMP is one satisfying the previous condition. Indeed, (T,F) is realised means that there exists an extension F' of F satisfying both the structure and separation conditions (by Theorem 6). By Lemma 5, F' satisfies the structure and kinship conditions, and $l_{F'}$ satisfies condition C2. This is equivalent to the existence of an AMP having an extension F' of F such that the labelling $l_{F'}$ satisfies condition C2.

Thanks to Lemma 11 and to the definition of $T(p)$, which collects the edges violating condition C2 in the primary route p, there exists an isomorphism between the set of realised AMPs and the set of primary routes p of the GIN such that $T(p) = \emptyset$. This concludes the proof. □

4.4 An algorithm for finding all words realising (T,F)

Algorithm 2: `Explore-GIN`: Explore exhaustively all primary routes of the GIN

Input:$GIN(T,F) = (V,R,B)$ **Output**: print all words realising (T,F)

`// for each edge` (u,v), `initialise a counter` $q(u,v)$ `to its multiplicity`

for $(u,v) \in B$ **do** $q(u,v) \leftarrow c(v)$;

for $(u,v) \in R$ **do** $q(u,v) \leftarrow n(u)$;

for $v \in V$ **do** `// initialise their sets of forbidden symbols`

  $i_p(v) \leftarrow \emptyset$;

  for $w \in Chin_T(v)$ **do** $i_p(v) \leftarrow i_p(v) \cup \{l_F(v,w)[1]\}$;

return `Explore-GIN-rec(`$GIN(T,F)$, $\bot_T$, $\bot_T$, ε`);` `// explore from the root with` $y := \varepsilon$

Building on Theorem 12, we exhibit an algorithm that decides whether (T,F) is realised or not, and finds all distinct words realising it by exploring all primary routes of $GIN(T,F)$. More exactly, the algorithm produce all words up to a permutation of the alphabet. Indeed, as the labels of the GIN depends on an arbitrary alphabetisation of T (see p. 3), additional words realising (T,F) can be obtained by permuting the letters in the output words. For any route p satisfying the conditions of Theorem 12, the corresponding word y is given by the concatenation of the edges of

Procedure Explore-GIN-rec: recursive traversal of all primary routes of the GIN

Input: $GIN(T,F) = (V,R,B)$, two nodes v and u, y the word being computed

```
1  if v = ⊥_T and |y| = #V_T^leaf then print y;
2  else
3      for w a node of B ∪ R such that q(v,w) > 0 do
4          q(v,w) ← q(v,w) − 1 ;                          // decrease its counter
5          if (v,w) ∈ B then
6              if v = u then //no interdictions, update y, explore from w on
7                  Explore-GIN-rec(GIN(T,F),w,w,y.l^B(v,w));      // add l^B(v,w) to y
8              else
9                  x ← u;
10                 while r(x) ≠ v do x ← r(x); // move x till the node preceding v
11                 if l^B(v,w)[1] ∉ i_p(x) then
12                     while r(u) ≠ v do // update u to the node preceding v
13                         i_p(u) ← i_p(u) ∪ l^B(v,w)[1];
14                         u ← r(u);
                       // update y as above, explore from w on
15                     Explore-GIN-rec(GIN(T,F),w,w,y.l^B(v,w));
16         else // then (v,w) ∈ R:  nothing to check; keep u and y; explore from w on
17             Explore-GIN-rec(GIN(T,F),w,u,y);
```

B traversed along p, *i.e.* $y = l^B(p)$. Algorithm 2 (`Explore-GIN`) explores all primary routes of the GIN whose associated word realises the input (T,F) (but not only these routes, see below), and prints those words. It calls the procedure `Explore-GIN-rec` (p. 18), which performs a recursive depth first exploration. The stop conditions of line 1 ensure that i/ the route is primary, ii/ all edges have been visited (since we have enough letters in the output word y - see the proof of $l^R(p) = l^B(p)$). Along the current route, it computes a word y by concatenating the labels of the B-edges it traverses. Moreover, it builds on-the-fly the set i_p of forbidden symbols for each B-edge, and when it traverses an edge, if the label violates the set of interdictions, it stops exploring the current route. Then, the recursive procedure resumes with the next possible route. That way, all the portions shared between several routes are explored only once (see Figure 11(c) on p.23). The complexity of Algorithm 2 is bounded by the exponential number of Eulerian cycles in a graph.

By simplicity, we use a notation for denoting the node pointed by a potential suffix link. Let $v \in V$, there exists a unique node $u \in V$ such that $(v,u) \in R$: we set $r(v) := u$. The algorithm maintains the current node, denoted by v, and the last visited node u that is the end-point of a B-edge. So, u precedes v on the route and only R-edges were traversed between them. To be sure that each edge (u,v) of the GIN is visited once, the algorithm maintains a counter $q(u,v)$, which is initialised with the edge's multiplicity.

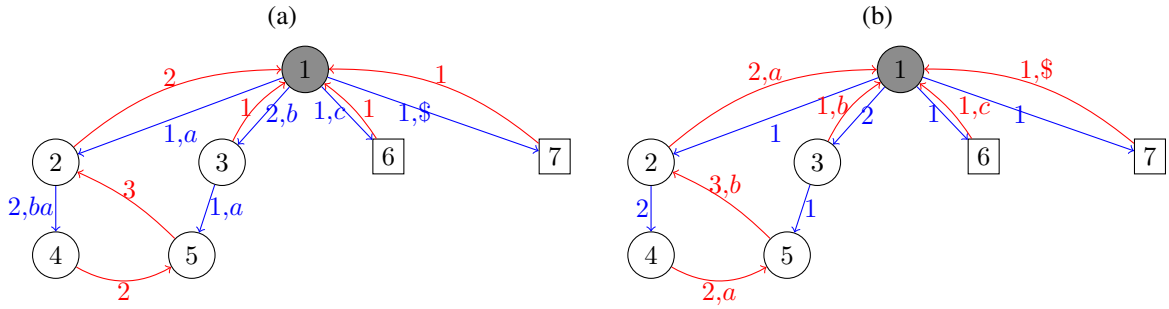


Figure 8: Running example: The bi-coloured GIN with labels on edges of B in blue (a), or on edges of R in red (b) as defined in Section 4.3. The number near each multiedge gives its multiplicity. The label of an edge in R corresponds to the first letter of the branch from the starting node: the label between $(4,5)$ is a because the branch of 4 is labelled by a between the root and node 2. This is the letter "lost" by traversing the suffix link starting in 4. The label of an edge in B is the label of the corresponding internal branch of T as computed using the links of F (see Section 2).

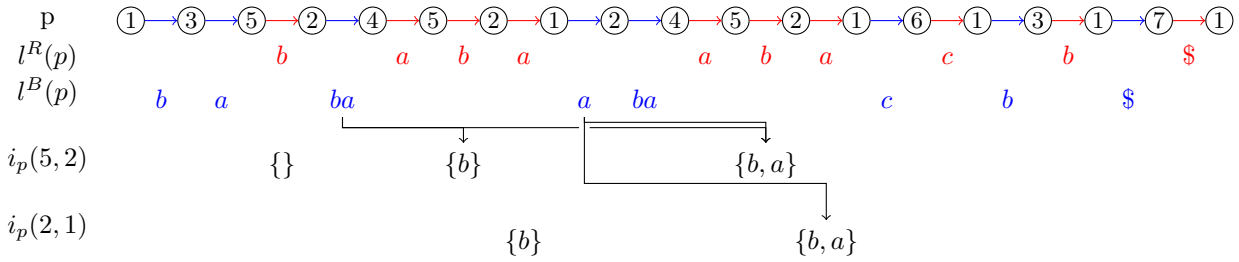


Figure 9: Running example. p is a primary route in the GIN displayed in Figure 8. Below, one sees that its two labellings $l^R(p)$ and $l^B(p)$ are equal ($l^R(p) = l^B(p)$). The fourth line shows how the set of prohibited symbols $i_p(5,2)$ evolves along p for edge $(5,2)$. Initially, on the first occurrence of $(5,2)$: no letters are forbidden for $i^0(5,2)$ is empty. Then, a blue edge labelled ba is traversed before reaching the second edge $(5,2)$, thus at this occurrence, the symbol b is added to $i_p(5,2)$. Then, the blue edge $(1,2)$ labelled with a is traversed, and finally this symbol is added to $i_p(5,2)$ at the third occurrence of $(5,2)$. The last line shows the evolution of $i_p(2,1)$, which is initially set to $i^0(2,1) = \{b\}$, for one child of node 2 is labelled with ba. Later, the symbol a is added to $i_p(2,1)$ because edge $(1,2)$ has been traversed.

5 Differences between Suffix Tour Graph and GIN

This section emphasises and illustrates the differences between the approach of [5] and ours. For instance, we exhibit an instance on a 4-letter alphabet on which their approach runs an exponential number of times their linear-time algorithm for distinct labelling functions, while ours is run once and takes linear time in the size of (T,F). First, we propose an informal summary of our approach and another view of the one of [5]. Last, we review important differences between the two approaches.

5.1 An informal summary

Consider an instance (T,F) of the problem SLI-REST.

Our approach First, we check that (T,F) satisfies the separation condition in linear time (Algorithm 1, Prop. 7). Second, we build the GIN and check its connectivity. Then, we seek F', an extension of F, a potential suffix link is assigned to each leaf. We know that a primary route p implies an extension (T,F') satisfying the structure and kinship conditions. By Theorem 6, we know that (T,F) is realised if (T,F') also satisfies the separation condition. The latter is true if the route p has no violations, i.e. if $T(p)$ is empty. Our traversal algorithm computes $T(p)$ dynamically while traversing p in time and space that are linear in the length of p.

A reformulation of I et al's approach In addition to (T,F), they also consider an order and a labelling function g on T. This function associates to each edge a single letter of the alphabet as the first letter of the edge's label. Given this input, they seek a word that realises (T,F) and is compatible with g; in this case, let us say that (T,F,g) is realised. They filter out instances on natural preconditions and build the unique Suffix Tour Graph, STG_g. These conditions ensure that any cover by cycles of the STG_g implies an extension F' such that (T,F') satisfies the separation condition. Indeed, the authors show that in the STG_g, the in- and out-degree of each node are equal; thus, each connected component of the STG_g is Eulerian. Consequently, it remains to show that (T,F') satisfies the structure condition. Actually, I et al. show that if the cycle cover is made of a single Eulerian cycle then (T,F') satisfies the structure condition. This means that (T,F,g) is realised if and only if there exists a Eulerian cycle on the STG_g, i.e. if and only if STG_g is Eulerian. The disadvantage is that checking whether (T,F) is realised requires to consider all possible orders and labelling functions.

Theoretical insights First, our work shows that the order on T does not ease the search (see Proposition 8). Second, Theorem 12 is the corner stone of the approach since it provides a characterisation of the realisation based on two conditions that are simple to verify. Third, it is possible to check on-the-fly, while traversing a primary route, whether it generates some violations when it assigns labels to the edges of chleaves. Finally, I et al's approach explores the set of STG_g for all possible labelling functions g, while ours explores all potential primary routes of a single graph

(the GIN). It remains open to determine which from both search spaces is the less complex from a combinatorial viewpoint.

Figure 11 allows to compare both approaches on the example taken from [5, Figure 8]. All labelling functions and the corresponding STG are shown in [5], while Figure 11 shows primary routes in the GIN. One sees that neither approach subsumes the other. Indeed, on one hand, an interrupted route in the GIN corresponds to an invalid labelling function in their approach. On the other, a labelling function leading to a disconnected STG (their first case), cannot be a primary route in the GIN; thus the GIN avoids such cases. Our results suggest an improvement for I *et al.*'s approach: (1) use the internal labelling procedure (see p. 6) to compute the labels of internal edges, (2) limit the search space of labelling functions to those labelling only the leaf edges.

5.2 Combinatorial differences

Recall that the Suffix Tour Graph of I et al is built for an input (T,F), and importantly for a chosen *order* among the children of a node and a chosen *labelling function*. To choose a labelling function, I et al. also require an order on the children of any node, that is an order on their labels (*e.g.*. the first child label starts with an a, the second with a b, etc.) For a node of T having d children, restricting to labelling functions in agreement with the chosen order divides the number of labelling functions by a factor of $d!$ ($C_k^d = \frac{k!}{d!(k-d)!}$ instead of $A_k^d = \frac{k!}{(k-d)!}$, where k is the alphabet's size). However, enforcing an order can lead to errors. Indeed by Proposition 2, the labelling of internal nodes is completely determined. So, if unfortunately the labelling does not comply with the order, their algorithm returns false even if (T,F) is a positive instance of SLI-REST. Then other orders have to be considered, and for them several labelling functions.

For instances whose GIN is disconnected, we can verify in linear time whether (T,F) is realised or not. For the same case, the Suffix Tour Graph is built and tested for all possible labelling functions. Figure 10(a) shows an example (not the running one) of such case: the left branch is divided into l levels of two internal nodes each (here $l = 2$) and a complete subtree with 4 leaves is attached to the middle internal node. This subtree creates a disconnected component on 3 internal nodes in the GIN (Figure 10(b)). For a chosen order, one obtains $(C_4^2)^l = 6^l$ possible labelling functions for a number of leaves of T that is linear in l (in fact, $10l+2$ leaves). In this example on a small alphabet (here 4), the number of labelling functions is exponential in the size of T. Clearly, this example input satisfies the three preconditions of [5]: (1) the root has as many children as the alphabet letters, (2) one of its children is a leaf, and it can be seen that the labelling function satisfies condition (3).

Finally, if k-multiedges are seen as weighted edges, the GIN is linear in the number of internal nodes plus the number of children of the root of T. The Suffix Tour Graph is linear in the number of leaves of T. As in a tree, the number of internal nodes is strictly smaller that of leaves, the GIN takes less space in memory than the Suffix Tour Graph. Hence, some procedures, such as checking the connectivity, will be faster for the GIN.

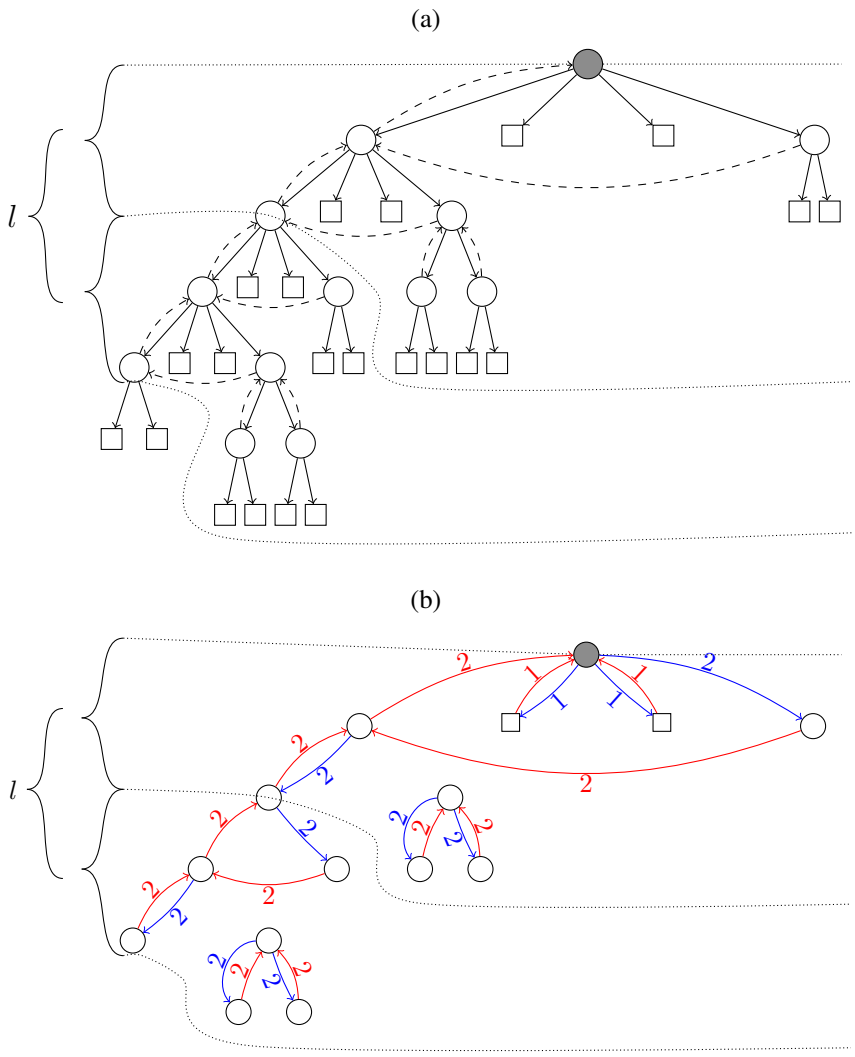


Figure 10: (a) An example of input (T, F), whose GIN is not connected (b). We build it for $l = 2$ and have a number of leaves that is linear in l. The algorithm from [5] must check the connectivity of the Suffix Tour Graph for an exponential number (6^l) of possible labelling functions. Our algorithm simply checks once that the GIN is disconnected (b).

6 Conclusion

In this work, we examined the problem of Reverse Engineering on Suffix Trees and Links, which in fact formulates the question of a characterisation of suffix trees. First, we exhibit necessary conditions for a candidate tree. Second, we define a novel bi-coloured directed graph on a subset of the ST's internal nodes: the nodes having a child that is a leaf. After defining labels on its edges, we show that finding a specific Eulerian route through this graph on which the labels satisfy necessary conditions gives a word realising the input suffix tree and links. This algorithm explores potential Eulerian routes in a graph whose size is linear in the input size. Basically, the results

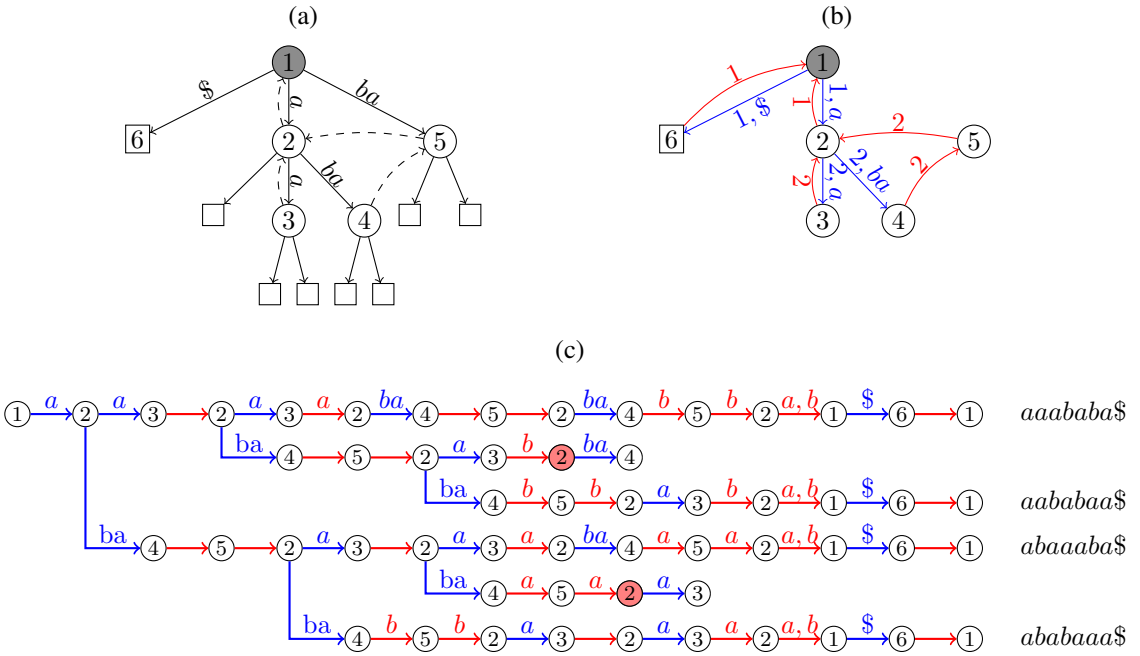


Figure 11: Example illustrating the comparison of I et al's approach and ours. (a) Instance of the example shown in [5] and (b) the associated GIN. (c) A tree representation of the routes visited by Algorithm 2 on the GIN with their associated words (for the sake of legibility, we omit all primary routes ending at the root that are too short). Edges of B and their labels are blue, edges of R and their set of forbidden first letters in red. Routes that include a violation have their next to last node painted in red.

comes from the novel graph, the GIN, whose nodes are a subset of internal nodes of T, rather than all nodes of T as in [5], and also from the fact that our algorithm does not require a labelling function as input. To illustrate the gain in efficiency, we also exhibit a general negative instance, where the approach from I et al. must consider an exponential number of labelling functions, while ours simply detect that the GIN is disconnected in linear time.

In future work, the current problem could be studied for implicit suffix trees (that is without a terminating dollar symbol), and of course the Reverse Engineering problem of Suffix Tree without links must be studied. However, the complexity of the version studied here let us think that the general case could be hard since the length of each label could not be computed from the potential suffix links.

Acknowledgements

This work is supported by ANR Colib'read (ANR-12-BS02-0008) and Défi MASTODONS SePhHaDe from CNRS.

References

[1] Alberto Apostolico. The myriad virtues of suffix trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Word*, pages 85–96. Springer-Verlag, 1985.

[2] Bastien Cazaux and Eric Rivals. Reverse engineering of compact suffix trees and links: A novel algorithm. *J. of Discrete Algorithms*, 28:9–22, 2014. StringMasters 2012-2013 Special Issue (Volume 1).

[3] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences*. Computer Science and Computational Biology. Cambridge University Press, 1997.

[4] Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Inferring strings from suffix trees and links on a binary alphabet. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2011*, pages 121–130, 2011.

[5] Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Inferring strings from suffix trees and links on a binary alphabet. *Discrete Applied Mathematics*, 163(3):316–325, 2014.

[6] E. McCreight. A space-economical suffix tree construction algorithm. *Journal of the Association for Computing Machinery*, 23(2):262–272, April 1976.

[7] N. Philippe. Caractérisation et énumération des arbres compacts des suffixes. Master's thesis, Université de Rouen, 2007.

[8] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14(3):249–260, 1995.

[9] P. Weiner. Linear pattern matching algorithms. In *Conf. Record of the 14th Annual Symposium on Swithcing and Automata Theory*, pages 1–11, 1973.