Pattern Discovery in Colored Strings

Liptak, Zsuzsanna

2021-12

Pattern Discovery in Colored Strings

ZSUZSANNA LIPTÁK, University of Verona, Department of Computer Science, Italy

SIMON J. PUGLISI, Helsinki Institute of Information Technology (HIIT) and Department of Computer Science, University of Helsinki, Finland

MASSIMILIANO ROSSI, University of Florida, Department of Computer and Information Science and Engineering, United States

In this paper, we consider the problem of identifying patterns of interest in colored strings. A colored string is a string where each position is assigned one of a finite set of colors. Our task is to find substrings of the colored string that always occur followed by the same color at the same distance. The problem is motivated by applications in embedded systems verification, in particular, assertion mining. The goal there is to automatically find properties of the embedded system from the analysis of its simulation traces.

We show that, in our setting, the number of patterns of interest is upper-bounded by $O(n^2)$, where n is the length of the string. We introduce a baseline algorithm, running in $O(n^2)$ time, which identifies all patterns of interest satisfying certain minimality conditions, for all colors in the string. For the case where one is interested in patterns related to one color only, we also provide a second algorithm which runs in $O(n^2 \log n)$ time in the worst case but is faster than the baseline algorithm in practice. Both solutions use suffix trees, and the second algorithm also uses an appropriately defined priority queue, which allows us to reduce the number of computations. We performed an experimental evaluation of the proposed approaches over both synthetic and real-world datasets, and found that the second algorithm outperforms the first algorithm on all simulated data, while on the real-world data, the performance varies between a slight slowdown (on half of the datasets) and a speedup by a factor of up to 11. [1]

CCS Concepts: • **Theory of computation** → **Design and analysis of algorithms**; • **Computer systems organization** → Embedded systems.

Additional Key Words and Phrases: property testing, suffix tree, pattern mining, efficient algorithm

1 INTRODUCTION

In recent years, embedded systems have become increasingly pervasive and are becoming fundamental components of everyday life. In line with this, embedded systems are required to perform more and more demanding tasks, and in many circumstances, peoples' lives are now dependent on the correct functioning of these devices. This, in turn, has led to an increasingly complex design

---

[1]A preliminary version of this paper was presented at SEA2020 [25]

Authors' addresses: Zsuzsanna Lipták, zsuzsanna.liptak@univr.it, University of Verona, Department of Computer Science, Strada le Grazie, 15, Verona, Italy; Simon J. Puglisi, puglisi@cs.helsinki.fi, Helsinki Institute of Information Technology (HIIT), Department of Computer Science, University of Helsinki, Helsinki, Finland; Massimiliano Rossi, rossi.m@ufl.edu, University of Florida, Department of Computer and Information Science and Engineering, Gainesville, United States.

process for embedded systems, where a major design task is to evaluate and check the correctness of the functionality from the early stages of the development process. This functionality checking is usually done using *assertions* — logic formulae expressed in temporal logic such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) — that provide a way to express desirable properties of the device. Assertions are typically written by hand by the designers, and it might take months to obtain a set of assertions that is small and effective (i.e. it covers all functionalities of the device) [17]. In order to help designers with the verification process, methodologies and tools have been developed which automatically generate assertions from simulation traces of an implementation of the device [9, 10, 26, 37]. The objective is to provide a small set of assertions that cover all behaviors of the device, in order to extend the basic manually-defined set of assertions.

A simulation trace can be viewed as a table that records, for every simulation instant T, the value assumed by the input and output ports of the device. Figure 1a shows an example of a simulation trace of a device with three input ports $\mathcal{I} = \{i_1, i_2, i_3\}$ and two output ports $O = \{o_1, o_2\}$. An assertion is a logic formula expressed in temporal logic that must remain true in the whole trace. The simplest assertions involve only conditions occurring at the same simulation instant. In the simulation trace in Figure 1a, from the solid and dashed shaded boxes, we can assert that each time we have $i_1 = 1$, $i_2 = 0$, and $i_3 = 1$, then $o_1 = 1$ and $o_2 = 1$. On the other hand, we cannot assert that each time we have $i_1 = 1$, $i_2 = 1$, and $i_3 = 0$, then $o_1 = 1$ and $o_2 = 1$, because there is a counterexample in the simulation trace, namely at instant $T = 9$, where $o_1 = 0$ and $o_2 = 0$. Note that the assertions do not need to contain all input and output variables, e.g. we can assert that $i_1 = 0$ and $i_3 = 0$ implies $o_2 = 0$.

Among all possible types of assertions that can be expressed in temporal logic, an interesting one is given by chains of *next:* sequences of consecutive input values that, when provided to the device, uniquely determine their output, after a certain number of simulation instants. For example, in the simulation trace in Figure 1a, we can assert that each time we have, for (i_1, i_2, i_3), the values $(0, 1, 0)$, $(1, 1, 0)$, $(0, 1, 0)$ in consecutive simulation instants, then, three instants later, we will see $o_1 = 1$ and $o_2 = 0$.

We model simulation traces with *colored strings*. A colored string is a string over an alphabet Σ, where each position is additionally assigned a color from an alphabet Γ. We will set Σ as the set of tuples of possible values for the input ports $i_1, \ldots, i_k$ and Γ as that of the output traces $o_1, \ldots, o_r$. The objective then is to identify patterns in the string whose occurrence is always followed by the same color at some given distance.

1.1 Related Work

Pattern mining was originally motivated by the need to discover frequent itemsets and association rules in basket data, i.e. items that were frequently bought together in a retail store. The seminal *Apriori algorithm* [1] can discover that type of pattern and has become very popular (with many extensions and variations) due to its wide applicability in other data-intensive domains. Time relationship, e.g., between entries of the database in which the basket data are stored, were later considered in so-called *sequential pattern mining* [2].

In sequential pattern mining, *episodes* are partially ordered sequences of events that appear close to each other in the sequence [29]. Given episodes of the sequence, it is possible to build *episode rules* that establish antecedent-consequent relations among episodes. Sequential pattern mining has many applications (see,e.g., [6, 12, 24]) and has been surveyed extensively [18, 27, 32].

Unfortunately, the above setting is not applicable to our problem, since here time is given only in a relative sense, i.e., whether an event happens before (or after) another event, while we need to count exactly the instants that occurs between the two events.
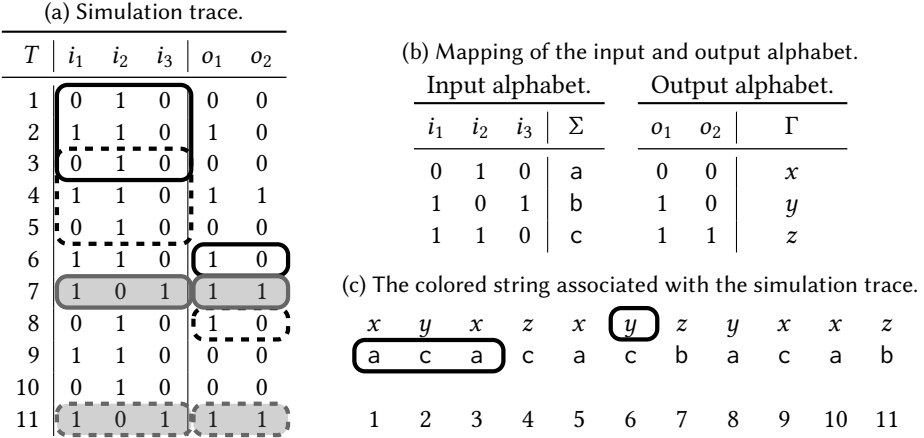
(a) Simulation trace.

T	i_1	i_2	i_3	o_1	o_2
1	0	1	0	0	0
2	1	1	0	1	0
3	0	1	0	0	0
4	1	1	0	1	1
5	0	1	0	0	0
6	1	1	0	1	0
7	1	0	1	1	1
8	0	1	0	1	0
9	1	1	0	0	0
10	0	1	0	0	0
11	1	0	1	1	1

(b) Mapping of the input and output alphabet.

Input alphabet.

i_1	i_2	i_3	Σ
0	1	0	a
1	0	1	b
1	1	0	c

Output alphabet.

o_1	o_2	Γ
0	0	x
1	0	y
1	1	z

(c) The colored string associated with the simulation trace.

x	y	x	z	x	y	z	y	x	x	z
a	c	a	c	a	c	b	a	c	a	b
1	2	3	4	5	6	7	8	9	10	11

Fig. 1. Example of a simulation trace of a device with input ports $\mathcal{I} = \{i_1, i_2, i_3\}$, and output ports $O = \{o_1, o_2\}$. The mapping of the input and output values of the trace into the input and output alphabet respectively. The colored string associated to the simulation trace, after the mapping. The solid and dashed shaded and non-shaded boxed values in the simulation trace highlight that every time we see the sequence of input values, then we have the corresponding output value. The solid non-shaded boxed characters in the colored string are the mapping of the corresponding solid non-shaded boxed values in the simulation trace.

In the *string mining problem* [11, 14–16, 36], one aims to discover strings that appear as a substring in more than ω strings in a collection, where ω is a user-defined parameter called *support* of the string. This can be also used to find strings that discriminate between two collections, i.e., strings that are frequent in one collection and not frequent in the other. These strings are called *emerging strings* and find important applications in data mining [33], knowledge discovery in databases [5] and in bioinformatics [3]. In the field of knowledge discovery on databases, the problem has been extended to mining frequent subsequences [22] and distinguishing subsequence patterns with gap constraints [23, 31, 38, 39].

In [21] Hui proposed a solution for the *color set size problem*. Here, given a tree and a coloring of its leaves, the objective is to find, for all internal nodes of the tree the number of distinct colors in the leaves of its subtree. In [21], the color set size problem is applied to several string matching and string mining problems, e.g., given a collection of m strings, find the longest pattern which appears in at least $1 \leq k \leq m$ strings. Note that if the tree of the color set size problem is the suffix tree of a string s, then s with the coloring of its suffixes can be seen as a colored string. In spite of this similarity, both, the problems that we solve, and the approaches we use, are different.

In assertion mining, the two existing tools, *GoldMine* [37] and *A-Team* [9], are based on data mining algorithms. In particular, *GoldMine* [37] extracts assertions that predicate only on one instant of the simulation trace—i.e. they do not involve any notion of time−, using decision tree based mining or association mining [1]. Furthermore, using static analysis techniques together with sequential pattern mining, it extracts temporal assertions. The tool A-Team [9], requires the user to provide the template of the temporal assertions that they want to extract. For example, in order to extract the properties of our example in Fig. 1a, one needs to provide a template stating that we want a property of the form: "a property p_1, at the next simulation instant a property p_2, at the next simulation instant a property p_3, then after three simulation instants a property p_4". Given a set of templates, the software, using an Apriori algorithm, extracts propositions (logic

formulae containing the logical connectives $\neg$, $\vee$, and $\wedge$) from the trace. Once the propositions have been extracted, the tool generates the assertion by instantiating the extracted propositions in the templates, using a decision-tree-based algorithm to find formulas that fit in the template and are verified in the simulation trace, i.e. if the trace contains no counterexample.

1.2 Our Contribution

In this work we introduce colored strings, and propose and analyze two pattern discovery problems on colored strings which correspond to a useful simplification of pattern mining w.r.t. assertion mining. In both problems, we are given a colored string as input. Given a colored string and a color as input, in the first problem, we must find all minimal substrings that occur followed always at the same distance by the given color. In the second problem, the color is not fixed, thus we want to find all minimal substrings which occur followed always at the same distance by the same color. We define these problems formally in Section 2.

Although these problems are simpler than the original assertion mining problem, the solution to our problem contains all the information, possibly filtered, to recover the desired set of minimal assertions in a second stage. For example, let us assume that the device that produced the simulation trace in Figure 1a has the following behavior: every time that $i_1 = 0$, at the next instant $i_1 = 1$, and at the next instant $i_1 = 0$, then after three instants $o_1 = 1$ and $o_2 = 0$. A solution to our problem will include all patterns of length 3 for which i_1 takes consecutive values $0, 1, 0$, while i_2 and i_3 have arbitrary values, since all of these will result in $o_1 = 1$ and $o_2 = 0$ three instances later.

We first upper bound the number of minimal patterns by $O(n^2)$. We then propose two algorithms which find all minimal patterns, when only one color is of interest (base), and when one is interested in all colors (base-all). Both of these algorithms use the suffix tree of the reverse string as underlying data structure. We note that since this is a pattern mining problem, every efficient algorithm for the problem will necessarily use a dedicated string data structure (or index), such as a suffix tree, since all occurrences of substrings have to be considered concurrently.

Then we show that, in the case of one color, the first algorithm can be improved. The new algorithm, referred to as skipping, also uses the suffix tree as its underlying data structure, together with an appropriately defined priority queue. This allows us to reduce the number of computations in practice, even though the theoretical running time of the new algorithm is worse, namely $O(n^2 \log n)$. We provide an experimental evaluation of the proposed approaches. Finally, we consider (practically motivated) restrictions on the patterns and show that under these restrictions performance of the skipping algorithm is further improved.

The paper is structured as follows. In Section 2 we fix definitions and notations and give the problem statements. In Section 3 we present baseline algorithms, base and base-all, which solve these problems. In Section 4 we present the modified algorithm skipping, which solves the pattern discovery problem for only one color. In Section 5 we introduce real-world data oriented restrictions on the output. In Section 6 we present an experimental evaluation of the proposed approaches. Conclusions and future work can be found in Section 7.

2 BASICS

Let Σ be a finite ordered set. We refer to Σ as *alphabet* and to its elements as *characters*. A *string over* Σ is a finite sequence of characters $S = S[1, n]$, where $|S| = n$ is the *length* of string S. We denote by ε the *empty string*, the unique string of length 0. Note that we number strings starting from 1, and we use the array-notation for strings: we denote the i'th character of S by $S[i]$ and use $S[i, j]$ to refer to the string $S[i] \cdots S[j]$, if $i \leq j$, while $S[i, j] = \varepsilon$ if $i > j$. Given string $S = S[1, n]$, the

reverse string is the string $S^{\text{rev}} = S[n]S[n-1]\cdots S[1]$. For string S and $1 \leq i \leq n$, $\text{Pref}_i(S) = S[1,i]$ is called the *i'th prefix* of S, and $\text{Suf}_i(S) = S[i,n]$ is called the *i'th suffix* of S. A *substring* of a string S is a string T for which there exist i, j s.t. $T = S[i,j]$; in this case the position i is referred to as an *occurrence* of T in S. A substring T of S is called *proper* if $T \neq S$. When S is clear from the context, then we may refer to T simply as a *substring*.

2.1   Colored strings

Given two finite sets Σ (the alphabet) and Γ (the colors), a *colored string* over (Σ, Γ) is a string $S = S[1,n]$ over Σ together with a coloring function $f_S : \{1, \ldots, n\} \to \Gamma$. We denote by $\sigma = |\Sigma|$ and $\gamma = |\Gamma|$ the number of characters resp. of colors. Given a colored string S of length n, its reverse is denoted S^{rev}, and its coloring function $f_{S^{\text{rev}}}$ is defined by $f_{S^{\text{rev}}}(i) = f_S(n-i+1)$, for $i = 1, \ldots, n$. When S is clear from the context, we write f for f_S and f^{rev} for $f_{S^{\text{rev}}}$.

We are interested in those substrings which are always followed by a given color y, at a given distance d. Look at the following example.

Example 2.1. Let $S = acacacbacab$, with colors $xyxzxyzyxxz$:

x	y	x	z	x	y	z	y	x	x	z
a	c	a	c	a	c	b	a	c	a	b
1	2	3	4	5	6	7	8	9	10	11

The substring aca occurs 3 times in S, at positions $1, 3$, and 8. In positions 1 and 3 it is followed by a y at distance 3, while at position 8, the corresponding position is beyond the end of the string.

This leads to the following definitions:

Definition 2.2 (y-good, y-unique, minimal). Let S be a colored string over (Σ, Γ), $y \in \Gamma$ a color, $d \leq n$ a non-negative integer, and $T = T[1,m]$ a substring of S.

   (1) An occurrence i of T is called *y-good with delay d* (or *(y,d)-good*) if $f(i+m-1+d) = y$.
   (2) T is called *y-unique with delay d* (or *(y,d)-unique*) if for every occurrence i of T, i is (y,d)-good or $i+m-1+d > n$.
   (3) T is called *minimally (y,d)-unique* if there exists no proper substring U of T which is *y-unique* with delay d', for some d' s.t. $U = T[i,j]$ and $d' = d + |T| - j$.

In the Example 2.1, the occurrence of aca in position 1 is $(y,3)$- and $(y,5)$-good, that in position 3 is $(y,1)$- and $(y,3)$-good, while that in position 8 is not (y,d)-good for any d. Therefore, the substring $T = $ aca is a $(y,3)$-unique substring of S, since every occurrence i of aca is either $(y,3)$-good (position 1 and 3) or $i+m-1+d > n$ (position 8). However, aca is not minimal, since its substring ca is also $(y,3)$-unique (where $d' = d$, since ca is a suffix of aca).

The introduction of minimally (y,d)-unique substrings serves to restrict the output size. Let $T = aXb$ be (y,d)-unique, with $a, b \in \Sigma$ and $X \in \Sigma^*$. We call T *left-minimal* if Xb is not (y,d)-unique, and *right-minimal* if aX is not $(y,d+1)$-unique. We make the following simple observations about (y,d)-unique substrings. (Note that *2* is a special case of *3*.)

OBSERVATION 1. *Let $S \in \Sigma^*$ and let T be a (y,d)-unique substring of S.*

   (1) *T is minimal if and only if it is left- and right-minimal.*
   (2) *If T is a suffix of T', then T' is also (y,d)-unique.*
   (3) *If $T' = UTV$ is a superstring of T such that $|V| \leq d$, then T' is $(y,d-|V|)$-unique.*

We are now ready to formally state the problems treated in this paper.

PROBLEM 1 (PATTERN DISCOVERY PROBLEM). *Given a colored string S and a color y, report all pairs (T, d) such that T is a minimally (y, d)-unique substring of S.*

PROBLEM 2 (UNRESTRICTED-OUTPUT PATTERN DISCOVERY PROBLEM). *Given a colored string S, report all triples (T, y, d) such that T is a minimally (y, d)-unique substring of S.*

We next give an upper bound on the number of minimally (y, d)-unique substrings.

LEMMA 2.3. *Given string S of length n, the number of minimally (y, d)-unique substrings of S, over all $y \in \Gamma$ and $d = 0, \ldots, n$, is $O(n^2)$.*

PROOF. Note that, given a position j and a delay d, every substring occurrence ending in j is $(f_S(j + d), d)$-good. Therefore, for a substring U with an occurrence ending in position j, and for fixed d, it holds that, if U is (y, d)-unique for some y, then $y = f_S(j + d)$. Moreover, it follows from Observation 1 that, given y, d, and j, at most one minimally (y, d)-unique substring can end at position j. Altogether we have that the number of minimally (y, d)-unique substrings is $O(n^2)$, over all y and d.   □

2.2 Suffix trees and suffix arrays

Let S be a string over Σ and \$ a new character not belonging to Σ. We denote by $\mathcal{T}(S)$ the *suffix tree* of $S\$$, i.e. the compact trie of the suffixes of $S\$$. For a general introduction to suffix trees, see, e.g., [20, 28, 35]. Here we recall some basic facts.

The suffix tree $\mathcal{T}(S)$ is a rooted tree in which all internal nodes are branching. Each edge is labeled with a non-empty substring of S such that the labels of any two outgoing edges from the same node start with a different character. Edge labels are stored in form of two pointers $[i, j]$ into the string with the property that $S[i, j]$ equals the label of the edge. If $|S| = n$, then $\mathcal{T}(S)$ has exactly $n + 1$ leaves, each labeled by a position from $\{1, \ldots, n + 1\}$, denoted $ln(v)$ (for *leaf number*). For a node v in $\mathcal{T}(S)$, we denote by $L(v)$ the concatenation of the edge labels on the path from the root to node v. The string $L(v)$ is sometimes referred to as the substring represented by node v. If v is a leaf with $ln(v) = i$, then $L(v)$ is equal to the i'th suffix of $S\$$, $Suf_i(S\$)$. For a node v, we denote by $td(v)$ its *treedepth*, the number of edges on the path from the root to v, and by $sd(v) = |L(v)|$ its *stringdepth*, the length of the string represented by v. Given node v not equal to the root, $parent(v)$ is the node which is next on the path from v to the root. Given a node v which is not a leaf and a character $c \in \Sigma$, $child(v, c)$ returns the unique node u with parent v such that the label of the edge (v, u) starts with character c, or the empty pointer if no such node exists.

Given a node u with parent v, a *locus* is a pair (u, t) s.t. $sd(v) < t \leq sd(u)$. Let $[i, j]$ be the label of edge (v, u) and $k = t - sd(v)$. We define $L(u, t)$ as the string $L(v) \cdot S[i, i + k - 1]$, the substring represented by locus (u, t). Note that if $t = sd(u)$, then $L(u, t) = L(u)$. It is an important property of suffix trees that there is a one-to-one correspondence between loci of $\mathcal{T}(S)$ and substrings of $S\$$. This allows us to define, for a substring T of S (which is also a substring of $S\$$), the *locus of* T, $loc(T) = loc(T, \mathcal{T}(S))$ as the unique locus (u, t) in $\mathcal{T}(S)$ with the property that $L(u, t) = T$. Given a substring T of S with locus $loc(T) = (u, t)$, the set of occurrences of T is given by the set $\{ln(v) \mid v$ is leaf in the subtree rooted in $u\}$.

Let u be a node and $L(u) = aT$, where $a \in \Sigma$ and $T \in \Sigma^*$. The *suffix link* of u is defined as $slink(u) = loc(T)$. It can be shown that for any node u, $slink(u)$ is a node of $\mathcal{T}(S)$ (rather than just a locus). Suffix links can also be defined for loci: for locus (u, t) with $L(u, t) = aT$, define $slink(u, t) = loc(T)$; these are also called *implicit suffix links*. Suffix links are often represented by directed edges, see Figure 2.

Given a suffix tree $\mathcal{T}(S)$ with k nodes, and a node u of $\mathcal{T}(S)$, let r be the rank of the node u in the breath-first search traversal of the tree. We define the reverse index BFS of u as $iBFS(u) = k - r$. Refer to Figure 3 for an example of the reverse index BFS values.

Given the string S of length n, we denote by $SA_S[1, n + 1]$ the *suffix array* of $S\$$. We refer the reader to, e.g., [28], for a general introduction to suffix arrays.

The suffix array $SA_S[1, n+1]$ of a string $S\$$ is a permutation of $\{1, \ldots, n+1\}$ such that $SA_S[i] = j$ if and only if $S[j, n]\$$ is i-th suffix in the lexicographically ordered list of suffixes of $S\$$. The suffix array SA_S and the suffix tree $\mathcal{T}(S)$ are deeply related. We can obtain the SA_S by listing the leaves of the suffix tree $\mathcal{T}(S)$ from left to right, assuming that the children are ordered according to the first characters of their edge labels. In particular, for an inner node u, the leaves in the subtree rooted in u yield an interval of the suffix array $SA_S[i, j]$ such that $\{ln(v) \mid v$ is leaf in the subtree rooted in $u\} = \{SA_S[k] \mid i \le k \le j\}$.

2.3 Maximum-oriented indexed priority queue

A maximum-oriented indexed priority queue [34, Sec. 2.4] denoted by IPQ, is a data structure that collects a set of m items with keys $k_1, \ldots, k_m$, and provides the following operations:

- insert(i,k): insert the element at index i with key $k_i = k$.
- promote(i,k): increase the value of the key k_i, associated with i, to $k \ge k_i$.
- demote(i,k): decrease the value of the key k_i, associated with i, to $k \le k_i$.
- $(i,k) \leftarrow$max$()$: return the index i and the value k of the item with maximum key k_i; if two items have the same key value, we report the item with larger index.
- $k \leftarrow$keyOf(i): return the value of the key k_i associated with index i.
- $b \leftarrow$isEmpty$()$: return *true* if the IPQ is empty and *false* otherwise.
- delete(i): remove the element at index i from the IPQ.

The operations insert, promote, demote and delete run in $O(\log(m))$ time, while the operations max, keyOf and isEmpty are performed in $O(1)$ time.

For our purposes, we also require a function $b \leftarrow$allNegative$()$ that returns *true* if all key values are negative, and *false* otherwise.

We use the IPQ to store keys associated to nodes u of a suffix tree $\mathcal{T}(S)$ using $iBFS(u)$ as index. For ease of presentation, in slight abuse of notation, we will use u and $iBFS(u)$ interchangeably.

2.4 Rank, select, and range maximum query

A *bitvector* $B[1, n]$ of length n is an array of n bits. For all $1 \le i \le n$ and $b \in \{0, 1\}$, we define rank$_b(B, i)$ as the number of occurrences of b in $B[1, i]$, and select$_b(B, i)$ as the index of the i-th occurrence of the symbol b in B. If $i >$ rank$_b(B, n)$ then select$_b(B, i) = n + 1$. Furthermore, we set select$_b(B, 0) = 0$. For both rank and select operations, if b is omitted we assume $b = 1$. Given a bitvector B, rank and select operations can be supported in $O(1)$ time using $o(n)$ bits of extra space [7].

For an array $A[1, n]$ of n integers and $1 \le i \le j \le n$, a *range maximum query* rMq$_A(i, j)$ returns the position of the maximum element of $A[i, j]$. This answer can be provided in $O(1)$ time using $2n + o(n)$ bits of space [13].

Given $A[1, n]$ and the range maximum query data structure for A, we can compute the position of the second greatest element of $A[i, j]$ in $O(1)$ time. In particular, let $a =$ rMq$_A[i, j]$, we have three cases: (i) if $a = i$, then the position of the second greatest element of $A[i, j]$ is $c =$ rMq$_A[a + 1, j]$; (ii) if $a = j$, then the position of the second greatest element of $A[i, j]$ is $b =$ rMq$_A[i, a - 1]$; (iii)

otherwise, let $b = \mathrm{rMq}_A[i, a-1]$, and $c = \mathrm{rMq}_A[a+1, j]$. The position of the second greatest element of $A[i, j]$ is b if $A[b] \geq A[c]$, otherwise it is c, since $A[c] > A[b]$.

3 A PATTERN DISCOVERY ALGORITHM FOR COLORED STRINGS USING THE SUFFIX TREE

Our main tool will be the suffix tree of the reverse string, $\mathcal{T} = \mathcal{T}(S^{\mathrm{rev}})$. Note that loci in $\mathcal{T}$ correspond to ending positions of substrings of S in the following sense. Given a locus (u, t) of $\mathcal{T}$, let $U = L(u, t)^{\mathrm{rev}}$. Then U is a substring of S, and its occurrences are exactly the positions $i - |U| + 1$, where $i = n - ln(v) + 1$ for some leaf v in the subtree rooted in u. In the next lemma we show how to identify (y, d)-unique substrings of S with $\mathcal{T}$, the suffix tree of S^{rev}.

LEMMA 3.1. *Let U be a substring of S, $\mathcal{T} = \mathcal{T}(S^{\mathrm{rev}})$, and $(u, t) = loc(U^{\mathrm{rev}}, \mathcal{T})$. Then U is y-unique with delay d in S if and only if for all leaves v in the subtree rooted in u, $S^{\mathrm{rev}}[ln(v) - d]$ is colored y under f^{rev}. In particular, U is y-unique with delay 0 in S if and only if all leaves in the subtree rooted in u are colored y under f^{rev}.*

PROOF. It is easy to see that position $i - |U| + 1$ is a y-good occurrence of U in S with delay 0 if and only if U^{rev} is a prefix of $\mathrm{Suf}_{n-i+1}(S^{\mathrm{rev}})$ and $f^{\mathrm{rev}}(n - i + 1) = y$. By the properties of the suffix tree, all occurrences of U^{rev} correspond to the leaves of the subtree rooted in u, where $(u, t) = loc(U^{\mathrm{rev}}, \mathcal{T})$. Thus, U is $(y, 0)$-unique if and only if all of its occurrences are $(y, 0)$-good, which is the case if and only if all leaves of the subtree rooted in u are colored y under f^{rev}. More generally, position $i - |U| + 1$ is a y-good occurrence of U in S with delay d if and only if $\mathrm{Suf}_{n-i+1}(S^{\mathrm{rev}})$ is prefixed by U^{rev} and $f^{\mathrm{rev}}(n - i + 1 - d) = y$. Thus U is (y, d)-unique if and only if for all leaves v in the subtree rooted in u, $S^{\mathrm{rev}}[ln(v) - d]$ is colored y under f^{rev}.   □

In the following, we will refer to a *node* u of $\mathcal{T}$ as (y, d)-*unique* if $L(u)^{\mathrm{rev}}$ is a (y, d)-unique substring of S. We can now state the following corollary:

COROLLARY 3.2. *Let U be a substring of S, $\mathcal{T} = \mathcal{T}(S^{\mathrm{rev}})$, and $(u, t) = loc(U^{\mathrm{rev}}, \mathcal{T})$ such that u is an inner node of $\mathcal{T}(S)$. Then U is (y, d)-unique in S if and only if all children of u are (y, d)-unique.*

3.1 Finding all (y, d)-unique substrings

Our first algorithm ALGO1 uses the suffix tree $\mathcal{T}$ of the reverse string to identify all (y, d)-unique substrings of S, not only the minimal ones, for fixed y and d. It marks the (y, d)-unique nodes of $\mathcal{T}$ in a postorder traversal of the tree. Note that if $i > n - d$, then position i cannot be (y, d)-good, simply because the position in which we would expect a y lies beyond the end of string S. In correspondence with the definition of (y, d)-unique substrings (Definition 2.2), we will treat such positions as if they were (y, d)-good.

The function $g(u) : V(\mathcal{T}) \rightarrow \{0, 1\}$ is defined as follows:

- for a leaf u with leaf number $ln(u) = i$:

$$g(u) = \begin{cases} 1 & \text{if either } i \leq d \text{ or } f^{\mathrm{rev}}(i - d) = y, \\ 0 & \text{otherwise,} \end{cases}$$

- for an inner node u:

$$g(u) = \begin{cases} y & \text{if } g(v) = 1 \text{ for all children } v \text{ of } u, \\ 0 & \text{otherwise.} \end{cases}$$

The algorithm computes $g(u)$ for every node u in a bottom-up fashion, assigning $g(u) = 1$ if and only if u is (y, d)-unique or if it is too close to the beginning of the string S^{rev}. If $g(u) = 1$,

---

Algorithm 1: ALGO1

---

input : A colored string S, the suffix tree $\mathcal{T}$ of S^{rev}, and $y \in \Gamma$.
output: All pairs (T, d) such that T is a (y, d)-unique substring of S.

1 **for** $d \leftarrow 0$ *to* n **do**
2 $\quad$ UNIQUE($root, y, d$)

3 **procedure** UNIQUE(u, y, d):
4 $\quad$ **if** *u is a leaf* **then** $\qquad\qquad\qquad\qquad\qquad\qquad$ `// u is a leaf`
5 $\quad\quad$ $i \leftarrow ln(u)$
6 $\quad\quad$ **if** $i \leq d$ **or** $f^{rev}(i - d) = y$ **then**
7 $\quad\quad\quad$ $g(u) \leftarrow 1$
8 $\quad\quad$ **else**
9 $\quad\quad\quad$ $g(u) \leftarrow 0$

10 $\quad$ **else** $\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ `// u is an inner node`
11 $\quad\quad$ $g(u) \leftarrow \wedge_{v \text{ child of } u} \text{UNIQUE}(v, y, d)$

12 $\quad$ **if** $g(u) = 1$ **then**
13 $\quad\quad$ **if** *u is a leaf* **then** $\qquad\qquad\qquad$ `// do not output $-substrings`
14 $\quad\quad\quad$ output $L(u, t)^{\text{rev}}$ for every $t = sd(parent(u)) + 1, \ldots, sd(u) - 1$
15 $\quad\quad$ **else**
16 $\quad\quad\quad$ output $(L(u, t)^{\text{rev}}, d)$ for every $t = sd(parent(u)) + 1, \ldots, sd(u)$

17 $\quad$ **return** $g(u)$

---

in addition it outputs all strings represented along the incoming edge of u, except for substrings which contain the $-sign, i.e. suffixes of S^{rev}\$. For details, see Algorithm 1.

Analysis: For fixed d, computing g takes amortized $O(n)$ time over the whole tree, since computing $g(u)$ is linear in the number of children of u, and therefore, charging the check whether for a child v, $g(v) = 1$, to the child node, we get constant time per node. So, for fixed d, we have $O(n+K) = O(n^2)$ time, where K is the number of (y, d)-unique substrings. Altogether, for $d = 0, \ldots, n$, the algorithm takes $O(n^3)$ time.

Example 3.3. In the running example (Fig. 2), for color y and delay $d = 3$, the leaf nodes 9, 2, 7, 1, and 3 are marked with 1, and therefore the only inner node u which gets $g(u) = 1$ is the parent of leaves number 9, 2, 7. ALGO1 outputs the $(y, 3)$-unique substrings baca, cbaca, acbaca, cacbaca, acacbaca, cacacbaca, acacacbaca, caca, acaca, ca, aca, ab, cab, acab, bacab, cbacab, acbacab, cacbacab, acacbacab, cacacbacab, bac, cbac, acbac, cacbac, acacbac, cacacbac, acacacbac.

Remark: Note that some of these substrings do not occur even once in a position such that the last character is followed by a y with delay $d = 3$. For instance, the only occurrence of the substring bac in S is at position 7, so we would expect to see color y at position $9 + 3 = 12$, but the string S ends at position 11. We will treat this and similar questions in Section 5.

3.2 Outputting only minimally (y, d)-unique substrings

We next modify Algorithm ALGO1 to output only minimally (y, d)-unique substrings. As already noted, the work done by ALGO1 in each node is constant except for the output step, which is proportional to the length of the edge label leading to u.

In terms of the suffix tree $\mathcal{T}$ of S^{rev}, minimality can be translated into conditions on the parent node and on the suffix link parent node (equivalently: the suffix link) in $\mathcal{T}$. We first need another definition:

Definition 3.4 (Left-minimal nodes, left-minimal labels). Let u be a node of $\mathcal{T} = \mathcal{T}(S^{\text{rev}})$, different from the root, and let $v = parent(u)$. We call u *left-minimal* for (y, d) if u is (y, d)-unique but v is not and the label of the edge (v, u) is not equal to $. If u is (y, d)-unique and left-minimal, then we can define $Left\text{-}min(u) = x_1 \cdot L(v)^{\text{rev}}$, the left-minimal (y, d)-unique substring of S associated to u, where $x = x_1 \cdots x_k \in \Sigma^+$ is the label of edge (v, u).

Example 3.5. In our running example, let node u be the parent of leaf nodes 9, 2, 7, i.e. $u = loc(\mathcal{T}, \text{aca})$. Then u is left-minimal, since it is $(y, 3)$-unique but its parent is not. Its left-minimal label is $Left\text{-}min(u) = \text{ca}$. See Fig. 2.

It is easy to modify Algorithm 1 to output only left-minimal substrings: Whenever for an inner node u we get $g(u) = 0$, then for every child v of u with $g(v) = 1$, we output $Left\text{-}min(v)$ (if defined). This can be done by replacing lines 12 to 16 in Algorithm 1 by:

```
12  if g(u) = 0 then
13      for each child v of u with g(v) = 1 do
14          if Left-min(v) is defined then
15              output (Left-min(v), d)
```

Example 3.6. The resulting algorithm now outputs, for color y and $d = 3$, the left-minimal substrings ca, ab, bac.

However, we are interested in substrings which are both left- *and* right-minimal. While left-minimality can be identified by checking the parent of a node u, for right-minimality, Observation 1 part (3) tells us that we need to check whether the string without the last character is $(y, d + 1)$-unique. In $\mathcal{T}$, this translates to checking the suffix link of the locus of the left-minimal substring $Left\text{-}min(u)$.

PROPOSITION 3.7. *Let u be an inner node of $\mathcal{T} = \mathcal{T}(S^{rev})$, different from the root, such that $L(u)^{rev}$ is (y, d)-unique in S. Let $v = parent(u)$, and x_1 be the first character on the edge (v, u). Further, let $t = sd(v) + 1$, and $(u', t') = slink(u, t)$. Then the substring $U = x_1 \cdot L(v)^{rev}$ is minimally (y, d)-unique in S if and only if v is not (y, d)-unique and u' is not $(y, d + 1)$-unique.*

PROOF. For sufficiency, let U be minimally (y, d)-unique in S. Since $x_1 L(v)^{\text{rev}} = U$, and U is left-minimal, therefore v is not (y, d)-unique. Similarly, if $U' = L(u', t')^{\text{rev}}$, then we have that $U = U'a$, and u' is not $(y, d + 1)$-unique by right-minimality of U.

Conversely, since u is (y, d)-unique and v is not, by definiton of left-minimality, $U = Left\text{-}min(u)$ is left-minimal (y, d)-unique in S. Let $U' = L(u', t')^{\text{rev}}$, thus $U = U'a$, for some character $a \in \Sigma^+$. Since U' is not $(y, d + 1)$-unique, therefore U is right-minimal.   □

We can use Proposition 3.7 as follows. Once a left-minimal (y, d)-unique node u has been found, check whether u' is $(y, d+1)$-unique, where u' is the node below the locus $slink(u, sd(parent(u))+1)$. It is easy to find node u' by noting that $u' = child(slink(parent(u)), x_1)$, where x_1 is the first character of the edge label leading to u. But how do we know whether u' is $(y, d + 1)$-unique?

The answer is that we will process the distances d in descending order, from $d = n$ down to $d = 0$. At the end of the iteration for d, we retain the information, keeping a flag on every node u which

---

Algorithm 2: ALGO2

---

input : a colored string S, the suffix tree $\mathcal{T}$ of S^{rev} with suffix links, and $y \in \Gamma$.
output: all pairs (T, d) such that T is a minimally (y, d)-unique substring of S.

1 **for** $d \leftarrow n$ *downto* 0 **do**
2   MinUnique($root, y, d$)

3 **procedure** MinUnique(u, y, d):
4   **if** *u is a leaf* **then**   `// u is a leaf`
5          $i \leftarrow ln(u)$
6   **if** $i \leq d$ **or** $f^{rev}(i - d) = y$ **then**
7              $g(u) \leftarrow 1$
8   **else**
9              $g(u) \leftarrow 0$
10   **else**   `// u is an inner node`
11          $g(u) \leftarrow \wedge_{v \text{ child of } u} \text{MinUnique}(v, y, d)$
12   **if** $g(u) = 0$ **then**   `// outputting minimal substrings for children`
13   **for** *each child v with* $g(v) = 1$ **do**
14   **if** *Left-min(v) is defined* **then**
15                  $(v', t) \leftarrow slink(v, sd(u) + 1)$
16   **if** v' *is not* $(y, d + 1)$*-unique* **then**   `// flag from previous round`
17   output ($Left$-$min(v), d$)

18   **return** $g(u)$

---

was identified as (y, d)-unique (i.e. which had $g(u) = 1$). During the iteration for $d - 1$, we can then query node u' to find out whether it is (y, d)-unique. For details, see Algorithm 2.

Example 3.8. In the running example, we know from the previous round that the only nodes that are $(y, 4)$-unique are the leaves number 4, 2, 1, 10, 3, and 8. We can now deduce that the substring ca is right-minimal, because $u = loc(\text{ca})$ is not $(y, 4)$-unique, and $slink(loc(\mathcal{T}, \text{ca}^{\text{rev}})) = (u, 1)$. Looking at the string S we see that ca is indeed right-minimal, since c is not $(y, 3)$-unique: it has an occurrence, in position 6, which is not followed by a y but by an x at position $10 = 6 + 4$ (delay 4). Similarly, the left-minimal substring ab is also right-minimal, since its suffix link is not $(y, 4)$-unique, while the left-minimal substring bac is not $(y, 3)$-unique, because its suffix link is $(y, 4)$-unique, see Fig. 2.

Analysis: For fixed d, the time spent on each leaf is constant (lines 5 to 10 in ALGO2); we charge the check of $g(v)$ in line 12 to the child v, as well the work in lines 14 to 18 (computing $Left$-$min(v)$ and checking the flag on v' from the previous round); these are all constant time operations, so we have amortized constant time per node, and thus $O(n)$ time for fixed d. Therefore, the total time taken by Algorithm 2 is $O(n^2)$.

3.3 An algorithm for all colors

In some situations, one is interested in all minimally (y, d)-unique substrings, *for any color y*. Our third algorithm deals with this case (Problem 2). It is similar to Algo2, except it uses a different coloring function g'. The new function $g' : V \rightarrow \Gamma \cup \{*, 0\}$, is defined as follows:

- for a leaf u with leaf number $ln(u) = i$:

$$g'(u) = \begin{cases} f^{\mathrm{rev}}(i - d) & \text{if } i - d > 0, \\ * & \text{if } i - d \leq 0 \end{cases}$$

- for an inner node u:

$$g'(u) = \begin{cases} * & \text{if for all children } v \text{ of } u: g'(v) = *, \\ y \in \Gamma & \text{if } u \text{ has at least one child } v \text{ with } g'(v) = y \text{ and} \\ & \quad \text{for all other children } v' \text{ of } u: g'(v') \in \{y, *\}\,, \\ 0 & \text{otherwise.} \end{cases}$$

Thus a node u is colored y if and only if all leaves of the subtree rooted in u are either colored y or $*$, and at least one leaf is colored y. We refer to such a node as *monochromatic*. A node is colored $*$ if all leaves in the subtree are within d of the beginning of S^{rev}; such a node can have monochromatic ancestors in the tree. Finally, a node is colored 0 if in its subtree there are at least two leaves which are colored by different colors from Γ. For a node colored 0, all of its ancestors are also colored 0.

Example 3.9. In our example, for $d = 3$, the leaves 11, 4, and 8 are colored z, the leaves 9 and 7 are colored y, the leaves 5, 10, and 6 are colored x, and the leaves 1,2, and 3 are colored $*$. The only monochromatic inner nodes are $loc(\mathsf{b})$ (colored x), and $loc(\mathsf{aca})$ (colored y), while all others are colored 0. See Figure 2b.

Thus, Algorithm 3 finds all minimally (y, d)-unique substrings for all colors simultaneously, using the same ideas as Algo2. The main difference is that now the coloring function g' is not binary, and accordingly, the information we have to store from the previous round (which will be needed to decide whether the substring is right-minimal) is no longer binary. See Algo3 for details.

Analysis: The algorithm has n iterations, every iteration takes $O(n)$ time, so altogether we have again $O(n^2)$ time.
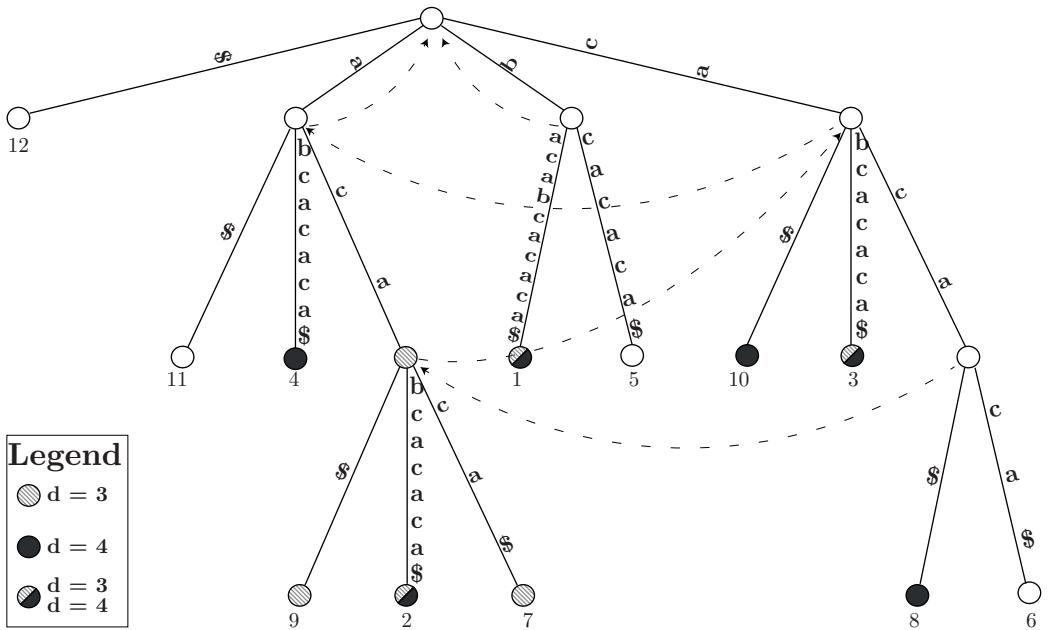
Therefore, if the color of interest is not part of the input, we can solve the problem in $O(n^2)$ time, which is also a worst-case lower bound on the output size, see Sec. 2. However, if the color y is part of the input, then this algorithm can be further improved. We will present this improvement in the next section.
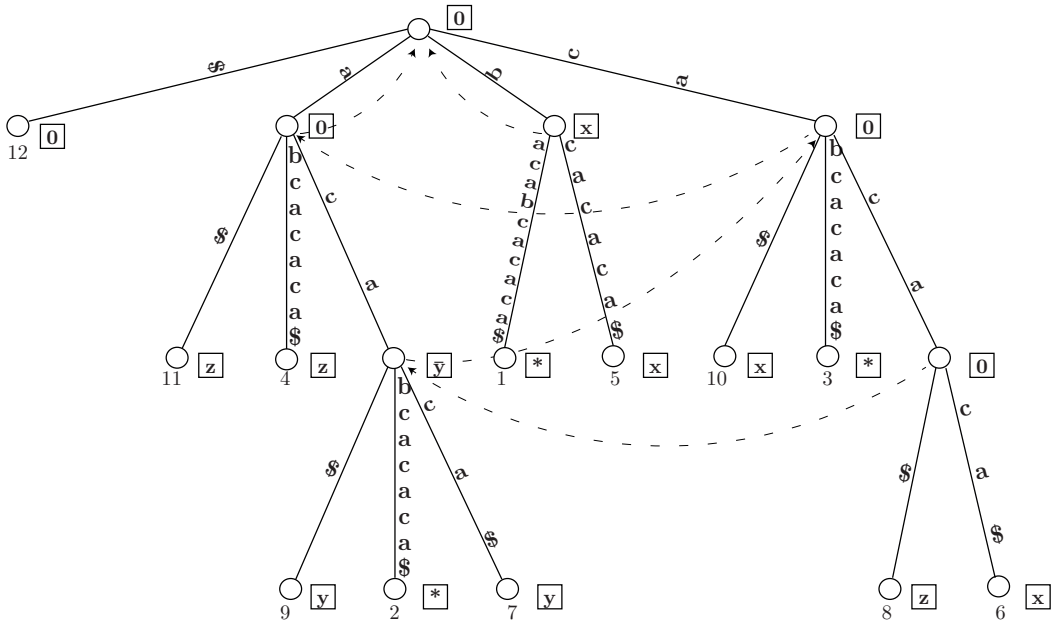
4 SKIPPING ALGORITHM

In this section, we discuss the discovering of (y, d)-unique substrings that are minimal. As in the baseline algorithm, we build the suffix tree $\mathcal{T}(S^{\mathrm{rev}})$ and, intuitively, we navigate it discovering all left-minimal (y, d)-unique substrings one by one, reporting only those that are minimal. Thus, according to Proposition 3.7, we have to discover all left-minimal $(y, d+1)$-unique substrings before discovering left-minimal (y, d)-unique substrings.

To this end, fixing ℓ, for each node u of $\mathcal{T}(S^{\mathrm{rev}})$, we determine the largest delay d smaller than ℓ such that $L(u)^{\mathrm{rev}}$ can be (y, d)-unique, denoted by $h(u, \ell)$. We consider four different cases:

- If u is a leaf, then $L(u)^{\mathrm{rev}}$ is the j-prefix of S, where $j = n - ln(u) + 1 = |L(u)|$

(a) The nodes are colored according to function g for the character y, for $d = 3$ (dashed) and for $d = 4$ (solid), see Example 3.3.

(b) The nodes are marked according to function g' for $d = 3$, see Example 3.9.

Fig. 2. The suffix tree $\mathcal{T}$ of the reverse string $S^{\text{rev}} = \text{bacabcacaca}$, where $S = \text{acacacbacab}$, see Example 2.1. For clarity, the edges carry the label itself rather than a pair of pointers into the string. Suffix links are drawn as dotted directed edges.

Algorithm 3: ALGO3

---

input : a colored string S, and the suffix tree $\mathcal{T}$ of S^{rev} with suffix links.
output: all triples (T, y, d) such that T is a minimally (y, d)-unique substring of S

1 **for** $d \leftarrow n$ *downto* 0 **do**
2 | ALLCOLORSMINUNIQUE($root, d$)

3 **procedure** ALLCOLORSMINUNIQUE(u, d):
4 | **if** u *is a leaf* **then**   // u is a leaf
5 | | $i \leftarrow ln(u)$
6 | | **if** $i \leq d$ **then**
7 | | | $g'(u) \leftarrow *$
8 | | **else**
9 | | | $g'(u) \leftarrow f^{\text{rev}}(i - d)$
10 | **else**   // u is an inner node
11 | | $X \leftarrow \{$ ALLCOLORSMINUNIQUE(v, y, d) $\mid v$ child of $u\}$
12 | | **if** $X = \{*\}$ **then**
13 | | | $g'(u) \leftarrow *$
14 | | **else**
15 | | | **if** $X = \{y\}$ **or** $X = \{y, *\}$ *with* $y \in \Gamma$ **then**
16 | | | | $g'(u) \leftarrow y$
17 | | | **else**
18 | | | | $g'(u) \leftarrow 0$
19 | **if** $g'(u) = 0$ **then**   // outputting minimal substrings for children
20 | | **for** *each child* v *with* $g'(v) = y \in \Gamma$ **do**
21 | | | **if** *Left-min(v) is defined* **then**
22 | | | | $(v', t) \leftarrow slink(v, sd(u) + 1)$
23 | | | | **if** v' *is not* $(y, d + 1)$-*unique* **then**   // flag from previous round
24 | | | | | output $(Left\text{-}min(v), y, d)$
25 | **return** $g'(u)$

- If $ln(u) < \ell$, then $j + \ell - 1 > n$ thus $L(u)^{\text{rev}}$ is $(y, \ell - 1)$-unique since the position of the color is beyond the end of the string, thus $h(u, \ell) = \ell - 1$.
- If $ln(u) \geq \ell$ and there exists an $i < \ell$ such that $f(j + i) = y$, then the highest possible value $d < \ell$ such that $L(u)^{\text{rev}}$ is (y, d)-unique is given by the position of the furthest occurrence of y within a distance of $\ell - 1$ from j, thus $h(u, \ell) = \max\{i < \ell \mid f(j + i) = y\}$.
- Otherwise, if such i does not exists, we set $h(u, \ell) = -1$.
- If u is an internal node of $\mathcal{T}(S^{\text{rev}})$, then let $k = \min\{h(v, \ell) \mid v$ child of $u\}$, since it is not possible that $L(u)^{\text{rev}}$ is (y, d')-unique, for any $k < d' < \ell$, thus $h(u, \ell) = k$.

When u is an inner node in general, we do not know if $L(u)^{\text{rev}}$ is (y, d)-unique for $d = h(u, \ell)$, unless for all nodes v in the subtree rooted in u, there exists an ℓ_v such that $h(u, \ell) < \ell_v \leq \ell$ and $h(v, \ell_v) = h(u, \ell)$. This is true if $h(v, d + 1) = h(u, \ell)$ for all v.

The definition of $h(u, \ell)$ is as follows:

$$h(u, \ell) = \begin{cases} \ell - 1 & \text{if } u \text{ is a leaf and } ln(u) < \ell, \\ \max\{i < \ell \mid f(n - ln(u) + 1 + i) = y\} & \text{if } u \text{ is a leaf and such } i \text{ exists,} \\ \min\{h(v, \ell) \mid v \text{ child of } u\} & \text{if } u \text{ is an inner node,} \\ -1 & \text{otherwise.} \end{cases}$$

LEMMA 4.1. *Let u be a node of $\mathcal{T}(S^{rev})$, fix d, $h(u, d + 1) = d$ if and only if u is (y, d)-unique.*

PROOF. We first prove that if $h(u, d + 1) = d$ then u is (y, d)-unique. We consider two cases. If u is a leaf, then, by definition of $h(u, d + 1)$, we have that u is (y, d)-unique. If u is an inner node, then $d = \min\{h(v, d + 1) \mid v \text{ child of } u\}$. Since for all nodes v, $h(v, d + 1) \leq d$, then for all children v of u, we have that $h(v, d + 1) = d$. In particular, this holds for all leaves in the subtree rooted in u, thus u is (y, d)-unique.

We first prove that if u is (y, d)-unique then $h(u, d + 1) = d$. We consider again two cases. If u is a leaf, then, by definition of (y, d)-unique, we have that either $ln(u) < d + 1$ or $f(n - ln(u) + 1 + d) = y$. Thus, in both cases, $h(u, d + 1) = d$. If u is an inner node, then, by Lemma 3.1, all leaves in the subtree rooted in u are (y, d)-unique. Thus, for the previous case, for all leaves v in the subtree rooted in u we have that $h(v, d + 1) = d$, thus $h(u, d + 1) = d$. □

To evaluate $h(u, \ell) = \max\{i < \ell \mid f(j + i) = y\}$ when u is a leaf and such i exists, we define a bitvector $b_y[1, 2n]$ such that $b_y[i] = 1$ only if $f(i) = y$ or $i > n$. We preprocess b_y for $O(1)$-time rank and select queries [7]. Given node u with $ln(u) \geq \ell$, let $j = n - ln(u) + 1$. We have that $h(u, \ell) = \max\{\text{select}(b_y, \text{rank}(b_y, j + \ell)) - j, -1\}$.

Example 4.2. In our running example, whose suffix tree is depicted in Figure 2, let us consider the node u corresponding to the substring aca in the string S^{rev}. In order to compute $h(u, 9)$, we have to recursively compute the h function for all children of u. Let v, s, and t be the leaves corresponding to the 9-th, 2-nd, and 7-th suffix of S^{rev}, respectively. If we remove the dollar character from the end of the string S^{rev}, then the 9-th, 2-nd, and 7-th suffix of S^{rev} corresponds to the 3-rd, 10-th, and 8-th prefix of S, respectively. We have that $h(s, 9) = h(t, 9) = 8$, since the furthest possible y at distance smaller than 9 from the 10-th and 8-th prefix of S are beyond the end of the string. While, the furthest possible y at distance smaller than 9 from the 3-rd prefix of S is at distance 5. Thus $h(v, 9) = h(u, 9) = 5$. The intuition is that the highest possible d, smaller than 9 such that the substring aca can be (y, d)-unique cannot be larger than 5, since there is an occurrence of aca that has no y's at distance between 6 and 8.

Let us now compute $h(u, 3)$. We have that $h(s, 3) = 2$, since the furthest possible y at distance smaller than 9 corresponding to the 10-th prefix of S is beyond the end of the string. For the 8-th prefix of S we have that the furthest y at distance smaller than 3 is at distance 1, thus $h(t, 3) = 1$. While, for the leaf v there is no y at distance smaller than 3, thus $h(v, 3) = -1$. Hence, we have that $h(u, 3) = -1$.

We use the $h(u, \ell)$ function in the following way, during the discovery process of all (y, d)-unique substrings of S, provided that we have already discovered all $(y, d + 1)$-unique substrings of S. Let $\ell = d + 1$, for all nodes u of $\mathcal{T}(S^{rev})$ we store the values $h(u, \ell)$. We discover the minimally (y, d)-unique substrings of S, finding all nodes u such that $h(u, \ell) = d$. Among those, the nodes that are also left-minimal are those nodes u such that, $h(parent(u), \ell) < d$. We then check if u is also right-minimal by checking if its suffix-link parent is $(y, d + 1)$-unique, as in Algorithm 2.

The key idea of the skipping algorithm is to keep the values $h(u, \ell)$ updated during the process. Let $H(u)$ be the array that, at the beginning of the discovery of all (y, d)-unique substrings of

S, stores the values $h(u, \ell)$. We want to keep the array H updated in a way such that, after we discovered all (y, d)-unique substrings of S, for all nodes u, $H(u) = h(u, \ell - 1)$. Thus, once we discover that a node u is left-minimal (y, d)-unique, we update the value of $H(u) = h(u, \ell - 1)$. We then update the following values:

- for all nodes v in the subtree rooted in u, we update the values $H(u) = h(u, \ell - 1)$.
- for all nodes p ancestors of u, we update the values $H(p) = \min(H(p), h(u, \ell - 1))$

LEMMA 4.3. *Given $\mathcal{T}(S^{rev})$, fix d, for all nodes u of $\mathcal{T}(S^{rev})$, let $H(u) = h(u, d + 1)$. If for all nodes u such that $H(u) = d$ we (i) set $H(u) = h(u, d)$, and (ii) for all ancestors p of u, set $H(p) = \min\{H(p), h(u, d)\}$, then, for all nodes u of $\mathcal{T}(S^{rev})$, $H(u) = h(u, d)$.*

PROOF. Let $H'(u)$ be the array after all the updates. We now proceed by cases. If u is a leaf and $H(u) = d$, then we set $H(u) = h(u, d)$. Since u, is a leaf, the value $H(u)$ is not modified by any other operation, thus $H'(u) = h(u, d)$. If u is a leaf and $H(u) < d$, then, by definition of $h(u, d + 1)$, we have that $h(u, d) = h(u, d + 1)$. Since u is a leaf and its value $H(u)$ is not modified by ant other operation, $H'(u) = h(u, d)$.
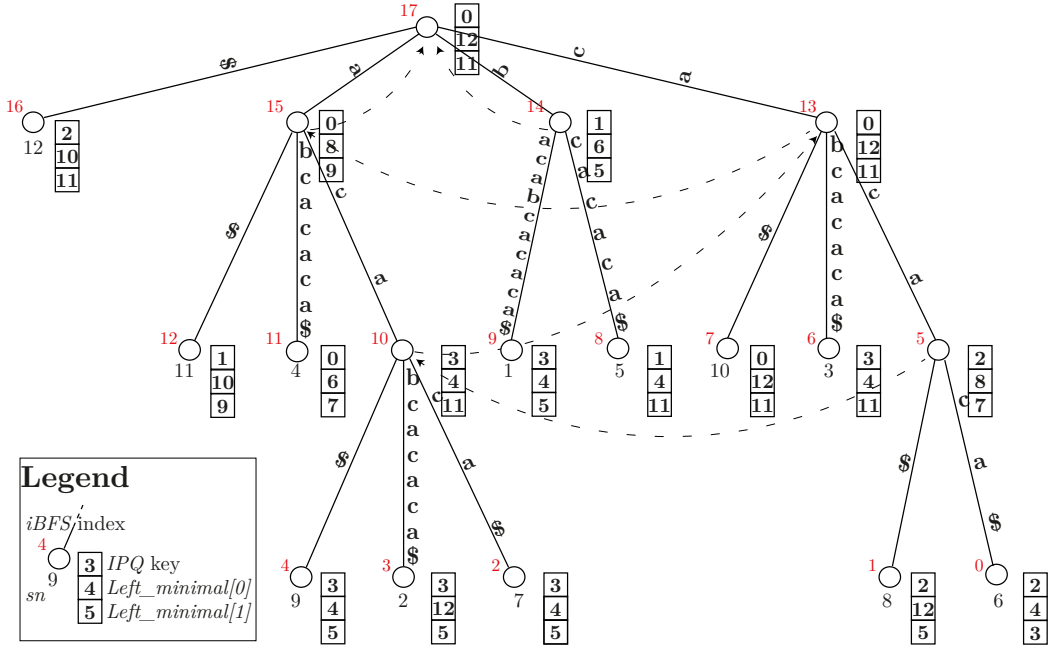
If u is an internal node and $H(u) = d$, then we set $H(u) = h(u, d)$. Moreover, for all nodes v in the subtree rooted in u, we have that $H(v) = d$ and when they perform (ii), they update $H(u) = \min\{H(u), h(v, d)\} = \min\{h(u, d), h(v, d)\} = h(u, d)$ by definition of $h(u, d)$. thus, $H'(u) = h(u, d)$. Finally, if u is an internal node and $H(u) < d$, then if for all nodes v in the subtree rooted in u, $H(v) < d$, we have that $h(u, d + 1) = h(u, d)$. Otherwise, let $\mathcal{L}$ be the set of nodes v in the subtree rooted in u such that $H(v) = d$. Each node $v \in \mathcal{L}$ change the value of $H(u)$ as $\min\{H(u), h(v, d)\}$, thus we have that $H'(u) = \min\{h(v, d) \mid v \text{ leaf in the subtree rooted in } u\} = h(u, d)$. □

In order to efficiently find all nodes u such that $h(u, \ell) = d$ and $h(parent(u), \ell) < d$, we use a *maximum-oriented indexed priority queue*, storing the values of $H(u)$ as keys and $iBFS(u)$ as index. Under this condition, if two nodes have the same key value, then parents have higher priority than their children in IPQ. We keep the priority queue updated using a demote operation while we discover left-minimal nodes and we update the values of the array H stored as keys of IPQ. Algorithm 4 shows how to compute $h(u, \ell)$ for a given node u, and how we update the values of the keys in the IPQ for all children v of u.
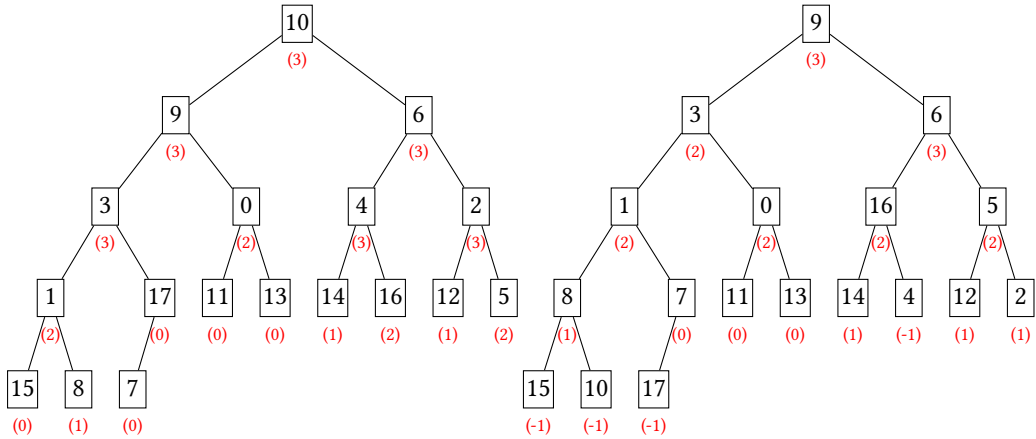
The skipping algorithm summarized in Algorithm 5 initializes priority queue IPQ by inserting all nodes of $\mathcal{T}(S^{rev})$ with key $n + 1$. We then repeat the following as long as there exists a node with non-negative key: extract the max element (u, ℓ) of IPQ; decide whether or not it has to be reported, i.e. if it is right-minimal; apply Algorithm 4 to update the key values of all nodes in the subtree of u and then update the values of the keys of all ancestors of u.

Analysis: For all nodes u in $\mathcal{T}(S^{rev})$, the key value associated to u in IPQ is initially $n + 1$. Each time Algorithm 5 and Algorithm 4 visit a node, the key value of u in IPQ is decreased (via demote()) until it becomes negative. Thus, for each node we perform at most $n + 1$ demote() operations. Since the number of nodes in $\mathcal{T}(S^{rev})$ is linear in n, Algorithm 5 runs in $O(n^2 \log(n))$ time.

Example 4.4. In our running example, we want to report all minimally (y, d)-unique substrings of the colored string, for the character y. Using Figure 3, we now show how we discover that the substring ca is $(y, 3)$-unique. We show in Figure 3b the indexed priority queue after 48 iterations of Algorithm 4. The maximum element in the indexed priority queue IPQ is the node of $\mathcal{T}(S^{rev})$ corresponding to the 10-th node in the reverse index BFS of the tree, as shown in Figure 3a. The associated key value of the maximum element is 3, which means that the corresponding substring is left-minimal $(y, 3)$-unique. In order to decide if the corresponding substring is also right-minimal, we check if the suffix link parent of the node number 10, which is the node number 13, is left-minimal for $d = 4$. The value of the last even value such that the node number 13 has been left-minimal is

(a) The suffix tree of $\mathcal{T}$ of S^{rev}, reporting the values of IPQ, $left\_minimal[0]$, and $left\_minimal[1]$ for each node, after 48 iterations of Algorithm 4.

(b) Indexed priority queue IPQ after 48 iterations of Algorithm 4.

(c) Indexed priority queue IPQ after 49 iterations of Algorithm 4.

Fig. 3. Top (3a): The suffix tree of $\mathcal{T}$ of the reverse string S^{rev} = bacabcacaca, reporting the values of IPQ, $left\_minimal[0]$, and $left\_minimal[1]$ for each node, after 48 iterations of Algorithm 4. In red, in the upper left of each node, we report the reverse index BFS of the node, below each leaf we report the associated suffix number, on the right of the node we report the values of IPQ, $left\_minimal[0]$, and $left\_minimal[1]$. Bottom: The indexed priority queue IPQ after 48 (3b) and 49 (3c) iterations of Algorithm 4. In the nodes of the priority queue we have the index of the nodes of the suffix tree $\mathcal{T}(S^{\text{rev}})$ numbered in the reverse index BFS. Below each node, in red and in brackets, the value of the key associated to each index.

Algorithm 4: HIGHEST POSSIBLE VALUE OF d.

---

    input   : A node u in the suffix tree $\mathcal{T}(S^{\text{rev}})$ and integer ℓ.
    output: Maximum delay $d < \ell$ such that $L(u)^{\text{rev}}$ can be (y, d)-unique.

1 **procedure** $h(u, \ell)$:
2 $min_d \leftarrow \ell - 1$
3   **if** u *is a leaf* **then**
4 $j \leftarrow n - ln(u) + 1$
5 $min_d \leftarrow \max\{\texttt{select}(b_y, \texttt{rank}(b_y, j + \ell)) - j, -1\}$
6   **else**
7   **forall** *children* v *of* u **do**
8 $d = h(v, \ell)$
9   **if** $min_d < d$ **then**
10 $min_d \leftarrow d$
11 $IPQ.\texttt{demote}(u, min_d)$
12   **return** min_d

Algorithm 5: SKIPPING

---

    input   : A colored string S, and a color $y \in \Gamma$
    output: All minimal (y, d)-unique substrings of S.

1 **forall** *nodes* v *of* $\mathcal{T}(S^{rev})$ **do**
2 $IPQ.\texttt{insert}(v, n + 1)$
3 **while** $IPQ.\texttt{allNegative}() = false$ **do**
4 $(u, d) \leftarrow IPQ.\texttt{max}()$
5 $(u', t) = slink(u, sd(parent(u)) + 1)$
6   **if** u' *is not* $(y, d + 1)$*-unique* **then**   // flag from previous round
7   output $(d, \textit{Left-min}(u))$
8 $min_d = h(u, d)$
9   **forall** *ancestors* v *of* u **do**
10   **if** $IPQ.\texttt{keyOf}(v) > min_d$ **then**
11 $IPQ.\texttt{demote}(v, min_d)$

---

set to 12. Thus the node 10 is minimally $(y, 3) - unique$ and has to be reported. We now compute the $h(u, \ell)$ function for the node number 10, u, and $\ell = 3$. As shown in Example 4.2, we have that the h function for the nodes number 2, 3, and 4 are 1, 2, −1. Thus, the value of the h function for the node number 10 is −1. We then update the values of all parents of the node number 10. This results in an update of the values of the indexed priority queue IPQ as reported in Figure 3c.

4.1 Right-minimality check

According to Proposition 3.7, in order to decide if a node is left-minimal (y, d)-unique, we have to check that the suffix link parent $u' = slink(u)$ is not a left-minimal $(y, d + 1)$-unique node. Since we

discover (y, d)-unique substrings in decreasing order of d, it is enough to store, for each node, the previous value of d such that the node is left-minimal.

Given a node u, we store this information in two arrays, indexed by the $iBFS(u)$ values. In one array we store the last even values of d such that the node was left-minimal. In the other array we store the last odd values of d such that the node was left-minimal. This prevents possible overwriting of information, e.g., let $v = slink(u)$ such that v is left-minimal $(y, d + 1)$-unique and left-minimal (y, d)-unique node. Let us assume that v is processed before the node u that is left-minimal (y, d)-unique. If we had only one array holding the information of the last value of d such that a node was left-minimal, then this value for v would now be d, instead of $d + 1$. Thus, we would erroneously conclude that v is also right-minimal, hence that it is minimally (y, d)-unique. Using one array to store even values of d and one array to store odd values of d, we avoid this problem, since v updates the array associated to the parity of d, while u queries the one associated to the parity of $d + 1$.

We can replace lines 6 to 7 with the following lines of code, where we set at the beginning $left\_minimal[b][u] = \infty$ for all $b = \{0, 1\}$ and for all nodes u.

> **6** $report \leftarrow (left\_minimal[(d + 1) \bmod 2][u'] \neq d + 1)$
> **7** **if** $report$ **then**
> **8** $\quad$ output $(d, Left\text{-}min(u))$
> **9** $left\_minimal[d \bmod 2][u] \leftarrow d$

See Figure 3a for an example of the values of the arrays $left\_minimal[0]$ and $left\_minimal[1]$.

5 OUTPUT RESTRICTIONS AND ALGORITHM IMPROVEMENT

We now discuss some practically-minded output restrictions. They can be implemented as a filter to the output, thus discarding some solutions, but if they are considered as part of the problem specification, then they lead to an improvement for the skipping algorithm.

Note that our definition of (y, d)-unique allows that a substring occurs only once, or that none of its occurrences is followed by a y with delay d, because they are all close to the end of string. We now restrict our attention to (y, d)-unique substrings with at least two occurrences followed by y with delay d.

Given a colored string S, let T be minimally (y, d)-unique. We report (T, d) if and only if the following holds:

(1) There are at least two occurrences of T in S.
(2) Let i be the second smallest occurrence of T in S, then $i + |T| - 1 + d \leq n$.

A substring T that satisfies the above conditions is called a *real type* minimally (y, d)-unique substring. Note that any algorithm that computes all minimally (y, d)-unique substrings can be easily modified to output only those that are of real-type, by checking the two conditions before outputting (in line 7 of Algorithm 5, resp. line 17 of Algorithm 2): if the node u is not a leaf and the value of the second largest suffix of S^{rev} in the subtree rooted in u is greater than or equal to d. Since each node u in the suffix tree $\mathcal{T}(S^{\text{rev}})$, corresponds to an interval $[i, j]$ of the suffix array of S^{rev}, we can find the second largest suffix using a range maximum query rMq data structure [13] built on the suffix array of S^{rev} (see Sec 2).

We now turn to the skipping algorithm specifically, which we can modify such that it only computes real-type solutions. The $h(u, \ell)$ function is used in Algorithm 5 in order to find left-minimal nodes in the suffix tree. If we consider the output restrictions as part of the problem, then we do not have to report minimally (y, d)-unique substrings that occur only once, i.e., leaves

in $\mathcal{T}(S^{\mathrm{rev}})$. Then, for all nodes u such that all children of u are leaves, we can directly compute the highest value of $d < \ell$ such that $L(u)^{\mathrm{rev}}$ is (y, d)-unique. This leads to the definition of the *fast_h*(u, ℓ) function for a node u of $\mathcal{T}(S^{\mathrm{rev}})$. The function *fast_h*$(u, \ell)$ is defined similarly to the function $h(u, \ell)$ with the additional following case:

- If all children of u are leaves, we can directly compute the highest value of $d < \ell$ such that $L(u)^{\mathrm{rev}}$ is (y, ℓ)-unique as the largest value $d < \ell$ such that, for each child v of u, $h(v, d+1) = d$. In other words, we are looking for the largest $d < \ell$ such that all children of v are (y, d)-unique.

The definition of *fast_h*(u, ℓ) is as follows:

$$fast\_h(u, \ell) = \begin{cases} \ell - 1 & \text{if } u \text{ is a leaf and } ln(u) < \ell, \\ \max\{i < \ell \mid f^{\mathrm{rev}}(ln(u) - i) = y\} & \text{if } u \text{ is a leaf and such } i \text{ exists}, \\ \max\{i < \ell \mid fast\_h(v, i+1) = i & \text{if all children of } u \text{ are leaves} \\ \qquad \text{for all } v \text{ child of } u\} & \qquad \text{and such } i \text{ exists}, \\ \min\{fast\_h(v, \ell) \mid v \text{ child of } u\} & \text{if } u \text{ is an inner node}, \\ -1 & \text{otherwise.} \end{cases}$$

The additional case of *fast_h*(u, ℓ) can be computed as follows. Let u be a node such that all children of u are leaves. We set $i = \ell$, and compute the values $h(v, i)$ where v is a child of u. We update the value of $i = \min(i, h(v, i) + 1)$, compute the value of $h(v', i)$ where v' is the next child of u, and update the value of $i = \min(i, h(v', i) + 1)$. We continue iterating until all children v of u have the same value $h(v, i)$, possibly -1. Algorithm 6 summarizes these improvements to Algorithm 4. In order to use the *fast_h*(u, ℓ) function in Algorithm 5, it is enough to replace the $h()$ function at line 8 by the *fast_h*$()$ function.

6 EXPERIMENTAL RESULTS

We implemented the algorithms presented in the previous sections and measured their performance on randomly generated datasets and on real-world datasets. The implementation is available online at https://github.com/maxrossi91/colored-strings-miner.

6.1 Setup

Experiments were performed on a 3.4 GHz Intel Core i7-6700 CPU equipped with 8 MiB L3 cache and 16 GiB of DDR4 main memory. The machine had no other significant CPU tasks running, and only a single thread of execution was used.

The OS was Linux (Ubuntu 16.04, 64bit) running kernel 4.4.0. All programs were compiled using g++ version 5.4.0 with `-O3 -DNDEBUG -funroll-loops -msse4.2` options. All given runtimes were recorded with the C++11 `high_resolution_clock` time measurement facility.

6.2 Data

We used two different datasets; the first one consists of randomly generated data, while the second one consists of real-world data.

The randomly generated data are colored strings generated using the C library function `rand()`. We varied the length $n = 100, 1000, 10\,000, 100\,000$, the alphabet size $\sigma = 2, 4, 8, 16, 32$, and the number of colors $\gamma = 2, 4, 8, 16, 32$. In all cases except for $n = 100\,000$, we used seeds 0, 9843, 27 837, 19 341, 29 044; for $n = 100\,000$, we used only seed 0. The string is generated one character (and its color) at a time, i.e. fixing σ and γ, the string of length $n = 1000$ is a prefix of the string $n = 10\,000$. The strings are generated using a uniform distribution of characters and colors. We report only the results of experiments for the values of length $n = 1000, 10\,000, 100\,000$, alphabet size $\sigma = 2, 8, 32$,

---

Algorithm 6: HIGHEST POSSIBLE VALUE OF d.

---

    input : A node u in the suffix tree $\mathcal{T}(S^{\text{rev}})$, and integer ℓ.
    output: Maximum delay $d < \ell$ such that $L(u)^{\text{rev}}$ can be (y, d)-unique.

1 **procedure** $fast\_h(u, \ell)$:
2      $min_d \leftarrow \ell - 1$
3   **if** u *is a leaf* **then**
4          $j \leftarrow n - ln(u) + 1$
5          $min_d \leftarrow \max\{\text{select}(b_y, \text{rank}(b_y, j + \ell)) - j, -1\}$
6   **else if** *all children of* u *are leaves* **then**
7   **repeat**
8             $is\_changed \leftarrow false$
9   **forall** *children* v *of* u **do**
10                $d = fast\_h(v, min_d + 1)$
11   **if** $min_d < d$ **then**
12                   $is\_changed \leftarrow true$
13                   $min_d \leftarrow d$
14   **until** *is_changed AND* $min_d \geq 0$
15   **else**
16   **forall** *children* v *of* u **do**
17             $d = fast\_h(v, \ell)$
18   **if** $min_d < d$ **then**
19                $min_d \leftarrow d$
20      $IPQ.\text{demote}(u, min_d)$
21   **return** min_d

---

number of colors $\gamma = 2, 8, 32$, and seed 0, since these are representative of the trend we observed in all our experiments.

The real-world data is the result of a simulation on a set of established benchmarks in embedded systems verification [4, 8, 30], reported in Table 1. The benchmarks are descriptions of hardware design at the register-transfer level (RTL) of abstraction. Each design consists of a set of primary input bits (PIs) and a set of primary output bits (POs). Primary inputs and primary outputs are grouped into ports. The simulation of designs is a sequence of temporal events which act to capture the effects of the values given as inputs for the design into the design itself, and consequently the effects of the input values on the values assumed by the outputs. We simulated the benchmarks providing as inputs randomly generated sequences using an automatic test pattern generator (ATPG). The result of the simulation is collected in a simulation trace, which stores, for each temporal event, the values of the primary inputs and of the primary outputs. For each simulation event, we consider the values of all primary inputs as characters of the alphabet Σ, and the values of a port of the primary outputs as colors. In other words, for simulation event i, $S[i]$ is the value of the primary inputs, and $f_S(i)$ is the value of the primary outputs.

6.3 Algorithms

We compared the following implementations:

Design	Description	PIs	POs	n	σ	γ	n_y
b03	Resource arbiter [8]	6	4	100 000	17	5	3210
b06	Interrupt handler [8]	4	6	100 000	5	4	44 259
s386	Shynthetized controller [4]	9	7	100 000	129	2	8290
camellia	Symmetric key block cypher [30]	262	131	103 615	70	224	2292
serial	Serial data transmitter	11	2	100 000	118	2	16 353
master	Wishbone bus master [30]	134	135	100 000	417	80	759

Table 1. Real-world datasets used in the experiments. In the column *Design* and *Description* we report the name and the description of the hardware design that we used to generate the simulation trace. In column *PIs* we give the number of primary inputs of the design, while in *POs* that of its primary outputs. In column n we report the length of the simulation trace, and in columns σ and γ the size of the alphabet and the number of colors, respectively. For each design we fixed a color y, and the value n_y refers to the number of y characters in the simulation trace.

- base: the baseline algorithm (Algorithm 2)
- skip: the skipping algorithm (Algorithm 5) using the h function (Algorithm 4)
- real: the skipping algorithm (Algorithm 5) using the $fast\_h$ function (Algorithm 6)
- base-all: the baseline algorithm for all colors (Algorithm 3)

All algorithms report minimally (y, d)-unique substrings only if they are *real type*. We used the sdsl-lite library [19] for compressed suffix trees, range maximum query, and rank and select supports for bit vector implementations.

6.4 Results

We performed all experiments five times and report the average execution time over the five runs. Experimental results are reported in Figures 4 and 5, and Table 2.

Single color analysis. Figure 4 shows the results of the executions of base, skip, and real algorithms over the randomly generated strings data.

We can observe how the algorithms scale

(1) with respect to an increase in the numbers of colors, which has the effect of reducing the number of y-colored characters;
(2) with respect to an increase in the alphabet size; and
(3) with respect to an increase in the length of the text.

We see that all three algorithms behave the same in all cases. Increasing the number of colors (case 1), the running time decreases. Conversely, when the size of the text alphabet increases (case 2), the running time increases also. Finally, we observe a quadratic dependence of the running time on text length (case 3); this is in accordance with our theoretic results (see Sec. 3 and 4).

Figure 4 shows that the skip algorithm is almost always faster than the base algorithm, and that the average speedup is 1.30, with a maximum of 1.75. Moreover, we have that the the real algorithm is almost always faster than the skip algorithm, and the average speedup is 1.25, with a maximum of 1.64. Finally, the average speedup between real and base is 1.65, with a maximum of 2.60 in the case of $N = 100\,000$, $\sigma = 32$ and $\gamma = 32$.

Figure 5 shows the results for base, skip and real algorithms on the real-world dataset. Here, we observe a similar trend to the random data, but the speedup of real with respect to base is much higher — 3.40 on average, with a maximum of 11.88 on the master device. However, on three of the six datasets, base is faster than skip, and faster than real.
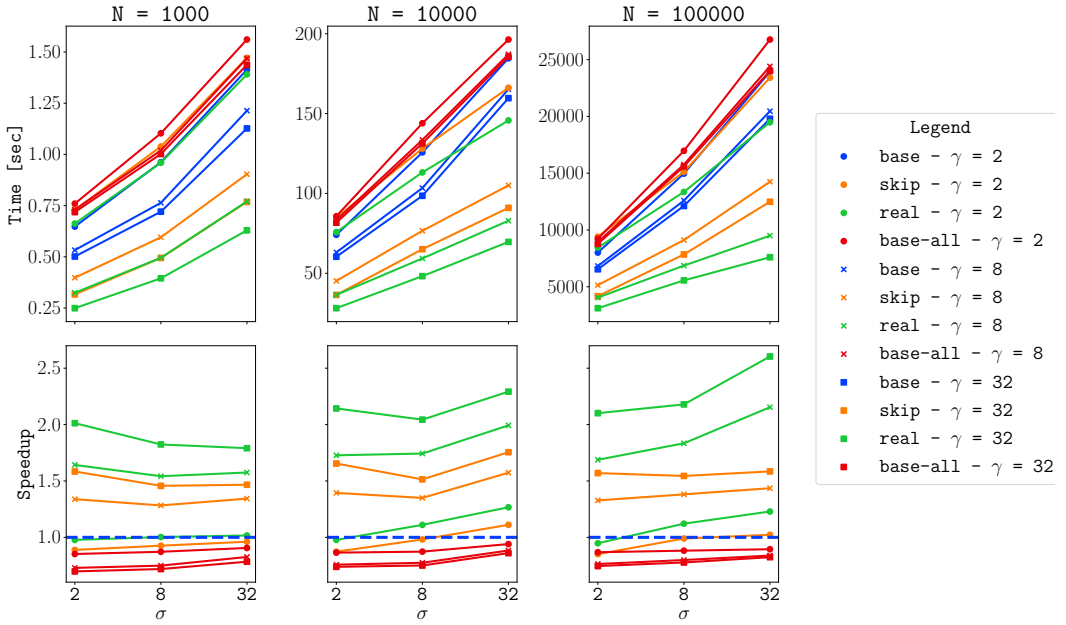
Fig. 4. Results of the execution of algorithms base (color blue), skip (color orange), real (color green), and base-all (color red) over the randomly generated data for $N = 10^3$, 10^4, and 10^5. The x axis represents the values of $\sigma = \{2, 8, 32\}$, and the different markers represents the values of $\gamma = \{2, 8, 32\}$ (circles, crosses, and boxes, respectively). The three plots in the first row report execution times. The plots in the second row report speedups of skip, real, and base-all with respect to algorithm base represented as the dashed blue line at constant 1.0.

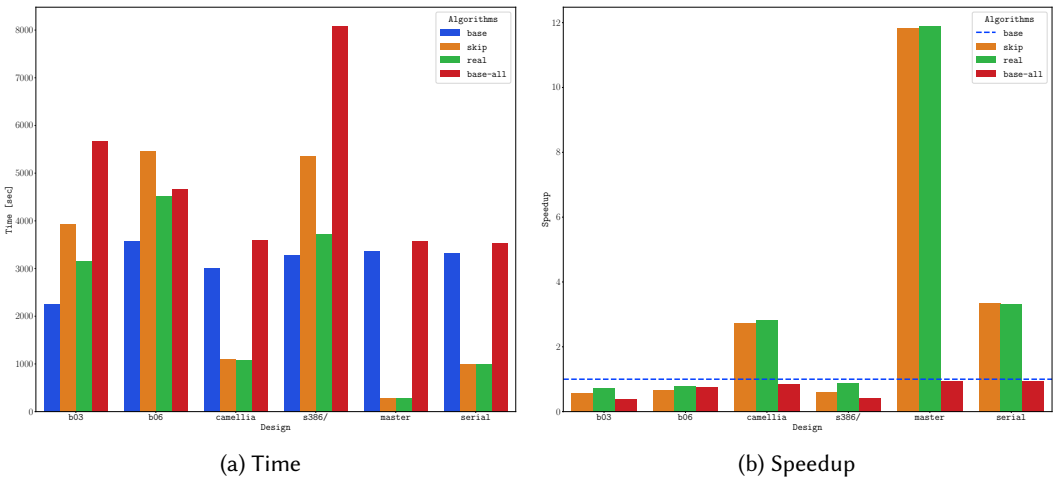(a) Time   (b) Speedup

Fig. 5. Results of the execution of algorithms base, skip, real, and base-all over the real-world dataset. The plot in Figure 5a reports execution times. The plot in Figure 5b reports speedups of skip, real, and base-all with respect to algorithm base represented as the dashed blue line at constant 1.0.

Alphabets		Number of Properties	
σ	γ	base	base-all
2	2	26 894	50 922
	8	1745	15 563
	32	76	3996
8	2	30 219	56 241
	8	1516	12 919
	32	75	3306
32	2	25 120	46 758
	8	1245	10 578
	32	40	2585

(a) Number of properties for N =1000.

Alphabets		Number of Properties	
σ	γ	base	base-all
2	2	2 374 231	4 699 647
	8	187 202	1 447 913
	32	11 767	370 303
8	2	2 844 680	5 607 007
	8	167 431	1 294 765
	32	9989	317 444
32	2	1 466 242	2 892 791
	8	83 320	642 806
	32	4948	156 947

(b) Number of properties for N =10 000.

Alphabets		Number of Properties	
σ	γ	base	base-all
2	2	239 039 415	473 572 454
	8	17 680 770	145 246 888
	32	1 129 991	37 254 203
8	2	279 720 849	552 304 418
	8	15 517 256	127 614 675
	32	947 858	31 264 100
32	2	243 283 926	479 770 368
	8	11 982 556	98 601 898
	32	713 137	23 592 691

(c) Number of properties for N =100 000.

	Number of Properties	
Design	base	base-all
b03	999 191	361 224 140
b06	223 070 824	409 476 680
s386	5 012 263	558 001 254
camellia	77 261	2 470 894
serial	2 085 855	11 653 080
master	252 231	34 812 555

(d) Number of properties for real-world dataset.

Table 2. Number of properties for randomly generated strings and the real-world dataset. The first two columns of 2a, 2a, and 2a report the size of the text alphabet σ and the number of colors γ, while the first column of 2d reports the name of the design where the simulation trace is retrieved. The last two columns report the number of properties extracted from the base and the base-all algorithms, respectively.

All colors analysis. Next, we compare the experimental results of the base and base-all algorithms. The setup is the same as in the previous case, i.e. we performed five runs of each experiment and give the average execution time. The results on the randomly generated and real-world datasets are reported in Figure 4 and 5, respectively.

From Figures 4 and 5, we can observe that the running time of the base algorithm is not heavily affected by the fact that we are looking for a specific color: there is a small increase in running time from base (reporting all real-type minimally (y, d)-unique substrings for any d and just *one* color y), to base-all, reporting all real-type minimally (y, d)-unique substrings for any d and for *all* colors y.

On the real-world data, base outperforms base-all on all six datasets. Note that the number of patterns is considerably larger for base-all.

7   CONCLUSION

We studied pattern discovery problems on colored strings motivated by applications in embedded system verification. To the best of our knowledge this is the first principled algorithmic treatment of these problems.

Colored strings are strings such that each position of the string is assigned a color from a finite set of colors. We studied two different pattern discovery problems on colored strings. The first problem is to find all minimally (y, d)-unique substrings of the colored string, for a given color y and any delay d. We proposed two different approaches, which we refer to as baseline approach and skipping approach. Both algorithms use a suffix tree on the reverse of the colored string as underlying data structure. They discover the patterns starting from the ones with highest delay to the ones with the lowest delay. The two algorithms differ in the way in which minimality information is propagated along the suffix tree. The baseline algorithm traverses the whole tree separately for each delay value, propagating a coloring function from the leaves to the root of the suffix tree. During each traversal, the algorithm goes through all distinct substrings of the text, and uses the coloring function to identify which substrings are minimally (y, d)-unique.

On the other hand, the skipping algorithm stores, for each distinct substring, the next delay value such that the substring is (y, d)-unique, during the discovery process. It uses a maximum-oriented indexed priority queue to find these values and to identify minimally (y, d)-unique substrings.

Even though the theoretical analysis we provided for the skipping algorithms results in a worse upper bound on the running time than for the baseline algorithm, we show in our experiments that it is faster in practice on simulated data, and on half of the real-world data. Even though the skipping algorithm outperforms the baseline algorithm only on half of the real-world datasets (camelia, master, serial), it is significant that the gain on these is considerable, as opposed to only a slight slowdown on the others (b03, b06, s386), see Figure 5. Moreover, the traces on which we see a speedup are derived from devices which perform more complex tasks, thus indicating that our algorithm may be well suited for the application in embedded system verifiaction.

We also proposed a variant of the minimality condition oriented toward real-world application instances. Those conditions allow us to develop a faster core function of the skipping algorithm, resulting in a more effective performance in practice.

The second problem we proposed is to find all minimally (y, d)-unique substrings of the colored string, for *all* colors y concurrently. We modified our baseline algorithm, defining a new coloring function, noting that for fixed d, a substring can be (y, d)-unique for at most one color y. The introduction of the new coloring function and the fact that now all colors y are of interest, increases the running time with respect to the baseline algorithm only negligibly, an effect we observed in the experiments on both randomly generated data and real-world data.

We are currently working with colleagues in embedded systems to integrate these algorithms into their analysis workflows.

REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast algorithms for mining association rules. In *Proc. 20th International Conference on Very Large Data Bases (VLDB)*, Vol. 1215. Morgan Kaufmann, 487–499.

[2] Rakesh Agrawal and Ramakrishnan Srikant. 1995. Mining sequential patterns. In *Proc. Eleventh International Conference on Data Engineering (ICDE)*, Vol. 95. IEEE Computer Society, 3–14.

[3] Fabian Birzele and Stefan Kramer. 2006. A new representation for protein secondary structure prediction based on frequent patterns. *Bioinformatics* 22, 21 (2006), 2628–2634.

[4] Franc Brglez, David Bryan, and Krzysztof Kozminski. 1989. Combinational profiles of sequential benchmark circuits. In *Proc. 1989 IEEE international symposium on circuits and systems (ISACS)*, Vol. 3. IEEE, 1929–1934.

[5] Sarah Chan, Ben Kao, Chi Lap Yip, and Michael Tang. 2003. Mining emerging substrings. In *Proc. Eighth International Conference on Database Systems for Advanced Applications (DASFAA 2003)*. IEEE, 119–126.

[6] Chung-Wen Cho, Ying Zheng, Yi-Hung Wu, and Arbee LP Chen. 2008. A tree-based approach for event prediction using episode rules over event streams. In *Proc. International Conference on Database and Expert Systems Applications (DEXA 2008) (LNCS)*, Vol. 5181. Springer, 225–240.

[7] David Clark. 1997. *Compact PAT trees*. Ph.D. Dissertation. University of Waterloo.

[8] Fulvio Corno, Matteo Sonza Reorda, and Giovanni Squillero. 2000. RT-level ITC'99 benchmarks and first ATPG results. *IEEE Design & Test of computers* 17, 3 (2000), 44–53.

[9] Alessandro Danese, Nicolò Dalla Riva, and Graziano Pravadelli. 2017. A-TEAM: Automatic template-based assertion miner. In *Proc. 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[10] Alessandro Danese, Tara Ghasempouri, and Graziano Pravadelli. 2015. Automatic extraction of assertions from execution traces of behavioural models. In *Proc. 2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 67–72.

[11] Jasbir Dhaliwal, Simon J Puglisi, and Andrew Turpin. 2010. Practical efficient string mining. *IEEE Transactions on Knowledge and Data Engineering* 24, 4 (2010), 735–744.

[12] Lina Fahed, Armelle Brun, and Anne Boyer. 2018. DEER: Distant and Essential Episode Rules for early prediction. *Expert Systems with Applications* 93 (2018), 283–298.

[13] Johannes Fischer and Volker Heun. 2011. Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays. *SIAM J. Comput.* 40, 2 (2011), 465–492.

[14] Johannes Fischer, Volker Heun, and Stefan Kramer. 2005. Fast frequent string mining using suffix arrays. In *Proc. Fifth IEEE International Conference on Data Mining (ICDM 2005)*. IEEE, 609–612.

[15] Johannes Fischer, Volker Heun, and Stefan Kramer. 2006. Optimal string mining under frequency constraints. In *Proc. European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2006) (LNCS)*, Vol. 4213. Springer, 139–150.

[16] Johannes Fischer, Veli Mäkinen, and Niki Välimäki. 2008. Space efficient string mining under frequency constraints. In *Proc. Eighth IEEE International Conference on Data Mining (ICDM 2008)*. IEEE, 193–202.

[17] Harry D Foster, Adam C Krolnik, and David J Lacey. 2004. *Assertion-based design*. Springer Science & Business Media.

[18] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Rage Uday Kiran, Yun Sing Koh, and Rincy Thomas. 2017. A survey of sequential pattern mining. *Data Science and Pattern Recognition* 1, 1 (2017), 54–77.

[19] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. 2014. From Theory to Practice: Plug and Play with Succinct Data Structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014) (LNCS)*, Vol. 8504. Springer, 326–337.

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

[21] Lucas Chi Kwong Hui. 1992. Color Set Size Problem with Application to String Matching. In *Proc. Third Annual Symposium on Combinatorial Pattern Matching (CPM 1992) (LNCS)*, Vol. 644. Springer, 230–243.

[22] Koji Iwanuma, Ryuichi Ishihara, Yoh Takano, and Hidetomo Nabeshima. 2005. Extracting frequent subsequences from a single long data sequence a novel anti-monotonic measure and a simple on-line algorithm. In *Proc. Fifth IEEE International Conference on Data Mining (ICDM 2005)*. IEEE, 186–195.

[23] Xiaonan Ji, James Bailey, and Guozhu Dong. 2007. Mining minimal distinguishing subsequence patterns with gap constraints. *Knowledge and Information Systems* 11, 3 (2007), 259–286.

[24] Srivatsan Laxman, Vikram Tankasali, and Ryen W White. 2008. Stream prediction using a generative model based on frequent episodes in event sequences. In *Proc. 14th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD 2008)*. ACM, 453–461.

[25] Zsuzsanna Lipták, Simon J. Puglisi, and Massimiliano Rossi. 2020. Pattern Discovery in Colored Strings. In *Proceedings of 18th Symposium on Experimental Algorithms, SEA 2020 (Leibniz International Proceedings in Informatics)*, Vol. 160. 12:1–12:14.

[26] Lingyi Liu, David Sheridan, Viraj Athavale, and Shobha Vasudevan. 2011. Automatic generation of assertions from system level design using data mining. In *Proc. Ninth ACM/IEEE International Conference on Formal Methods and Models for Codesign*. IEEE Computer Society, 191–200.

[27] Nizar R Mabroukeh and Christie I Ezeife. 2010. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys (CSUR)* 43, 1 (2010), 3.

[28] Veli Mäkinen, Djamal Belazzougui, Fabio Cunial, and Alexandru I. Tomescu. 2015. *Genome-Scale Algorithm Design*. CUP.

[29] Heikki Mannila, Hannu Toivonen, and A Inkeri Verkamo. 1997. Discovery of frequent episodes in event sequences. *Data mining and knowledge discovery* 1, 3 (1997), 259–289.

[30] OpenCores. [n. d.]. Available at https://opencores.org/. Accessed 05-03-2019.

[31] Tinghai Pang, Lei Duan, Jesse Li-Ling, and Guozhu Dong. 2017. Mining Similarity-Aware Distinguishing Sequential Patterns from Biomedical Sequences. In *Proc. Second International Conference on Data Science in Cyberspace (DSC 2017)*. IEEE, 43–52.

[32] Jian Pei, Jiawei Han, and Wei Wang. 2007. Constraint-based sequential pattern mining: the pattern-growth methods. *Journal of Intelligent Information Systems* 28, 2 (2007), 133–160.

[33] Atif Raza and Stefan Kramer. 2019. Accelerating pattern-based time series classification: a linear time and space string mining approach. *Knowledge and Information Systems* (2019), 1–29.

[34] Robert Sedgewick and Kevin Wayne. 2011. *Algorithms*. Addison-Wesley Professional.

[35] Bill Smyth. 2003. *Computing Patterns in Strings*. Pearson Addison-Wesley, Essex, England.

[36] Niko Välimäki and Simon J Puglisi. 2012. Distributed string mining for high-throughput sequencing data. In *Proc. 12th International Workshop on Algorithms in Bioinformatics (WABI 2012) (LNCS)*, Vol. 7534. Springer, 441–452.

[37] Shobha Vasudevan, David Sheridan, Sanjay Patel, David Tcheng, Bill Tuohy, and Daniel Johnson. 2010. Goldmine: Automatic assertion generation using data mining and static analysis. In *Proc. 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 626–629.

[38] Xianming Wang, Lei Duan, Guozhu Dong, Zhonghua Yu, and Changjie Tang. 2014. Efficient mining of density-aware distinguishing sequential patterns with gap constraints. In *International Conference on Database Systems for Advanced Applications (DASFAA 2014) (LNCS)*, Vol. 8421. Springer, 372–387.

[39] Xindong Wu, Xingquan Zhu, Yu He, and Abdullah N Arslan. 2013. PMBC: Pattern mining from biological sequences with wildcard constraints. *Computers in Biology and Medicine* 43, 5 (2013), 481–492.