Efficiently Testing Simon's Congruence

Paweł Gawrychowski
Faculty of Mathematics and Computer Science, University of Wrocław, Poland

Maria Kosche
Computer Science Department, Universität Göttingen, Germany

Tore Koß
Computer Science Department, Universität Göttingen, Germany

Florin Manea
Computer Science Department, Universität Göttingen, Germany
Campus-Institut Data Science, Göttingen, Germany

Stefan Siemer
Computer Science Department, Universität Göttingen, Germany

Abstract

Simon's congruence $\sim_k$ is a relation on words defined by Imre Simon in the 1970s and intensely studied since then. This congruence was initially used in connection to piecewise testable languages, but also found many applications in, e.g., learning theory, databases theory, or linguistics. The $\sim_k$-relation is defined as follows: two words are $\sim_k$-congruent if they have the same set of subsequences of length at most k. A long standing open problem, stated already by Simon in his initial works on this topic, was to design an algorithm which computes, given two words s and t, the largest k for which $s \sim_k t$. We propose the first algorithm solving this problem in linear time $O(|s| + |t|)$ when the input words are over the integer alphabet $\{1, \dots, |s| + |t|\}$ (or other alphabets which can be sorted in linear time). Our approach can be extended to an optimal algorithm in the case of general alphabets as well.

To achieve these results, we introduce a novel data-structure, called Simon-Tree, which allows us to construct a natural representation of the equivalence classes induced by $\sim_k$ on the set of suffixes of a word, for all $k \geq 1$. We show that such a tree can be constructed for an input word in linear time. Then, when working with two words s and t, we compute their respective Simon-Trees and efficiently build a correspondence between the nodes of these trees. This correspondence, which can also be constructed in linear time $O(|s| + |t|)$, allows us to retrieve the largest k for which $s \sim_k t$.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory; Theory of computation → Design and analysis of algorithms

Keywords and phrases Simon's congruence, Subsequence, Scattered factor, Efficient algorithms

Digital Object Identifier 10.4230/LIPIcs.STACS.2021.34

Related Version *Full Version*: `https://arxiv.org/abs/2005.01112`

Funding The work of the four authors from Göttingen was supported by the DFG-grant 389613931 *Kombinatorik der Wortmorphismen*.

1 Introduction

A subsequence of a word w (also called scattered factor or subword, especially in automata and language theory) is a word u such that there exist (possibly empty) words $v_1, \dots, v_{\ell+1}$, $u_1, \dots, u_\ell$ with $u = u_1 \dots u_n$ and $w = v_1 u_1 v_2 u_2 \dots v_\ell u_\ell v_{\ell+1}$. Intuitively, the subsequences of a word w are exactly those words obtained by deleting some of the letters of w, so, in a sense, they can be seen as lossy-representations of the word w. Accordingly, subsequences may be a natural mathematical model for situations where one has to deal with input strings with missing or erroneous symbols sequencing, such as processing DNA data or digital signals [27]. Due to this very simple and intuitive definition as well as the apparently large potential for applications, there is a high interest in understanding the fundamental properties that can be derived in connection to the sets of subsequences of words. This is reflected in the consistent literature developed around this topic. J. Sakarovitch and I. Simon in [21, Chapter 6] overview some of the most important combinatorial and language theoretic properties of sets of subsequences. The theory of subsequences was further developed in various directions, such as combinatorics on words, automata theory, formal verification, or string algorithms. For instance, subword histories and Parikh matrices (see, e.g., [23, 26, 28]) are algebraic structures in which the number of specific subsequences occurring in a word are stored and used to define combinatorial properties of words. Strongly related, the binomial complexity of words is a measure of the multiset of subsequences that occur in a word, where each occurrence of such a factor is considered as an element of the respective multiset; combinatorial and algorithmic results related to this topic are obtained in, e.g., [25, 9, 20, 19], and the references therein. Further, in [35, 12, 18] various logic-theories were developed, starting from the subsequence-relation, and analysed mostly with automata theory and formal verification tools. Last, but not least, many classical problems in the area of string algorithms are related to subsequences: the longest common subsequence and the shortest common supersequence problems [22, 1, 3, 4], or the string-to-string correction problem [34]. Several algorithmic problems connected to subsequence-combinatorics are approached and (partially) solved in [7].

One particularly interesting notion related to subsequences was introduced by Simon in [30]. He defined the relation $\sim_k$ (called now Simon's congruence) as follows. Two words are $\sim_k$-congruent if they have the same set of subsequences of length at most k. In [30], as well as in [21, Chapter 6], many fundamental properties (mainly of combinatorial nature) of $\sim_k$ are discussed; this line of research was continued in, e.g., [15, 16, 17, 6, 2] where the focus was on the properties of some special classes of this equivalence. From an algorithmic point of view, a natural decision problem and its optimisation variant stand out:

▶ **Problem 1.** SIMK*: Given two words s and t over an alphabet Σ, with $|s| = n$ and $|t| = n'$, with $n \geq n'$, and a natural number k, decide whether $s \sim_k t$.*

▶ **Problem 2.** MAXSIMK*: Given two words s and t over an alphabet Σ, with $|s| = n$ and $|t| = n'$, with $n \geq n'$, find the maximum k for which $s \sim_k t$.*

The problems above were usually considered assuming (although not always explicitly) the Word RAM model with words of logarithmic size. This is a standard computational model in algorithm design in which, for an input of size n, the memory consists of memory-words consisting of $\Theta(\log n)$ bits. Basic operations (including arithmetic and bitwise Boolean operations) on memory-words take constant time, and any memory-word can be accessed in constant time. In this model, the two input words are just sequences of integers, each integer stored in a single memory-word. Without losing generality, we can assume the alphabet to be simply $\{1, \ldots, n + n'\}$ by sorting and renaming the letters occurring in the input in linear time. For a detailed discussion on the computational model, see the full version of this paper, available on arXiv [24].

Due to the central role played by the $\sim_k$-congruence in the study of piecewise testable languages, as well as in the many other areas mentioned above, both problems SIMK and MAXSIMK were considered highly interesting and studied thoroughly in the literature.

In particular, Hebrard [13] presents MAXSIMK as computing a similarity measure between strings and mentions a solution of Simon [29] for MAXSIMK which runs in $O(|\Sigma|nn')$ (the same solution is mentioned in [11]). He goes on and improves this (see [13]) in the case when

Σ is a binary alphabet: given two bitstrings s and t, one can find the maximum k for which $s \sim_k t$ in linear time. The problem of finding optimal algorithms for MaxSimK, or even SimK, for general alphabets was left open in [29, 13] as the methods used in the latter paper for binary strings did not seem to scale up. In [11], Garel considers MaxSimK and presents an algorithm based on finite automata, running in $O(|\Sigma|n)$, which computes all *distinguishing words* u of minimum length, i.e., words which are factors of only one of the words s and t from the problem's statement. Several further improvements on the aforementioned results were reported in [5, 32]. Finally, in an extended abstract from 2003 [31], Simon presented another algorithm based on finite automata solving MaxSimK which runs in $O(|\Sigma|n)$, and he conjectures that it can be implemented in $O(|\Sigma| + n)$. Unfortunately, the last claim is only vaguely and insufficiently substantiated, and obtaining an algorithm with the claimed complexity seems to be non-trivial. Although Simon announced that a detailed description of this algorithm will follow shortly, we were not able to find it in the literature.

In [8], a new approach to efficiently solving SimK was introduced. This idea was to compute, for the two given words s and t and the given number k, their shortlex forms: the words which have the same set of subsequences of length at most k as s and t, respectively, and are also lexicographically smallest among all words with the respective property. Clearly, $s \sim_k t$ if and only if the shortlex forms of s and t for k coincide. The shortlex form of a word s of length n over Σ was computed in $O(|\Sigma|n)$ time, so SimK was also solved in $O(|\Sigma|n)$. A more efficient implementation of the ideas introduced in [8] was presented in [2]: the shortlex form of a word of length n over Σ can be computed in linear time $O(n)$, so SimK can be solved in optimal linear time. By binary searching for the smallest k, this gives an $O(n \log n)$ time solution for MaxSimK. This brings up the challenge of designing an optimal linear-time algorithm for non-binary alphabets.

Our results. In this paper we confirm Simon's claim from 2003 [31]. We present a complete algorithm solving MaxSimK in linear time on Word RAM with words of size $\Theta(\log n)$. This closes the problem of finding an optimal algorithm for MaxSimK. Our approach is not based on finite automata (as the one suggested by Simon), nor on the ideas from [8, 2]. Instead, it works as follows. Firstly, looking at a single word, we partition the respective word into k-blocks: contiguous intervals of positions inside the word, such that all suffixes of the word inside the same block have exactly the same subsequences of length at most k (i.e., they are $\sim_k$-equivalent). Since the partition in $(k+1)$-blocks refines the partition in k-blocks, one can introduce the *Simon-Tree* associated to a word: its nodes are the k-blocks (for k from 1 to at most n), and each node on level k has as children exactly the $(k+1)$-blocks in which it is partitioned. We first show how to compute efficiently the Simon-Tree of a word. Then, to solve MaxSimK, we show that one can maintain in linear time a connection between the nodes on the same levels of the Simon-Trees associated to the two input words. More precisely, for all ℓ, we connect two nodes on level ℓ of the two trees if the suffixes starting in those blocks, in their respective words, have exactly the same subsequences of length at most ℓ. It follows that the value k required in MaxSimK is the lowest level of the trees on which the blocks containing the first position of the respective input words are connected. Using the Simon-Trees of the two words and the connection between their nodes, we can also compute in linear time a distinguishing word of minimal length for s and t. Achieving the desired complexities is based on a series of combinatorial properties of the Simon-Trees, as well as on a rather involved data structures toolbox.

Our paper is structured as follows: we firstly introduce the basic combinatorial and data structures notions, then we show how Simon-Trees are constructed efficiently, and, finally, we show how MaxSimK is solved by connecting the Simon-Trees of the two input words. We end this paper with a series of concluding remarks, extensions, and further research questions. The formal proofs of our results are given in the full version of this paper [24], due to space constraints; most statements are, however, accompanied by explanations substantiating our claims. Similarly, for space reasons, a discussion on how our results can be extended to an optimal algorithm for MaxSimK in the case of input words over general alphabets is also given in the paper's full version [24].

2 Preliminaries

Let $\mathbb{N}$ be the set of natural numbers, including 0. An alphabet Σ is a nonempty finite set of symbols called *letters*. A *word* is a finite sequence of letters from Σ, thus an element of the free monoid Σ^*. Let $\Sigma^+ = \Sigma^* \backslash \{\varepsilon\}$, where ε is the empty word. The *length* of a word $w \in \Sigma^*$ is denoted by $|w|$. The i^{th} letter of $w \in \Sigma^*$ is denoted by $w[i]$, for $i \in [1 : |w|]$. For $m, n \in \mathbb{N}$, we let $[m : n] = \{m, m+1, \ldots, n\}$ and $w[m : n] = w[m]w[m+1] \ldots w[n]$.

A word $u \in \Sigma^*$ is a *factor* of $w \in \Sigma^*$ if $w = xuy$ for some $x, y \in \Sigma^*$. If $x = \varepsilon$ (resp. $y = \varepsilon$), u is called a *prefix* (resp. *suffix* of w). For some $x \in \Sigma$ and $w \in \Sigma^*$, let $|w|_x = |\{i \in [1 : |w|] \mid w[i] = x\}|$ and $\mathrm{alph}(w) = \{x \in \Sigma \mid |w|_x > 0\}$ for $w \in \Sigma^*$; in other words, $\mathrm{alph}(w)$ denotes the smallest subset $S \subset \Sigma$ such that $w \in S^*$.

▶ **Definition 1.** *We call u a subsequence of length k of w, where $|w| = n$, if there exist positions $1 \leq i_1 < i_2 < \ldots < i_k \leq n$, such that $u = w[i_1]w[i_2] \cdots w[i_k]$. Let $\mathrm{Subseq}_{\leq k}(i, w)$ denote the set of subsequences of length at most k of $w[i : n]$. Accordingly, the set of subsequences of length at most k of the entire word w will be denoted by $\mathrm{Subseq}_{\leq k}(1, w)$.*

Equivalently, $u = u_1 \ldots u_\ell$ is a subsequence of w if there exist $v_1, \ldots, v_{\ell+1} \in \Sigma^*$ such that $w = v_1 u_1 \ldots v_\ell u_\ell v_{\ell+1}$. For $k \in \mathbb{N}$, $\mathrm{Subseq}_{\leq k}(1, w)$ is called the full k-spectrum of w.

▶ **Definition 2** (Simon's Congruence).

(i) *Let $w, w' \in \Sigma^*$. We say that w and w' are equivalent under Simon's congruence $\sim_k$ (or, alternatively, that w and w' are k-equivalent) if the set of subsequences of length at most k of w equals the set of subsequences of length at most k of w', i.e., $\mathrm{Subseq}_{\leq k}(1, w) = \mathrm{Subseq}_{\leq k}(1, w')$.*

(ii) *Let $i, j \in [1 : |w|]$. We define $i \sim_k j$ (w.r.t. w) if $w[i : n] \sim_k w[j : n]$, and we say that the positions i and j are k-equivalent.*

(iii) *A word u of length k distinguishes w and w' w.r.t. $\sim_k$ if u occurs in exactly one of the sets $\mathrm{Subseq}_{\leq k}(1, w)$ and $\mathrm{Subseq}_{\leq k}(1, w')$.*

Following the discussion from the introduction, for our algorithmic results we assume the Word RAM model with words of size $\Theta(\log n)$.

We start by recalling two data structures which play an important role in our results. These are the interval split-find and interval union-find data structures. Their formal definition is given in the full version of this paper [24]. Rather informally, in the union-find (respectively, split-find) structure we maintain a partition of an interval (also called universe) $V = [1 : n]$ in sub-intervals, under two operations: `union` of adjacent intervals (respectively, `split` an interval in two sub-intervals around an element of the interval), and `find` the representative of the interval containing a given value. In our algorithms, when using these structures, we usually describe the intervals stored initially in the structure, and then the `unions` (respectively, the `splits`) which are made, as well as the `find` operations, without going into the formalism behind these operations. In usual implementations of these structures the representative of each interval, which is returned by `find`, is its maximum; we can easily enhance the data structures so that the `find` operation returns both borders of the interval containing the searched value. The following lemma was shown in [10, 14].

▶ **Lemma 3.** *One can implement the interval split-find (respectively, union-find) data structures, such that, the initialisation of the structures followed by a sequence of $m \in O(n)$* `split` *(respectively,* `union`*) and* `find` *operations can be executed in $\mathcal{O}(n)$ time and space.*

3 Constructing the Simon-Tree of a word

In this section, we introduce a new data structure, which is fundamental to our approach – the *Simon-Tree*. The Simon-Tree is used as a representation for the equivalence classes in a word, which are explained in Section 3.1. The definition of Simon-Trees is then given in Section 3.2, and the construction is described in Section 3.3.

3.1 Equivalence classes of a Word

In this section, we develop a method to efficiently partition the positions of a given word w, of length n, into equivalence classes w.r.t. $\sim_k$, such that all suffixes starting with positions of the same class have the same set of subsequences of length at most k. As in this section we only deal with one input word w of length n, we will sometimes omit the reference to this word in our notation: e.g. $\mathrm{Subseq}_k(i) = \mathrm{Subseq}_k(i, w)$; in the case of such omissions, the reader may safely assume that we are referring to the aforementioned input word.

Firstly, we will examine the equivalence classes that each congruence relation $\sim_k$ induces on the set of suffixes of w for all k. Let $1 \leq i < j \leq n$, then $w[j : n]$ is a suffix of $w[i : n]$, hence $\mathrm{Subseq}_k(i) \supseteq \mathrm{Subseq}_k(j)$ holds for all $k \in \mathbb{N}$. For any $l \in [i : j]$ we obtain $\mathrm{Subseq}_k(i) \supseteq \mathrm{Subseq}_k(l) \supseteq \mathrm{Subseq}_k(j)$. If we additionally let $i \sim_k j$, then the sets of subsequences corresponding to i and j respectively are equal, so $\mathrm{Subseq}_k(i) = \mathrm{Subseq}_k(l) = \mathrm{Subseq}_k(j)$ and $i \sim_k l \sim_k j$. Hence, the equivalence classes of the set of suffixes of w w.r.t. $\sim_k$ correspond to sets of consecutive indices (i.e., intervals) in $[1 : |w|]$, namely the starting positions of the suffixes in each class. We call these classes *k-blocks.*

A k-block consisting only of a single position (i.e., it is a *singleton-k-block*), remains an ℓ-block for all $\ell > k$. For a k-block $b = [m_b : n_b]$, m_b is its starting position and n_b its ending position. For the ending position of a $k-$block we also use the following definition.

▶ **Definition 4.** *For some $k > 0$, if $i \sim_{k-1} i+1$ and $i \nsim_k i+1$, then we will say that i splits its $(k-1)$-block or that i is a k-splitting position.*

If $a = [m_a : n_a]$ is a k-block and $b = [m_b : n_b]$ is a $(k+1)$-block with $m_a \leq m_b \leq n_b \leq n_a$, then we say that b is a $(k+1)-$block in a (alternatively, of a).

Since, for $1 \leq i, j \leq n$, $i \sim_k j$ holds if $i \sim_{k+1} j$, the relation $\sim_{k+1}$ is a refinement of $\sim_k$. In our setting, this means that the $(k+1)-$blocks of w are obtained by partitioning the k-blocks of w into subintervals. To obtain a partition of the positions of w into equivalence classes and the corresponding blocks, we can use this refinement property. We get the following inductive procedure.

0-blocks. For any i, with $1 \leq i \leq n$, we have $\mathrm{Subseq}_0(i) = \{\varepsilon\}$. Thus, we refer to $[1 : n]$ as the 0-*block* of w. Note, however, that position n cannot be referred to as a 0-splitting position.

$(k+1)$-blocks. For $k \geq 0$ and a k-block $a = [m_a : n_a]$ in w with $|a| \geq 2$, we can find the $(k+1)$-splitting positions inside of a. Except for the case when a is a 0-block, position n_a marks a k-splitting position. So, if $k > 0$, we slice off position n_a and obtain a truncated block $a^\dagger$; if $k = 0$, then $a^\dagger = a$. Going from right to left through $a^\dagger$, the position of every character we encounter for the first time (so, which we have not seen before in this

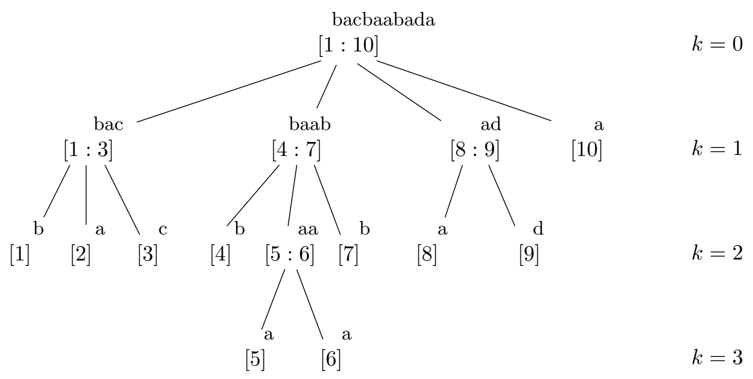

Figure 1 Simon-Tree of the word *bacbaabada*. Above each block $[i : j]$ we wrote the word $w[i : j]$.

traversal of $a^\dagger$) is a splitting position of $a^\dagger$. Consequently, those splitting positions and n_a (only if $k > 0$) will split a into $(k+1)$-blocks. The correctness of this approach follows from Lemma 5.

▶ **Lemma 5.** *Let $a = [m_a, n_a]$ be a k-block with $1 \le m_a < n_a \le n$. Let $a^\dagger = a$ for $k = 0$ and $a^\dagger = [m_a : n_a - 1]$ for $k > 0$. Then the following holds for all $i, j \in a^\dagger$:*
(i) *if $k > 0$ then $i \nsim_{k+1} n_a$;*
(ii) *$i \sim_{k+1} j$ if and only if $\mathrm{Subseq}_1(i, a^\dagger) = \mathrm{Subseq}_1(j, a^\dagger)$.*

3.2 Simon-Tree definition

Before introducing the Simon-Tree, we recall some basic notions. An ordered rooted tree is a rooted tree which has a specified order for the subtrees of a node. We say that the depth of a node is the length of the unique simple path from the root to that node. Generally, the nodes with smaller depth are said to be *higher* (the root is the highest node with depth 0), while the nodes with greater depth are *lower* in the tree.

We can now define a new data structure called *Simon-Tree*. The Simon-Tree of a word w gives us a hierarchical representation of the equivalence classes inside of w. While an example of a Simon-Tree can be seen in Figure 1, the formal definition of a Simon-Tree is as follows.

▶ **Definition 6.** *The* Simon-Tree *T_w associated to the word w, with $|w| = n$, is an ordered rooted tree. The nodes of depth k represent $k-$blocks of w, for $0 \le k \le n$, and are defined recursively.*

- *The root corresponds to the 0-block of the word w, i.e., the interval $[1 : n]$.*
- *For $k > 1$ and for a node a of depth $k-1$, which represents a $(k-1)$-block $[i : j]$ with $i < j$, the children of a are exactly the blocks of the partition of $[i : j]$ in k-blocks, ordered decreasingly (right-to-left) by their starting position.*
- *For $k > 1$, each node of depth $k-1$ which represents a singleton-$(k-1)$-block is a leaf.*

The nodes of depth k in a tree T_w are called *explicit k-nodes (or simply k-nodes)*; by abuse of notation, we identify each k-node by the k-block it represents.

With respect to their starting positions in the word, we number the children nodes (which are blocks) of a node b from right to left. That is, the i^{th} child of b is the i^{th} block of the partition of a, counted from right to left. The singleton-j-blocks, for $j < k$, are also k-blocks, but they do not appear explicitly as nodes of depth k in the tree T_w. We will say that they are *implicit k-nodes*. In other words, an explicit singleton-j-node is an implicit k-node, for all $k > j$, and the only child of a k-node $[i : i]$ is the implicit $(k + 1)$-node $[i : i]$. The nodes of depth k in the Simon-Tree T_w do not necessarily comprise all the k-blocks of w, but they contain explicitly exactly those k-blocks of w that were obtained by non-trivially splitting a $(k - 1)$-block of w which was not a singleton.

3.3 Simon-Tree construction

We are interested in constructing the Simon-Tree T_w associated to a word w, with $|w| = n$, in linear time. In this section we give a description of the construction algorithm and its analysis. The corresponding pseudocode can be seen in Algorithms 1–3.

For the algorithms, we use the array X of size n which holds, for a given position i, the next position of $w[i]$ in the word $w[i + 1 : n]$. We formally define this with $X[i] = \min\{j \mid w[j] = w[i], j > i\}$, while we assume $X[i] = \infty$ if $w[i] \notin \mathrm{alph}(w[i + 1 : n])$. The array can be calculated in $\mathcal{O}(n)$ time and space as stated in the full version of this paper [24]. As an example consider now the word $w =$ `bacbaabada`. The array X is then depicted as follows.

i	1	2	3	4	5	6	7	8	9	10	11
$w[i]$	b	a	c	b	a	a	b	a	d	a	\$
$X[i]$	4	5	∞	7	6	8	∞	10	∞	∞	∞

When applying our algorithm to w, we get the tree shown in Figure 1, where we represent each node with the block $[i : j]$ it represents accompanied by the word $w[i : j]$.

Algorithm 1 Building the Simon-Tree T_w for a word w.

```
   Input: Word $w$ with $|w| = n$
   Result: Simon-Tree $T_w$
1  $w' \leftarrow w\$$;
2  Let $T$ be the tree with the root associated to the block $[? : n + 1]$ of $w'$;
3  Let $p$ be a pointer to the root of $T$;
4  Compute the array $X[i]$;
5  for $i$=$n$ to 1 do
6      $a \leftarrow$ findNode($i$,$T$,$p$);
7      $(T, p) \leftarrow$ splitNode($i$,$T$,$a$);
8  end
9  Set starting position for all blocks from leftmost branch including the root to 1;
10 Remove $-letter from tree: Remove the node associated to $[n + 1 : n + 1]$ from $T$ and
     set all right ends $r$ of blocks on the rightmost branch to $n$;
11 return T ;
```

Algorithm description. In general, we consider the individual letters of the word w from right to left. After considering $w[i]$, the tree we constructed so far corresponds to the Simon-Tree of the suffix $w[i : n]$. By traversing the word from right to left, we also construct the Simon-Tree in a right-to-left manner. Accordingly, it holds that at each time step only the nodes on the leftmost branch of the tree are possible to be enhanced. This is because for the tree of the word $w[i + 1 : n]$, prepending a new letter to the word $w[i + 1 : n]$ can only affect the leftmost node/block on each level of the tree, as the nodes of level k store the

Algorithm 2 findNode.

```
Input: Position $i$ in $w$, Simon-Tree $T_{w[i+1:n]\$}$, Pointer to leftmost leaf $a$ of $T_{w[i+1:n]\$}$
Result: Pointer to node on leftmost branch of $T_{w[i+1:n]\$}$
1  while $a$ is not the root of $T_{w[i+1:n]\$}$ do
2      $r \leftarrow$ ending position of the block represented by $a$;
3      $r_p \leftarrow$ ending position of block represented by $parent(a)$;
4      if $X[i] \geq r$ and $X[i] < r_p$ then
5          return $a$;
6      else
7          Close the block represented by $a$: Set its starting position to $i+1$;
8          $a \leftarrow parent(a)$;
9      end
10 end
11 return $a$;
```

Algorithm 3 splitNode.

```
Input: Position $i$, Simon-Tree $T_{w[i+1:n]\$}$, Pointer to node $a$ on leftmost branch of
       $T_{w[i+1:n]\$}$
Result: Simon-Tree $T_{w[i:n]\$}$, Pointer to leftmost leaf of $T_{w[i:n]\$}$
1  $T \leftarrow T_{w[i+1:n]\$}$ ;
2  if $a$ is the leftmost leaf of $T$ then
       // node $a$ represents the open block $[? : i+1]$
3      Add a child to $a$ associated to the now completed block $[i+1 : i+1]$;
4      Add a leftmost child $b$ to $a$ associated to the open block $[? : i]$;
5  else
6      Add a leftmost child $b$ to $a$ associated to the open block $[? : i]$;
7  end
8  return ($T$, pointer to $b$)
```

k-blocks, and, accordingly, build a (possibly intermittent, if we only consider the explicit nodes) partition of the word w into non-overlapping intervals, for all k, while the nodes of one level are ordered with regard to their position in the word.

This means that a newly considered position of our word can be only added to a node on the leftmost branch of the tree that was constructed so far during the application of the algorithm. Therefore, we call the nodes on the leftmost branch *open blocks*. These open blocks are not complete and have a yet unknown starting position. We use $[? : i]$ to denote the open block with unknown starting position and ending position i. For the nodes on the leftmost branch, we only store the ending position (or splitting position) of their represented block. For all nodes that are not on the leftmost branch in the tree, we store both starting and ending position of their represented block.

In the beginning of the construction algorithm, we append the letter \$ at the end of w to ensure that all the positions of w are treated in a uniform way. More precisely, the usage of the \$-letter allows us to uniformly find the splitting points in a block according to case (ii) of Lemma 5 only. That is, by adding the letter \$ at the end, we avoid position n as being falsely recognized as a 0-splitting position since it is the ending position of the 0-block $[1:n]$ of w. As seen in Algorithm 1, we define $w' = w\$$ and start the algorithm with the tree that only has one node, the root, representing the open block $[?:n+1]$.

When considering a new position i of the word, and, essentially, inserting it into the current tree, we want to find the correct tree level where position i would mark the splitting of a new block or a new node, respectively. According to Algorithm 2, by starting at the leftmost leaf (which is the node associated to the open block $[?:i+1]$) and going up the leftmost branch of the current tree, we look for the first node where the character $w[i]$ occurs on a non-ending position. Let this be node A of depth k, representing an open k-block. Node A cannot be a leaf since leafs only represent singleton-blocks, consist therefore only of one position, and $w[i]$ could not occur on a non-ending position. Let the leftmost child of A be the node a of depth $k+1$. By utilizing Lemma 5, we get the information that position i is a $(k+2)$-splitting position in a, and consequently, our new block with ending position i is mapped to level $(k+2)$ of the Simon-Tree. Following Algorithm 3, we then insert the new block $[?:i]$ in the respective level of the Simon-Tree as a leftmost child of node a.

All nodes we traversed from leftmost leaf up to node a represent l-blocks with $l \geq (k+2)$. These blocks are closed during the process of finding the correct position as seen in Algorithm 2. Since i is a $(k+2)$-splitting position we set the starting position for all open l-blocks, with $l \geq k+2$, to $i+1$.

It remains only to mention the special case, where we do not find an occurrence of $w[i]$ on our traversal from leftmost leaf to the root. In this case, the letter did not appear yet in the word. It therefore marks a 1-splitting position and as per Algorithm 2, we return the tree root, to which the block $[?:i]$ is then added as a leftmost child as per Algorithm 3.

Algorithm analysis. The pseudocode for our algorithm is shown in Algorithms 1–3. Theorem 7 states the main result of this section. While the correctness of the algorithm follows mainly from the explanations above, its linear running time requires an amortized analysis. We observe that for each position i in w we traverse t nodes (representing open blocks) while going up on the leftmost branch, then insert one leaf on the leftmost branch while closing the t traversed nodes and moving them all to the right of the inserted leaf (so out from the leftmost branch). As the total number of nodes in the Simon-Tree T_w is linear in $|w|$, and each node is inserted once and traversed once, the conclusion follows. For the interested reader we point out that our analysis resembles to a certain extent the one of the algorithm constructing the Carthesian-Tree for a set of numbers [33].

▶ **Theorem 7.** *Given a word w, with $|w| = n$, we can construct its Simon-Tree in $O(n)$ time.*

4 Connecting two Simon-Trees

In this section, we propose a linear-time algorithm for the MaxSimK problem. The general idea of this algorithm is to analyse simultaneously the Simon-Trees of the two input words s and t of length n and n', respectively, and establish a connection between their nodes.

4.1 The S-Connection

In our solution of MaxSimK, we construct a relation called S-Connection (abbreviation for Simon-Connection) between the nodes of the Simon-Trees T_s and T_t constructed from the two input words s and t.

▶ **Definition 8.** *The (explicit or implicit) k-node a of T_s and the (explicit or implicit) k-node b of T_t are S-connected (i.e., the pair (a, b) is in the S-Connection) if and only if $s[i : n] \sim_k t[j : n']$ for all positions i in block a and positions j in block b.*

If two k-nodes a and b are S-connected, we say that b is a's S-Connection (and vice versa). Additionally, if two nodes are S-connected, then the corresponding blocks are said to be S-connected too.

Adapted from the equivalence classes within a word, each explicit or implicit k-node of T_s can be S-connected to at most one k-node of T_t (since they are then representing blocks which on their part represent the same equivalence class of the set of suffixes of w w.r.t. $\sim_k$).

▶ Remark 9. The S-Connection is *non-crossing.* This means that if the k-block $a = [m_a : n_a]$ of T_s is S-connected to the k-block $b = [m_b : n_b]$ of T_t, the k'-block $c = [m_c : n_c]$ of T_s is S-connected to the k'-block $d = [m_d : n_d]$ of T_t, and $m_a < m_c$, then $m_b < m_d$. Similarly, if $n_a < n_c$ then $n_b < n_d$.

4.2 The P-Connection

For constructing the S-Connection efficiently, we define a coarser relation called P-Connection (abbreviation for potential-connection) that covers the S-Connection. The P-Connection defines, for each node of T_s, a unique node of T_t to which it may be S-connected. Later, we will attempt to determine and *split*, for each level k from 1 to maximally n, all pairs of (explicit and implicit) k-nodes which were P-connected but are not S-connected. In a sense, this splitting allows us to gradually refine the P-Connection until we get exactly the S-Connection. The P-Connection for the words s and t is defined as follows.

▶ **Definition 10.** *The 0-nodes of T_s and T_t are P-connected. For all levels k of T_s, if the explicit or implicit k-nodes a and b (from T_s and T_t, respectively) are P-connected, then the i^{th} child of a is P-connected to the i^{th} child of b, for all i. No other nodes are P-connected.*

If k-nodes a and b are P-connected, we say that b is a's P-Connection (and vice versa).

According to its definition, the P-Connection can be computed efficiently in a straightforward manner. This definition is essentially based on the following Lemma 11. However, because Lemma 11 is not both necessary and sufficient (unlike, e.g., Lemma 5), it can only be used to define a relation coarser than the S-Connection and cannot be used to characterise (and, consequently, compute in a simple way) the S-Connection itself. Recall that in Simon-Trees the children of a node are numbered right to left.

▶ **Lemma 11.** *Let $k \geq 1$. Let $a = [m_a : n_a]$ be a k-block in s and $b = [m_b : n_b]$ a k-block in t with $a \sim_k b$. Then the i^{th} child of the node a of T_s can only be S-connected (but it is not necessarily connected) to the i^{th} child of the node b of T_t, for all $i \geq 1$.*

It is not hard to see that, in the spirit of Remark 9, the P-Connection is non-crossing. Moreover, by Lemma 11, if the k-blocks a and b are S-connected, they are also P-connected. It is very important to note that a pair of nodes whose parent-nodes are not S-connected is also not S-connected. So, as our approach is to refine the P-Connection till the S-Connection is reached, we can immediately decide that a pair of nodes (a, b) is not in the S-Connection when the pair consisting of their respective parent-nodes is not in the S-Connection.

4.3 From P- to S-Connection

Preliminary transformation. As mentioned, our algorithm solving MaxSimK uses the Simon-Trees of s and t. To make the exposure simpler, we make the following simple transformation of the trees. If a is a k-node such that a is a singleton, we add as a child of this node a $(k+1)$-node representing the same block a (this was an implicit node before, now made explicit); the newly added node on level $k+1$ does not have any children (i.e., this procedure is not applied recursively). Before, by Lemma 5, all blocks $[i:i]$ of w appeared explicitly exactly once in T_w. Therefore, each singleton-block $[i:i]$ of s (respectively, t) appears now exactly twice in T_s (respectively, T_t).

In general, these now explicit nodes are used to guarantee the existence of a P-connected node (implicit or explicit) for every explicit singleton node on some level $k+1$ that was on a splitting position on level k, so we can determine singleton nodes that are $\sim_k$- but not $\sim_{k+1}$-congruent to the corresponding nodes in the other tree. The transformation has the following direct consequence that we will use: each singleton-block a appears now on two consecutive levels. While the node corresponding to a on the higher level may be S-connected to a node corresponding to a non-singleton-block, the node corresponding to a on the lower level may be S-connected only to a singleton-node.

As a second consequence, it is worth noting that explicit nodes might be connected to implicit nodes, too. However, this is only true for explicit nodes which were added during the transformation described above, i.e., singleton explicit nodes. Explicit nodes which are not singletons cannot be connected to implicit nodes.

Refining the P-Connection. The main step of our approach is, while considering the levels of the trees T_s and T_t in increasing order, to identify the pairs of P-connected nodes from the respective levels which are not S-connected and consequently *split* them. At the same time, we identify the pairs of singleton-blocks occurring explicitly on higher levels (and only implicitly on the current levels) which are not S-connected on this level, and also *split* them *on the current level.* For simplicity of exposure, when we split two k-blocks, we say that we k-split them. In order to implement this idea, we use the following Lemma 12 to define a splitting criterium.

We introduce first some notations. For $w \in \{s,t\}$, a position $j \leq |w|$, and a letter x, we define $\texttt{next}_w(j,x)$ as the leftmost position where x occurs in $w[j:|w|]$, or as ∞ if $x \notin \text{alph}(w[j:|w|])$. For a block $a = [m_a : n_a]$ of the word w and a letter x, we define $\texttt{next}_w(a,x) = \texttt{next}_w(n_a,x)$. We generally omit the subscript w when it is clear from the context. Furthermore, we define $\text{alph}(a)$ for a block $a = [m_a : n_a]$ as $\text{alph}(w[m_a : n_a])$.

▶ **Lemma 12.** *Let $k \geq 1$. Let $a = [m_a : n_a]$ be a k-block in the word s and $b = [m_b : n_b]$ a k-block in the word t with $a \sim_k b$. Let $a' = [m_{a'} : n_{a'}]$ be a $(k+1)$-block in a and $b' = [m_{b'} : n_{b'}]$ be a $(k+1)$-block in b. Then $a' \nsim_{k+1} b'$ if and only if there exists a letter x such that $s[\texttt{next}(a',x)+1 : n] \nsim_k t[\texttt{next}(b',x)+1 : n']$.*

The main idea of this lemma (illustrated in Figure 2) is that two $(k+1)$-blocks a' and b' are not S-connected, although their parents were S-connected, if and only if we can find a letter x such that $s[\texttt{next}(a',x)+1 : n]$ and $t[\texttt{next}(b',x)+1 : n']$ are not k-equivalent but $(k-1)$-equivalent. That is, $\texttt{next}(a',x)+1$ and $\texttt{next}(b',x)+1$ should occur, respectively, in two k-blocks which were split, but whose parents were S-connected. A word distinguishing the suffixes starting in a' from those starting in b' has the first letter x, and is continued by the word of length k which distinguishes $s[\texttt{next}(a',x)+1 : n]$ and $t[\texttt{next}(b',x)+1 : n']$.

Identifying P-connected pairs to be split. When going through the trees level by level, the 1-blocks (all occuring explicitly on level 1 of T_s and respectively T_t) which are S-connected can be easily and efficiently identified: the i^{th} node $a = [m_a : n_a]$ on level 1 of T_s is connected to the i^{th} node $b = [m_b : n_b]$ of T_t if and only if $\text{alph}(s[n_a : n]) = \text{alph}(t[n_b : n'])$. All the other P-connected pairs of 1-blocks are not S-connected, so they are 1-split.

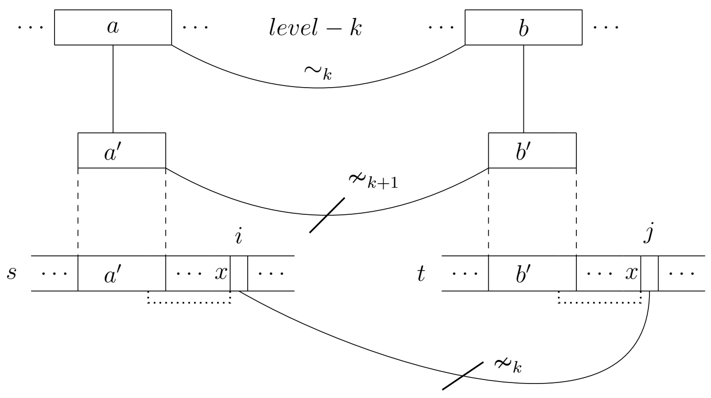

Figure 2 Illustration of Lemma 12.

The identification of the pairs of $(k+1)$-blocks and pairs of singletons which need to be $(k+1)$-split is based on Lemma 12. The idea is the following. A pair of P-connected $(k+1)$-blocks $a' = [m_{a'} : n_{a'}]$ of T_s and $b' = [m_{b'} : n_{b'}]$ of T_t is not S-connected if and only if there exists a letter x such that $s[\texttt{next}(a', x) + 1 : n] \nsim_k t[\texttt{next}(b', x) + 1 : n']$. So, in order to be able to $(k+1)$-split two nodes (whose parents are S-connected), we need to identify two positions i and j (and a corresponding letter x), with $i = \texttt{next}(a', x) + 1$ and $j = \texttt{next}(b', x) + 1$ which were k-split but not $(k-1)$-split. We search for position i inside the k-blocks of T_s, and try to see where position j may occur in the blocks of T_t such that these two positions are not in S-connected k-blocks. To find the position i (and the corresponding j) we analyse two cases.

The first case (A) is when i occurs inside an implicit k-node, which is the singleton-k-block $[i : i]$. On the lowest level where this block appeared as an explicit node, it was S-connected to a node $[j : j]$ representing a singleton too, according to Section 4.1 and Lemma 5. Thus, position i can only be k-split from the position j of t to which it was S-connected (it was already disconnected from all other positions on level ℓ). If i and j are both directly preceded by the same symbol (say x), then the pair (i, j) gives us exactly the positions we were searching for.

The second case (B) is when i occurs inside an explicit k-node in T_s. Let $A = [m_A : n_A]$ and $B = [m_B : n_B]$ be two $(k-1)$ blocks from T_s and T_t, respectively, such that $A \sim_{k-1} B$, and $a = [m_a : n_a]$ and $b = [m_b : n_b]$ be the ℓ^{th} child of A and B, respectively. Clearly, b might be explicit, implicit, or even empty. If b is non-empty, the following holds. All positions of a are k-split from the positions $[m_B : m_b - 1]$ and from the positions $[n_b + 1 : n_B]$, because a is not P-connected to the blocks covering those positions. Also, if a and b are not S-connected then all positions of a are also k-split from the positions of b.

An example. Let $s = acab$ and $t = acabba$ be two words. Their respective Simon-Trees T_s and T_t are depicted in Figure 3 along with their P- and S-Connection. Note that for the sake of not crossing edges and simplifying the presentation, the Simon-Trees in Figure 3 are rotated by 90 degrees to the right and to the left, respectively. Thus, the roots of the trees are on the outer left and right side of the figure. Additionally, the tree T_t on the right is mirrored, so that nodes from a P-connected pair are vis-à-vis. Also, the trees already contain the singleton nodes that were originally implicit but are now made explicit by our aforementioned

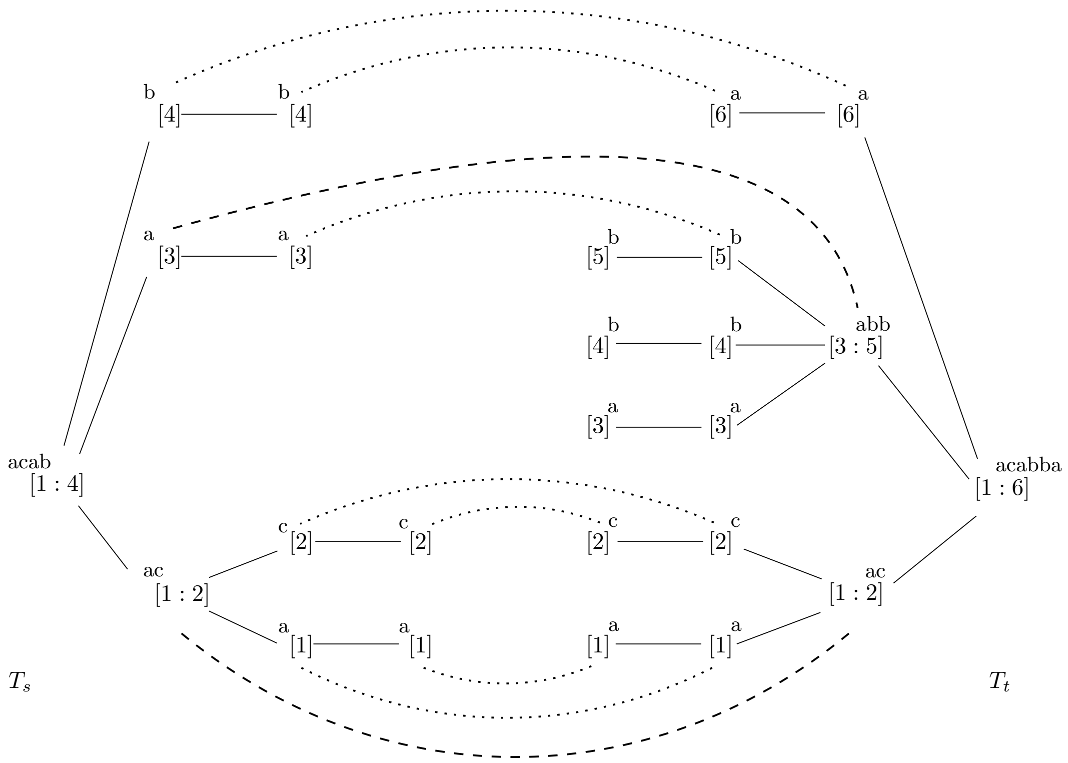

Figure 3 Two transformed Simon-Trees with their P- and S-Connection.

transformation. From the figures it becomes also clear that this transformation is needed in the case of the singleton-node $[5 : 5]$ from the 2nd level of T_t which is P-connected to the singleton-node $[3 : 3]$ from the 2nd level of T_s.

In the beginning we are considering all possibly connected blocks by determining all P-connected pairs. While the dashed and dotted lines connect, respectively, the nodes of all the pairs from the P-Connection, the S-Connection is obtained by splitting step by step P-connected pairs that cannot be equivalent with regard to their respective level. The dotted edges symbolize exactly these split pairs, and in the end, the S-Connection consists only of the pairs connected with a dashed edge.

Following Theorems 16 and 17 stated at the very end of this paper, we get the largest k for which the two words s and t are k-equivalent by finding the largest k for which the k-blocks containing position 1 of both words are S-connected. In our example, the blocks $s[1 : 4]$ and $t[1 : 6]$ representing the complete words are naturally 0-equivalent. Furthermore, as seen in Figure 3, the blocks $s[1 : 2]$ and $t[1 : 2]$ on level 1 are S-connected, but the blocks $s[1 : 1]$ and $t[1 : 1]$ are not S-connected on level 2. Thus, the largest k for which $s \sim_k t$ holds is 1.

Path to efficiency. Taking (i, j) to be each position of block a paired with each of the positions from which it is k-split, according to the above, might not be efficient. However, the combinatorial Lemma 12 allows us to switch slightly the point of view and ultimately obtain an efficient method. We will traverse the kth level of a Simon-Tree T_s from right to left and when considering a node a on this level, our approach is to determine which nodes should be $k + 1$-split due to any pair (i, j), where i is some position occurring in the block a.

The mechanism allowing us to do this is stated in Lemma 13, and this essentially explains how to determine all the $k+1$-splits determined by positions of a (and their corresponding pairing from b). We show how to proceed in both cases (A) and (B). Clearly, a symmetrical approach would also work (so looking at nodes in T_t and positions in t).

Firstly, we need a few more notations. For a block $a = [m_a : n_a]$ of s or t and a letter x, let $\texttt{prev}(a, x)$ be the rightmost occurrence of x in $s[1 : m_a - 2]$ (or 0 if $x \notin \text{alph}(s[1 : m_a - 2])$), and let $\texttt{right}(a, x)$ be the rightmost occurrence of x in $s[1 : n_a - 1]$ (or 0 if $x \notin \text{alph}(s[1 : n_a - 1])$).

The setting in which Lemma 13 is stated is the following. We have two P-connected k-blocks $a = [m_a : n_a]$ and $b = [m_b : n_b]$ from T_s and T_t, respectively, whose parent-nodes (explicit or implicit) are S-connected. The lemma defines a necessary and sufficient condition for a pair of (explicit or implicit) $(k+1)$-nodes (a', b') to be $(k+1)$-split because there exists a letter x and a pair of positions (i, j), with $i = \texttt{next}(a', x)+1$, $i \in [m_a : n_a]$, $j = \texttt{next}(b', x)+1$, and $s[i : n] \not\sim_k t[j : n']$. Such a pair (a', b') is called $(a, k+1)$-split (that is, a causes the respective split on level $k+1$). Note that a' and b' are the (explicit or implicit) children of either a and b or of two k-blocks which are left of a and b. In any way, their parents are S-connected; otherwise a' and b' would have already been split on a higher level.

This setting is also illustrated in Figure 4.

▶ **Lemma 13.** *For $k \geq 1$, let $a = [m_a : n_a]$ be a k-block in s and $b = [m_b : n_b]$ its P-Connection (which is a k-block in t). Then a pair of P-connected $(k+1)$-blocks $a' = [m_{a'} : n_{a'}]$ in s and $b' = [m_{b'} : n_{b'}]$ in t is $(a, k+1)$-split if and only if there exists a letter x in alph$(s[m_a - 1 : n_a - 1])$ such that at least one of the following holds:*

1. *a' ends strictly between $\texttt{prev}(a, x)$ and m_a (i.e., $\texttt{prev}(a, x) < n_{a'} < m_a$), and b' ends to the left of $\texttt{prev}(b, x) + 1$ (i.e., $n_{b'} \leq \texttt{prev}(b, x)$).*
2. *a' ends between $\texttt{prev}(a, x)$ and $\texttt{right}(a, x)$ (i.e., $\texttt{prev}(a, x) < n_{a'} \leq \texttt{right}(a, x)$), and b' ends between $\texttt{right}(b, x)$ and n_b (i.e., $\texttt{right}(b, x) < n_{b'} \leq n_b$).*
3. *$a \not\sim_k b$, a' ends between $\texttt{prev}(a, x)$ and $\texttt{right}(a, x)$ (i.e., $\texttt{prev}(a, x) < n_{a'} \leq \texttt{right}(a, x)$), and b' ends between $\texttt{prev}(b, x)$ and $\texttt{right}(b, x)$ (i.e., $\texttt{prev}(b, x) < n_{b'} \leq \texttt{right}(b, x)$).*

In Lemma 13, because $k \geq 1$ and the blocks a' and b' are the (explicit or implicit) children of two S-connected k-blocks, it follows that $\text{alph}(s[n_{a'} : n]) = \text{alph}(t[n_{b'} : n'])$. This means, in particular, that $\texttt{next}(a', x) \neq \infty$ for some letter x if and only if $\texttt{next}(b', x) \neq \infty$.

Now, we can explain how to algorithmically apply Lemma 13 and find the pairs of $(k+1)$-blocks which should be split. For this, we can define, and compute in the step where the k-split pairs were obtained, a list L_k of pairs of singleton-k-blocks which were k-split and a list H_k of all the explicit k-nodes of T_s and their P-connections.

We first consider each explicit k-node a of T_s and its P-Connection, the node b of T_t (in both cases: when a and b were k-split or when they were not). For $x \in \text{alph}(s[m_a - 1 : n_a - 1]) \cup \{s[m_a - 1]\}$ (note that the symbols $x \in \text{alph}(s[m_a - 1 : n_a - 1])$ can be identified as the first symbols of the $(k+1)$-blocks into which $a = s[m_a : n_a]$ is split, except the rightmost one; these are the children of node a except the rightmost one) we do the following:

1. identify each $(k+1)$-block $a' = [m_{a'} : n_{a'}]$ with $\texttt{prev}(a, x) < n_{a'} < m_a$ and its pair $b' = [m_{b'} : n_{b'}]$. Then (a', b') is not in the S-Connection if $n_{b'} \leq \texttt{prev}(b, x)$ (i.e., a' and b' are $(a, k+1)$-split).
2. identify each $(k+1)$-block $a' = [m_{a'} : n_{a'}]$ with $\texttt{prev}(a, x) < n_{a'} \leq \texttt{right}(a, x)$ and its pair $b' = [m_{b'} : n_{b'}]$. Then (a', b') is not in the S-Connection if $\texttt{right}(b, x) < n_{b'} \leq n_b$.
3. if $a \not\sim_k b$, identify each $(k+1)$-block a' with $\texttt{prev}(a, x) < n_{a'} \leq \texttt{right}(a, x)$ and its pair $b' = [m_{b'} : n_{b'}]$. Then (a', b') is not in the S-Connection if $\texttt{prev}(b, x) < n_{b'} \leq \texttt{right}(b, x)$.

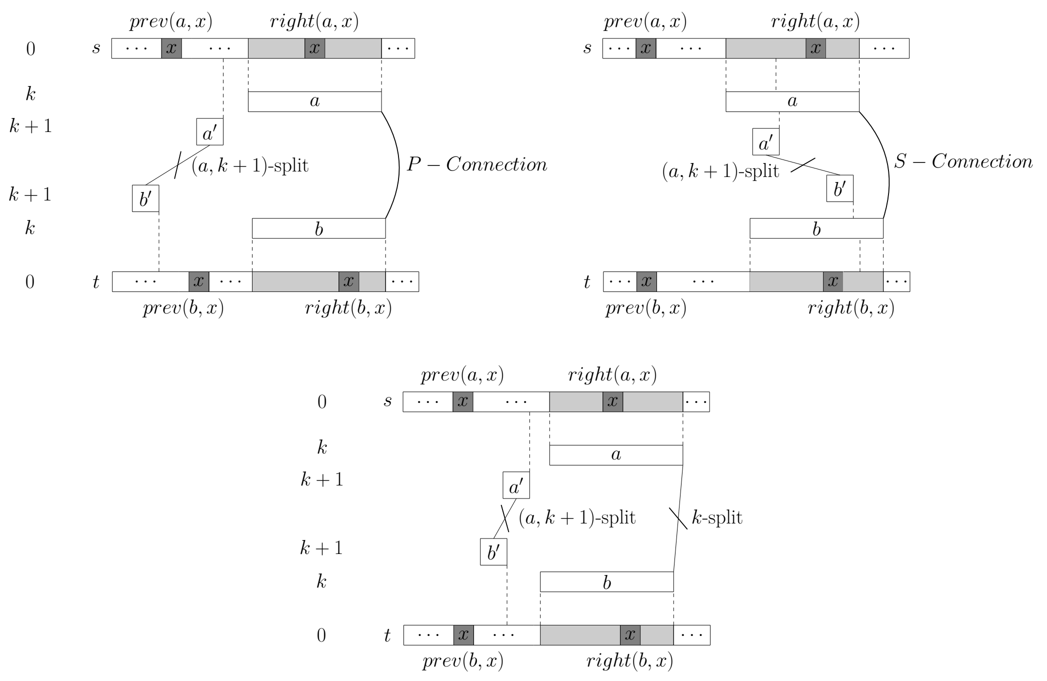

Figure 4 Illustration of the three cases of Lemma 13.

For every pair (a, b) of singleton-k-blocks which were k-split (from the list L_k), we only perform step 3 from above.

For each k-block a we considered (explicit or implicit node of T_s), we collect the singleton-$(k+1)$-blocks that were $(a, k+1)$-split, to be used when computing the $(k+2)$-splits.

The next step is to implement this idea, i.e., to describe data structures allowing us to identify efficiently the $(k+1)$-blocks a' and b' from above. We say that a pair of blocks/nodes (a', b') meets an interval-pair $([p : q], [p' : q'])$ if a' ends in $[p : q]$, and b' ends in $[p' : q']$.

Our approach is the following. We process the blocks on level k and, for each of them, get (at most) three lists of interval-pairs (one component is an interval of positions in s, the other an interval in t). On level $k+1$, we split each pair of P-connected blocks (a', b') which meets one interval-pair from our list. A crucial property here is that, for each interval-pair, the $(k+1)$-blocks of s which meet it, and are accordingly split from their P-Connections, are consecutive (explicit and implicit) $(k+1)$-nodes in T_s. Thus, in order to make use of Lemma 13, we draw on the technical results given by Lemmas 14 and 15.

▶ **Lemma 14.** *Let $a = [m_a : n_a]$ and $b = [m_b : n_b]$ be P-connected blocks of T_s and T_t, respectively, and $s_a = s[m_a - 1 : n_a - 1]$. We can compute in overall $O(|alph(a)|)$ time the three lists, associated to the pair (a, b), containing:*

1. *the interval-pairs $([\texttt{prev}(a,x) + 1 : m_a - 1], [0 : \texttt{prev}(b,x)])$, for all $x \in alph(s_a)$;*
2. *the interval-pairs $([\texttt{prev}(a,x)+1 : \texttt{right}(a,x)], [\texttt{right}(b,x)+1 : n_b])$, for all $x \in alph(s_a)$;*
3. *the interval-pairs $([\texttt{prev}(a,x) + 1 : \texttt{right}(a,x)], [\texttt{prev}(b,x) + 1 : \texttt{right}(b,x)])$, for all $x \in alph(s_a)$.*

▶ **Lemma 15.** *Given two words s and t, with $|s| = n$ and $|t| = n'$, $n \geq n'$, and their Simon-Trees T_s and T_t, we can check in $O(n)$ overall time for all pairs of P-connected 1-blocks (a, b), with $a = [m_a : n_a]$ and $b = [m_b : n_b]$, whether $alph(s[n_a : n]) = alph(t[n_b : n'])$.*

Efficiently constructing the S-Connection and solving MaxSimK. Based on the previous lemmas, we can now show our main technical theorem. We use Lemma 15 to see which 1-nodes are not S-connected. This is done in $O(n)$ time. Then consider the k-nodes, for each $k \geq 2$ in increasing order. For each pair (a, b) of $(k-1)$-nodes which were split (i.e., removed from the S-Connection) in the previous step, we split the pairs of k-nodes meeting one of the interval-pairs of the three lists of (a, b), as computed in Lemma 14. To do this efficiently, we maintain an interval union-find and an interval split-find structure for each word. While the concrete algorithm can be found in the full version of this paper [24], we can now state our main result in Theorem 16.

▶ **Theorem 16.** *Given two words s and t, with $|s| = n$ and $|t| = n'$, $n \geq n'$, we can compute in $O(n)$ time the following:*

- *the S-Connection between the nodes of the two trees T_s and T_t;*
- *for each $i \in [1 : n]$, the highest level k on which the (implicit or explicit) node $[i : i]$ is k-split from its P-Connection.*

Finally, in order to solve MaxSimK, we need to compute the largest k for which the k-block $a = [1 : n_a]$ of s is S-connected to the k-block $b = [1 : n_b]$ of t. Thus, we execute the algorithm of Section 4.3 and the aforementioned level k can be easily found by checking, level by level, the blocks that contain position 1 of s on each level of T_s and the block to which they are S-connected in T_t. As a consequence of Theorem 16, we can now show our main result.

▶ **Theorem 17.** *Given two words s and t, with $|s| = n$ and $|t| = n'$, $n \geq n'$, we can solve MaxSimK and compute a distinguishing word of minimum length for s and t in $O(n)$ time.*

5 Conclusions and future work

In this paper, we presented the first algorithm solving MaxSimK in optimal time. This algorithm is based on the definition and efficient construction of a novel data-structure: the Simon-Tree associated to a word. Our algorithm constructs the respective Simon-Trees for the two input words of MaxSimK, and then establishes a connection between their nodes. While the Simon-Tree is a representation of the classes induced, for all $k \geq 1$, by the $\sim_k$-congruences on the set of suffixes of a word, this connection allows us to put together the classes induced by the respective congruences on the set of suffixes of both input word, and to obtain, as a byproduct, the answer to MaxSimK.

The work presented in this paper can be continued naturally in several directions. For instance, it seems interesting to us to compute efficiently, for two words s and t, what is the largest k such that $\text{Subseq}_{\leq k}(s) \subseteq \text{Subseq}(t)$. Similarly, one could consider the following pattern-matching problem: given two words s and t, and a number k, compute efficiently all factors $t[i : j]$ of t such that $t[i : j] \sim_k s$. Finally, SimK could be extended to the following setting: given a word s and regular (or a context-free) language L, and a number k, decide efficiently whether there exists a word $t \in L$ such that $s \sim_k t$. A variant of MaxSimK can be also considered in this setting: given a word s and regular (or a context-free) language L, find the maximal k for which there exists a word $t \in L$ such that $s \sim_k t$.

References

1 Ricardo A. Baeza-Yates. Searching subsequences. *Theor. Comput. Sci.*, 78(2):363–376, 1991.

2 Laura Barker, Pamela Fleischmann, Katharina Harwardt, Florin Manea, and Dirk Nowotka. Scattered factor-universality of words. In *Proc. DLT 2020*, volume 12086 of *Lecture Notes in Computer Science*, pages 14–28. Springer, 2020.

3 Karl Bringmann and Bhaskar Ray Chaudhury. Sketching, streaming, and fine-grained complexity of (weighted) LCS. In *Proc. FSTTCS 2018*, volume 122 of *LIPIcs*, pages 40:1–40:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

4 Karl Bringmann and Marvin Künnemann. Multivariate fine-grained complexity of longest common subsequence. In *Proc. SODA 2018*, pages 1216–1235. SIAM, 2018.

5 Maxime Crochemore, Borivoj Melichar, and Zdenek Tronícek. Directed acyclic subsequence graph - overview. *J. Discrete Algorithms*, 1(3-4):255–280, 2003.

6 Joel D. Day, Pamela Fleischmann, Florin Manea, and Dirk Nowotka. k-spectra of weakly-c-balanced words. In *Proc. DLT 2019*, volume 11647 of *Lecture Notes in Computer Science*, pages 265–277. Springer, 2019.

7 Cees H. Elzinga, Sven Rahmann, and Hui Wang. Algorithms for subsequence combinatorics. *Theor. Comput. Sci.*, 409(3):394–404, 2008.

8 Lukas Fleischer and Manfred Kufleitner. Testing Simon's congruence. In *Proc. MFCS 2018*, volume 117 of *LIPIcs*, pages 62:1–62:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

9 Dominik D. Freydenberger, Pawel Gawrychowski, Juhani Karhumäki, Florin Manea, and Wojciech Rytter. Testing k-binomial equivalence. In Multidisciplinary Creativity*, a collection of papers dedicated to G. Păun 65th birthday*, pages 239–248, 2015. available in CoRR: `arXiv:1509.00622`.

10 Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *J. Comput. Syst. Sci.*, 30(2):209–221, 1985.

11 Emmanuelle Garel. Minimal separators of two words. In *Proc. CPM 1993*, volume 684 of *Lecture Notes in Computer Science*, pages 35–53. Springer, 1993.

12 Simon Halfon, Philippe Schnoebelen, and Georg Zetzsche. Decidability, complexity, and expressiveness of first-order logic over the subword ordering. In *Proc. LICS 2017*, pages 1–12, 2017.

13 Jean-Jacques Hebrard. An algorithm for distinguishing efficiently bit-strings by their subsequences. *Theoretical Computer Science*, 82(1):35–49, 22 May 1991.

14 Hiroshi Imai and Takao Asano. Dynamic segment intersection search with applications. In *Proc. 25th FOCS, 1984*, pages 393–402, 1984.

15 Prateek Karandikar, Manfred Kufleitner, and Philippe Schnoebelen. On the index of Simon's congruence for piecewise testability. *Inf. Process. Lett.*, 115(4):515–519, 2015.

16 Prateek Karandikar and Philippe Schnoebelen. The height of piecewise-testable languages with applications in logical complexity. In *Proc. CSL 2016*, volume 62 of *LIPIcs*, pages 37:1–37:22, 2016.

17 Prateek Karandikar and Philippe Schnoebelen. The height of piecewise-testable languages and the complexity of the logic of subwords. *Logical Methods in Computer Science*, 15(2), 2019.

18 Dietrich Kuske and Georg Zetzsche. Languages ordered by the subword order. In *Proc. FOSSACS 2019*, volume 11425 of *Lecture Notes in Computer Science*, pages 348–364. Springer, 2019.

19 Marie Lejeune, Julien Leroy, and Michel Rigo. Computing the k-binomial complexity of the Thue-Morse word. In *Proc. DLT 2019*, volume 11647 of *Lecture Notes in Computer Science*, pages 278–291, 2019.

20 Julien Leroy, Michel Rigo, and Manon Stipulanti. Generalized Pascal triangle for binomial coefficients of words. *Electron. J. Combin.*, 24(1.44):36 pp., 2017.

21 M. Lothaire. *Combinatorics on Words.* Cambridge University Press, 1997.

22 David Maier. The complexity of some problems on subsequences and supersequences. *J. ACM*, 25(2):322–336, April 1978.

23 Alexandru Mateescu, Arto Salomaa, and Sheng Yu. Subword histories and Parikh matrices. *J. of Comput. Syst. Sci.*, 68(1):1–21, 2004.

24 Paweł Gawrychowski, Maria Kosche, Tore Koß, Florin Manea, and Stefan Siemer. Efficiently testing Simon's congruence. *preprint, CoRR*, 2020. `arXiv:2005.01112`.

25 Michel Rigo and Pavel Salimov. Another generalization of abelian equivalence: Binomial complexity of infinite words. *Theor. Comput. Sci.*, 601:47–57, 2015.

26 Arto Salomaa. Connections between subwords and certain matrix mappings. *Theor. Comput. Sci.*, 340(2):188–203, 2005.

27 David Sankoff and Joseph Kruskal. *Time Warps, String Edits, and Macromolecules The Theory and Practice of Sequence Comparison.* Cambridge University Press, 2000 (reprinted). originally published in 1983.

28 Shinnosuke Seki. Absoluteness of subword inequality is undecidable. *Theor. Comput. Sci.*, 418:116–120, 2012.

29 Imre Simon. An algorithm to distinguish words efficiently by their subwords.

30 Imre Simon. Piecewise testable events. In *Proc. Autom. Theor. Form. Lang., 2nd GI Conf.*, volume 33 of *LNCS*, pages 214–222. Springer, 1975.

31 Imre Simon. Words distinguished by their subwords (extended abstract). In *Proc. WORDS 2003*, volume 27 of *TUCS General Publication*, pages 6–13, 2003.

32 Zdenek Tronícek. Common subsequence automaton. In *Proc. CIAA 2002 (Revised Papers)*, volume 2608 of *Lecture Notes in Computer Science*, pages 270–275, 2002.

33 Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980.

34 Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, January 1974.

35 Georg Zetzsche. The complexity of downward closure comparisons. In *Proc. ICALP 2016*, volume 55 of *LIPIcs*, pages 123:1–123:14, 2016.