Efficiently Testing Simon’s Congruence

Paweł Gawrychowski¹, Maria Kosche², Tore Koß², Florin Manea², and Stefan Siemer²

¹University of Wrocław, Faculty of Mathematics and Computer Science, gawry@cs.uni.wroc.pl
²Göttingen University, Computer Science Department, Germany, maria.kosche@cs.uni-goettingen.de, tore.koss@uni-goettingen.de, florin.manea@informatik.uni-goettingen.de, stefan.siemer@cs.uni-goettingen.de

Abstract

Simon’s congruence $\sim_k$ is defined as follows: two words are $\sim_k$-equivalent if they have the same set of subsequences of length at most $k$. We propose an algorithm which computes, given two words $s$ and $t$, the largest $k$ for which $s \sim_k t$. Our algorithm runs in linear time $O(|s| + |t|)$ when the input words are over the integer alphabet \{1, \ldots, |s| + |t|\} (or other alphabets which can be sorted in linear time). This approach leads to an optimal algorithm in the case of general alphabets as well. Our results are based on a novel combinatorial approach and a series of efficient data structures.

Keywords: Simon’s congruence, Subsequence, Scattered factor, Efficient algorithms.

1 Introduction

A subsequence (also called scattered factor or subword, especially in automata and language theory) of a word $w$ is a word $u$ such that there exist (possibly empty) words $v_0, \ldots, v_n, u_1, \ldots, u_n$ with $u = u_1 \ldots u_n$ and $w = v_0 u_1 v_1 u_2 \ldots u_n v_n$. 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. 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: combinatorics on words, automata theory, formal verification, or string algorithms. For instance, subword histories and Parikh matrices (see, e.g., [26,28,29]) 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., [9,22,25,27], and the references therein. Further, in [13,21,31] 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 subequations: the longest common subsequence and the shortest common supersequence problems \cite{23,25}, or the string-to-string correction problem \cite{34}. Several algorithmic problems connected to subequation-combinatorics are approached and (partially) solved in \cite{7}.

One particularly interesting notion related to subsequences was introduced by Simon in \cite{31}. He defined the relation \( \sim_k \) (called now Simon’s congruence) as follows. Two words are \( \sim_k \)-equivalent if they have the same set of subsequences of length at most \( k \). In \cite{31}, as well as in \cite{24} Chapter 6, many fundamental properties (mainly of combinatorial nature) of \( \sim_k \) are discussed; this line of research was continued in, e.g., \cite{15,16,18} 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 \( \Sigma \), 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 \( \Sigma \), 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: for an input of size \( n \), the memory consists of words consisting of \( \Theta(\log n) \) bits. Basic operations (including arithmetic and bitwise boolean operations) on words take constant time, and any 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 the generality, we can assume the alphabet \( \Sigma \) to be simply \( \{1, \ldots, n+n'\} \) by radix-sorting the letters occurring in the input and then replacing them by their ranks in the sorted list after removing the duplicates. Generally speaking, we can always reduce the alphabet to \( \{1, \ldots, n+n'\} \) by sorting, so as long as we are able to sort the input letters (i.e., \( n+n' \) integers) in linear time we can indeed make such an assumption. In more restricted models of computations, such as the comparison model, in which the only allowed operation on the letters is comparison, sorting \( O(n) \) letters requires \( \Theta(n \log n) \) time. While there are some problems on strings for which such more restricted models seem reasonable (see, e.g., \cite{4} for a more detailed discussion), this is not the case here. Indeed, it is known that in the comparison model testing whether two sets \( S \) and \( T \) of size \( \leq n \) are equal requires \( \Omega(n \log n) \) time \cite{6}. Representing each set as a sequence in which its elements occur in an arbitrary order, we can reduce this problem to that of testing whether for two words \( s \) and \( t \) we have \( s \sim_1 t \). So, already for \( k = 1 \), testing if \( s \sim_k t \) requires \( \Omega(n \log n) \) time in the comparison model. Accordingly, both SimK and MaxSimK require \( \Omega(n \log n) \) time in this model. To avoid this kind of trivial bottlenecks, and also to be able to compare our results to the existing literature, we assume, throughout the paper, the standard Word RAM model.

Both problems SimK and MaxSimK were considered thoroughly in the literature. In particular, Hebrard \cite{14} presents MaxSimK as computing a similarity measure between strings, and mentions a solution of Simon \cite{30} for MaxSimK which runs in \( O(|\Sigma|nn') \) (the same solution is mentioned in \cite{11}). He goes on and improves this (see \cite{14}) in the case when \( \Sigma \) 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 \cite{14,30} as the methods used in the latter paper for binary strings did not seem to scale up. In \cite{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. Finally, in an extended abstract from 2003 [32], Simon presents another algorithm based on finite automata solving \( \text{MAXSimK} \) 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 \( \text{SimK} \) 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 \( \Sigma \) was computed in \( O(|\Sigma|n) \) time, so \( \text{SimK} \) was also solved in \( O(|\Sigma|n) \). A more efficient implementation of the ideas introduced in [8] was presented in [1]: the shortlex form of a word of length \( n \) over \( \Sigma \) can be computed in linear time, so \( \text{SimK} \) can be solved in optimal linear time. By binary searching for the smallest \( k \), this gives an \( O(n \log n) \) time solution for \( \text{MAXSimK} \). This brings up the challenge of designing an optimal linear-time algorithm for non-binary alphabets.

In this paper we confirm Simon’s claim from 2003 [32]. We present a complete algorithm solving \( \text{MAXSimK} \) in linear time on Word RAM with words of size \( \Theta(\log n) \). This closes the problem of finding an optimal algorithm for \( \text{MAXSimK} \). Our approach is not based on finite automata (as the one suggested by Simon), nor on the ideas from [1, 8]. Instead, it works as follows. First, 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 \textit{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 \( \text{MAXSimK} \), 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 \( \ell \), we connect two nodes on level \( \ell \) of the two trees if the suffixes starting in those blocks, in their respective words, have exactly the same subsequences of length at most \( \ell \). It follows that the value \( k \) required in \( \text{MAXSimK} \) 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 \( \text{MAXSimK} \) 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.

2 Preliminaries

Let \( \mathbb{N} \) be the set of natural numbers, including 0. An alphabet \( \Sigma \) is a nonempty finite set of symbols called \textit{letters}. A \textit{word} is a finite sequence of letters from \( \Sigma \), thus an element of the free monoid \( \Sigma^* \). Let \( \Sigma^+ = \Sigma^* \setminus \{ \varepsilon \} \), where \( \varepsilon \) is the empty word. The \textit{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 \(w[m:n] = w[m]w[m+1]\ldots w[n]\). For \(i \in \mathbb{N}\), we say \(i \in w\) if \(1 \leq i \leq |w|\). Accordingly, we say \(i \in w[m:n]\) if \(m \leq i \leq n\).

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 \(u \in \Sigma^*\), let \(|u|_x = \{|i \in [1:\ell(w)] | w[i] = x\}\) and \(\text{alph}(w) = \{x \in \Sigma \mid |w|_x > 0\}\) for \(w \in \Sigma^*\); in other words, \(\text{alph}(w)\) denotes the smallest subset \(S \subseteq \Sigma\) such that \(w \in S^*\).

**Definition 1.** We call \(w'\) 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 \(w' = w[i_1]w[i_2]\ldots w[i_k]\). Let \(SF_{\leq k}(i,w)\) 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 \(SF_{\leq k}(1,w)\).

Alternatively, \(w' = w'_1\ldots w'_\ell\) is a subsequence of \(w\) if there exist \(x_1, \ldots, x_{\ell+1} \in \Sigma^*\) such that \(w = x_1w'_1\ldots x_\ell w'_\ell x_{\ell+1}\). For \(k \in \mathbb{N}\), \(SF_{\leq k}(1,w)\) 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\ ad \ 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., \(SF_{\leq k}(1,w) = SF_{\leq k}(1,w')\).

(ii) Let \(i, j \in w\). 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 \(SF_{\leq k}(1,w)\) and \(SF_{\leq k}(1,w')\).

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

We recall the following definitions of two important data structures.

**Definition 3** (Interval split-find and interval union-find data structures). Let \(V = [1:n]\) and \(S\) a set with \(S \subseteq V\). The elements of \(S = \{s_1, \ldots, s_p\}\) are called borders and are ordered \(0 = s_0 < s_1 < \ldots < s_p < s_{p+1} = n + 1\) where \(s_0\) and \(s_{p+1}\) are generic borders. For each border \(s_i\) we define \(V(s_i) = [s_{i-1}+1 : s_i]\) as an induced interval. Now \(P(S) := \{V(s_i) \mid s_i \in S\}\) gives an ordered partition of the set \(U\).

- The interval split-find structure maintains the partition \(P(S)\) under the operations:
  - For \(u \in V\), \(\text{find}(u)\) returns \(s_i \in S \cup \{n+1\}\) such that \(u \in V(s_i)\). In other words all elements in the interval \(V(s_i)\) have the representative \(s_i\).
  - For \(u \in V \setminus S\), \(\text{split}(u)\) updates the partition \(P(S)\) to \(P(S \cup \{u\})\). That is, we find the interval \(V(s_i)\) containing \(u\), split it into the interval containing elements \(\leq u\) and the interval of elements \(> u\), and update the partition so that further \(\text{find}\) and \(\text{split}\) operations can be performed.

- The interval union-find structure maintains the partition \(P(S)\) under the operations:
  - For \(u \in V\), \(\text{find}(u)\) returns \(s_i \in S \cup \{n+1\}\) such that \(u \in V(s_i)\).
  - For \(u \in S\), \(\text{union}(u)\) updates the partition \(P(S)\) to \(P(S \setminus \{u\})\). That is, if \(u = s_1\), then we replace the intervals \(V(s_i)\) of \(V(s_i)\) by the single interval \([s_{i-1}+1 : s_{i+1}]\), and update the partition so that further \(\text{find}\) and \(\text{union}\) operations can be performed.

When using the data structures of Definition 3, we employ a less technical language: we describe the intervals stored initially in the structure, and then, for the union-find structure, the
unions are made between adjacent intervals, while for the split-find structure we split intervals according to a value. Note that the representative of each interval is its maximum; we can easily enhance the data structures so that the \texttt{find} operation returns both borders of the interval containing the searched value. The following lemma was shown in \cite{10,15}.

\textbf{Lemma 1.} 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 \( O(n) \) time and space.

3 Constructing the Simon-tree of a word

In this section, we develop a method to efficiently partition the positions of a given word \( w \) 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 we only deal with one input word \( w \), we will sometimes omit the reference to this word in our notation: e.g. \( SF_k(i) = SF_k(i, w) \); 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 \( i < j \leq n \), then \( w[j:n] \) is a suffix of \( w[i:n] \), hence \( SF_k(i) \supset SF_k(j) \) holds for all \( k \in \mathbb{N} \). For any \( l \in [i:j] \) we obtain \( SF_k(i) \supset SF_k(l) \supset SF_k(j) \).

If we additionally let \( i \sim j \), then the sets of subsequences corresponding to \( i \) and \( j \) respectively are equal, so \( SF_k(i) = SF_k(l) = SF_k(j) \) and \( i \sim k \land j \sim k \). Hence, the equivalence classes of the set of suffixes of \( w \) w.r.t. \( \sim_k \) correspond to contiguous sets of indices in \( [1:|w|] \), namely the starting positions of the suffixes in each class. We call these classes \( k \)-blocks. For a \( k \)-block \( b = [m_b:n_b] \), \( m_b \) is its starting position and \( n_b \) its ending position.

It is worth noting that if \( i \sim_{k+1} j \), then \( i \sim_k j \), which means that \( \sim_{k+1} \) 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 contiguous blocks. In particular, if a \( k \)-block consists of a single position (i.e., it is a \textit{singleton-}\( k \)-block), then this position remains an \( \ell \)-block for all \( \ell > k \).

\textbf{Remark 1.} If \( i \sim_k i+1 \) and \( i \sim_{k+1} i+1 \), then we will say that \( i \) \textbf{splits} its \( k \)-block or that \( i \) is a \((k+1)\)-\textbf{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 \( a \)).

\textbf{Remark 2.} For any \( i \in w \) we have \( SF_0(i) = \{ \varepsilon \} \). Thus, we refer to \( w \) as the \textit{0-block} of \( w \).

We begin by analysing the simplest congruence, namely \( \sim_1 \): the set \( SF_1(1) \) naturally corresponds to the letters in \( \text{alph}(w) \) and, analogously, \( SF_1(i) \) corresponds to the letters in \( \text{alph}(w[i:n]) \). Using Definition \ref{def:splitting} we can completely characterize the equivalence classes induced by \( \sim_1 \) on the set of suffixes of \( w \):

\textbf{Lemma 2.} \( i \sim_1 j \) if and only if \( \text{alph}(w[i:n]) = \text{alph}(w[j:n]) \) for any \( i, j \in w \).

\textbf{Proof.} Follows from Definition \ref{def:splitting} because \( SF_1(i) = \text{alph}(w[i:n]) \). \hfill \Box

This means that two positions \( i, j \) are 1-equivalent if and only if every letter occurring between the two (in \( w[i:j-1] \)), also occurs in the suffix starting on the larger position (in \( w[j:n] \)). In the case \( j = i+1 \) this leads to: \( i \sim_1 i+1 \) if and only if \( w[i] \in \text{alph}(w[i+1:n]) \). Using Corollary \ref{cor:1-splitting} we can find the 1-splitting positions of a word by going right to left and starting a new 1-block whenever we encounter a character we haven’t seen before.

For \( k \geq 1 \) and a \( k \)-block \( a \) with \( |a| \geq 2 \), we can find its \((k+1)\)-splitting positions by basically using the same idea: instead of going through the whole block, we slice off its rightmost character.
Lemma 3. For \( k \geq 1 \) and a \( k \)-block \( a \) in \( w \) with \( |a| \geq 2 \), we let \( a = a^\dagger x \) for a suitable \( x \in \Sigma \). Furthermore, let \( n_a \) be the position of \( x \) in \( w \), and \( i, j \in a^\dagger \). Then the following hold: (i) \( n_a \sim_{k+1} i \) for all \( i \in a^\dagger \), (ii) \( i \sim_{k+1} j \) if and only if \( SF_1(i, a^\dagger) = SF_1(j, a^\dagger) \).

Proof. We begin by showing (i).

Let \( i \in a^\dagger \), \( v' \in SF_k(i + 1) \), and \( v = w[i]v' \). Assume \( i \sim_{k+1} n_a \). Now, \( v \in SF_{k+1}(i) \Rightarrow v \in SF_{k+1}(n_a) \Rightarrow v' \in SF_k(n_a + 1) \). As \( v' \) was chosen arbitrarily, we have \( i + 1 \sim_{k+1} n_a + 1 \). This is a contradiction, so the statement (i) holds.

To show (ii), we will prove the contrapositive \( i \sim_{k+1} j \iff SF_1(i, a^\dagger) \neq SF_1(j, a^\dagger) \).

1. Let \( i \sim_{k+1} j \) we will show that \( SF_1(i, a^\dagger) \neq SF_1(j, a^\dagger) \):
   - Let \( i < j \) and \( i \sim_{k+1} j \) and hence \( SF_{k+1}(i) \supseteq SF_{k+1}(j) \) and w.l.o.g. let \( v = cv \) for a \( c \in \Sigma \). Now \( v' \in SF_k(i) = SF_k(j) \). Hence \( v' \in SF_k(j) \), but \( v \notin SF_{k+1}(j) \).

   We will show that \( c \) occurs between \( i \) and \( j \), and that it does not occur in \( SF_1(j, a^\dagger) \).

   Firstly, let \( j' \) be the minimal index larger than \( i \), such that \( w[j'] = c \). If we assume \( j' \geq j \), then every subsequence in \( SF_{k+1}(i) \), whose first letter is a \( c \), would actually start at position \( j' \):

   \[
   v \in SF_{k+1}(i) \Rightarrow v \in SF_{k+1}(j') \Rightarrow v \in SF_{k+1}(j). \]

   Hence \( c \in \text{alph}(w[i : j - 1]) \) and therefore \( c \in SF_1(i, a^\dagger) \). Secondly, assume \( c \in SF_1(j, a^\dagger) \) and let \( j' \) be the minimal index larger than \( j \), such that \( w[j'] = c \). By assumption, we have \( j' < j \), hence \( j' + 1 \in a \), hence \( j \sim_k j' + 1 \).

   \[
   v' \in SF_k(j) \Rightarrow v' \in SF_k(j' + 1) \Rightarrow v \in SF_{k+1}(j') \Rightarrow v \in SF_{k+1}(j). \]

   Hence \( c \notin SF_1(j, a^\dagger) \), ergo \( SF_1(i, a^\dagger) \neq SF_1(j, a^\dagger) \), which concludes the first part of the second proof.

2. Let \( SF_1(i, a^\dagger) \neq SF_1(j, a^\dagger) \). We will show \( i \sim_{k+1} j \):

   We have \( i < j \) and \( SF_1(i, a^\dagger) \neq SF_1(j, a^\dagger) \). Hence \( SF_1(i, a^\dagger) \supseteq SF_1(j, a^\dagger) \). W.l.o.g. we choose \( c \in SF_1(i, a^\dagger) \setminus SF_1(j, a^\dagger) \) and let \( i', j' \) be the last/first position to the left/right of \( j \) respectively. To begin with, \( j' \) exists because, if there was no \( c \) after \( j \), then \( i \) and \( j \) would be in different 1-blocks of \( w \). Note that \( j' \notin a^\dagger \) because \( c \notin SF_1(j, a^\dagger) \). Let \( v' \in SF_k(i' + 1) \), then \( cv' \in SF_{k+1}(i') \). Assume \( cv' \in SF_{k+1}(j') \). Then \( cv' \in SF_{k+1}(j') \), so \( v' \in SF_k(j' + 1) \). We then obtain \( SF_k(j' + 1) = SF_k(i' + 1) \), a contradiction.

   Thus, \( cv \notin SF_{k+1}(j) \), but \( cv \in SF_{k+1}(i) \subset SF_{k+1}(i) \). In conclusion, we get \( i \sim_{k+1} j \).

We can now conclude that statement (ii) holds. □

Remark 3. A consequence of Lemma 3 is that any \( k \)-block which is not a singleton will be non-trivially split into \((k + 1)\)-blocks, i.e., it will be partitioned in at least \( 2 \) \((k + 1)\)-blocks.

An ordered rooted tree is a tree with a special node called root, and where the order of the subtrees of a node is important (i.e., the children of each node form a totally ordered set). We say that the level of a node is the length of the unique simple path from the root to that node. Generally, the nodes with smaller levels are said to be higher (the root is the highest node), while the nodes with greater levels are lower in the tree.
**Definition 4.** The Simon-tree $T_w$ associated to the word $w$, with $|w| = n$, is an ordered rooted tree. The nodes represent $k$-blocks of $w$, for $0 \leq k \leq n$, 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 $b$ on level $k - 1$, which represents a $(k - 1)$-block $[i : j]$ with $i < j$, the children of $b$ are exactly the blocks of the partition of $[i : j]$ in $k$-blocks, ordered decreasingly by their starting position.
- For $k > 1$, each node on the level $k - 1$ which represents a $(k - 1)$-block $[i : i]$ is a leaf.

For simplicity, the nodes on the $k^{th}$ level of $T_w$ are called explicit $k$-nodes (or simply $k$-nodes); by abuse of notation, we identify each $k$-node by the $k$-block it represents.

**Remark 4.** Note that, 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.

**Remark 5.** Note that, the $k^{th}$ level of the Simon-tree $T_w$ associated to the word $w$ does not necessarily contain all the $k$-blocks of $w$. The singleton-$j$-blocks, for $j < k$, are also $k$-blocks, but they do not appear explicitly as nodes on level $k$ of 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 $k > j$; the only (implicit) child of a $k$-node $[i : i]$ is the implicit $(k + 1)$-node $[i : i]$. Moreover, the $k^{th}$ level of the Simon-tree $T_w$ contains 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.

**Remark 6.** By Lemma 3, all blocks $[i : i]$ of $w$ will appear explicitly exactly once in $T_w$. Moreover, if the intersection of an $i$-block and a $j$-block, with $i \leq j$, is non-empty, then the $j$-block is included in the $i$-block.

We are interested in constructing the Simon-tree associated to a word $w$, with $|w| = n$. Let $N_i := \{ j \mid w[j] = w[i], j > i \}$ the set of all positions to the right of $i$ with the same character as on position $i$. We define the array $X$ with $n$ positions: $X[i] = \min N_i$, if $N_i \neq \emptyset$ (and $X[i] = \infty$ if $N_i = \emptyset$). One can show the following:

**Lemma 4.** Given a word $w$ with length $|w| = n$. The array $X$ can be calculated in $O(n)$ time and space for the entire word.

**Proof.** The word $w$ needs to be traversed only once from right to left by maintaining an array of size $|\Sigma|$ with the last occurrence of each character. Since $|\Sigma| \leq n$, the results follow immediately. ☐

Further, we proceed as follows. We incrementally build $T_w$ by inserting the letters of a word $w\$ (where $\$ is a letter not contained in alph($w$), needed for technical reasons) from right to left. Recall that the nodes of the Simon-tree are blocks inside the word, i.e., intervals $[\ell : r]$ where $\ell$ and $r$ are positions of the word. During the construction of the Simon-tree we will enforce the following invariant: for all nodes that are not on the leftmost branch in the tree we store explicitly both ends of the blocks that define them, while for the nodes on the leftmost branch we only store the rightmost end of the blocks that define them, as the leftmost end is, implicitly, the position of the last symbol we read.

For each insertion we use the subprocedures `findNode` and `splitNode`. When reading a new letter of the word (say the letter $w[i]$), `findNode` traverses bottom-up the leftmost branch of the current tree (so, starting from the leaf on this branch). Let $i + 1$ and $r$ be the interval borders of the current node (the leftmost end is $i + 1$ as it is the position of the symbol we last
As long as $X[i] < r$, we store $i+1$ as the leftmost end of the block represented by the current node, and then move to the parent of the current node. The last node we visit in this traversal of the leftmost branch during \textit{findNode} is either the root of the entire tree, or a node for which $X[i] \geq r$; in any case, the block represented by it does not have an explicitly stored left end. We apply \textit{splitNode} to it. If this node’s block is of size 1 (i.e., it was a leaf representing the block $[i+1 : i+1]$), then \textit{splitNode} adds two new nodes $[i : i]$ and $[i+1 : i+1]$ as left and right children, respectively, of the node we split. Otherwise (the node we found with \textit{findNode} represents a block $[i+1 : r]$ with $r > i+1$) we only add a leftmost child to it, representing the block $[i : i]$. In both cases, only the rightmost end of the interval $[i : i]$ is stored explicitly). Note also that \textit{splitNode} removes (from the leftmost branch) all nodes that were traversed during the execution of \textit{findNode}, except for the node on which \textit{findNode} stopped; now all the nodes removed from the leftmost branch have both ends of their blocks explicitly stored, so the invariant is preserved.

After all positions of the word $w$ are processed, we can remove the tree-node corresponding to the symbol `$` and adapt the blocks stored in all other nodes accordingly.

Given a word $w$, $|w| = n$, we can construct the Simon-tree $T_w$ in $O(n)$ time.

\textit{Proof.} According to the discussion above, we define the following algorithm.

\textbf{Algorithm 1: Building the Simon-tree $T_w$ for a word $w$}

\begin{itemize}
\item \textbf{Input:} $w$
\item \textbf{Result:} Simon-tree $T_w$
\end{itemize}

1. Let $w' = w''$; let $T$ be the tree with the root associated to the block $[n+1 : n+1]$ of $w'$; Let $p$ be a pointer to the root of $T$;
2. Compute the array $X[i]$ using Lemma 4;
3. for $i = n$ to 1 do
4. \hspace{1em} $(T, p) \leftarrow \text{findNode}(T, p, i)$;
5. end
6. Remove the node associated to $[n+1 : n+1]$ from $T$ and set all right ends $r$ of blocks to $\max\{n, r\}$;
7. return $T$;

The two functions we use, namely \textit{findNode} and \textit{splitNode}, are defined as follows.

\textbf{Algorithm 2: findNode}

\begin{itemize}
\item \textbf{Input:} Simon-tree $T_{w[i+1:n]}$, Pointer to leftmost leaf $a$ of $T_{w[i+1:n]}$, Position $i$ of $w$
\item \textbf{Result:} Simon-tree $T_{w[i:n]}$ and a pointer to its leftmost leaf
\end{itemize}

1. while $a$ is not the root of $T_{w[i+1:n]}$ do
2. \hspace{1em} Let $r$ be the right end of the block represented by $a$;
3. \hspace{1em} if $X[i] < r$ then
4. \hspace{2em} Set the left end of the block represented by $a$ to be $i$;
5. \hspace{2em} $a \leftarrow \text{parent}(a)$;
6. \hspace{1em} else
7. \hspace{2em} Let $p$ be an uninitialized node-pointer and $T$ an uninitialized tree;
8. \hspace{2em} $(T, p) = \text{split}(T_{w[i+1:n]}, a, i)$;
9. end
10. return $(T_{w[i:n]}, p)$

8
Algorithm 3: splitNode

**Input:** Simon-tree $T_{w[i+1:n]}$, Pointer to node $a$ on leftmost branch of $T_{w[i+1:n]}$, Position $i$

**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
   3. Replace $a$ by a node $c$ representing the block $[i : i + 1]$ (but only storing the rightmost end of the block);
   4. Set $c$ such that if has the left child $b$, associated to the block $[i : i]$ (but only storing the rightmost end of the block) and the right child $a'$, associated to the block $[i + 1 : i + 1]$ (storing both ends of the block);
5. else
   6. Add a leftmost child $b$ to $a$, associated to the block $[i : i]$ (but only storing the rightmost end of the block);
7. return $(T, \text{pointer to } b)$

The fact that this algorithm is correct essentially follows from Lemma 3. We concatenate the letter $\$ at the end of $w$ to ensure that all the positions of $w$ are treated in a uniform way (otherwise, we would have needed a special case for the processing of the last letter of $w$, namely $w[n]$). Now assume that we have the tree $T_{w[i+1:n]}$, for some $i \leq n$; for $i = n$, we will just have the tree $T_\$.

It is not hard to see that prepending a new letter to the word $w[i + 1 : n]\$ can only affect the leftmost blocks on each level of the tree, as the membership of a position in a block is only determined by the letters occurring after it. These blocks are found on the leftmost branch of $T_{w[i+1:n]}$. We need to find the largest level $k$ such that $i$ is included in the leftmost block on all levels $j \leq k$ and it is a separate block in all levels $j > k$. By Lemma 3, on the level $k$ we need to find, we will have that the letter $w[i]$ already appears in the leftmost block (but not on its rightmost position), while on the lower levels this is not true. Thus, the search of $k$ is clearly correctly implemented in the function findNode. Once we found this $k$, and the corresponding node $a$ from the leftmost branch on level $k$, we only need to add the block $[i : i]$ as its leftmost child on level $k + 1$. This is easily done by function splitNode (which only analyses the two cases: is $a$ a leaf or not).

For each position we do the following: traverse $t$ nodes while going up on the leftmost branch, then insert one leaf on the leftmost branch, while removing the $t$ traversed nodes from the leftmost branch (they will now be all right from the inserted leaf). As each position is processed exactly once, it is not hard to see that the complexity of the algorithm constructing the Simon-tree associated to a word of length $n$ is $O(n)$.

The following example shows how the Simon tree of a word looks.

**Example 1.** Consider the example word $w = bacbaabada$.

| position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|----------|---|---|---|---|---|---|---|---|---|----|----|
| $w$      | b | a | c | b | a | a | b | a | d | a  | $\$|
| $X$      |   | 4 | 5 | $\infty$ | 7 | 6 | 8 | $\infty$ | 10 | $\infty$ | $\infty$ | $\infty$ |

We get the following tree, where we represent each node $[i : j]$ by the word $w[i : j]$.
In our algorithms, we identify each node with the block \([i : j]\) it represents.

Remark 7. Some properties of Simon-trees are easy to note. Let \(T\) be a rooted tree with \(n\) nodes. The Lowest Common Ancestor \(LCA_T(i, j)\) of two nodes \(i, j\) is the root of the lowest subtree of \(T\) which contains both nodes \(i\) and \(j\). We can show (see below) that if \(a, b\) are two nodes of \(T_w\), \(a \neq b\), associated to the blocks \([i : i]\) and \([j : j]\) respectively, with \(i \neq j\), then \(level(LCA_{T_w}(a, b)) = k\) if and only if \(k\) is the maximum number such that \(i \sim_k j\). Therefore, to compute the length of the shortest distinguishing word between two suffixes \(w[i : n]\) and \(w[j : n]\) of \(w\), it is enough to compute \(LCA_{T_w}(i, j)\). According to [2], after a linear time preprocessing of \(T_w\), such queries can be answered in \(O(1)\) time.

Let \(T_w\) be the Simon-tree of the word \(w\) and take two nodes \(a, b \in T_w\), \(a \neq b\), associated to the blocks \([i : i]\) and \([j : j]\) respectively, with \(i \neq j\). Then \(level(LCA_{T_w}(a, b)) = k\) if and only if \(k\) is the maximum number such that \(i \sim_k j\).

Proof. This is a simple consequence of the fact that \(c = LCA_{T_w}(a, b)\) is associated to a block that contains both \(i\) and \(j\), so if \(k\) is its depth then \(i \sim_k j\), and no other node, which is not a predecessor of \(c\) in the tree, contains both \(i\) and \(j\), so they are separated w.r.t. the equivalence \(\sim_j\) for all \(j > k\).

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 \(t\) and establish a connection between their nodes.

In our solution of MAXSIMK we construct a relation (called S-connection, abbreviation from Simon-connection) between the explicit and implicit nodes of the Simon-trees \(T_s\) and \(T_t\).
Definition 5. 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 $b$.

Each explicit or implicit $k$-node of $T_s$ can be S-connected to at most one $k$-node of $T_t$. If two nodes are S-connected, the corresponding blocks are said to be S-connected too. If two $k$-nodes $a$ and $b$ are S-connected, we say that $b$ is $a$’s S-connection (and vice versa).

Remark 8. 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 does not have any children (i.e., this procedure is not applied recursively), and the block $a$ does not explicitly appear on any level $k' > k + 1$. By Remark 8, each singleton-block $[i : i]$ of $s$ (respectively, $t$) appears now exactly twice in $T_s$ (respectively, $T_t$).

This transformation has the following 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.

Remark 9. 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 $n_b < n_d$. Accordingly, we say that the S-connection is non-crossing.

The S-connection will be constructed in the following way. Initially, we compute for each explicit node of $T_s$ a unique node of $T_t$ to which it can (potentially) be S-connected. In this way, we establish a coarser relation (called P-connection, abbreviation from potential connection) that covers the S-connection. Then we attempt to determine and split, for each level $k$ from 1 to maximally $n$, in increasing order, 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. A special case is the handling of singleton-blocks: they appear explicitly on the level of the tree when they are created and on the next level, but they are only implicit nodes on lower levels. So, two explicit $\ell$-nodes representing singleton-blocks $[i : i]$ (of $s$) and $[j : j]$ (of $t$) can be S-connected, but, for some $k > \ell$, it might be the case that the nodes associated to these blocks (out of which at least one is implicit) are not S-connected anymore, as $s[i : n] \sim_k t[j : n']$. Thus, when we consider level $k$, after splitting explicit $k$-nodes, we also need to consider pairs of singleton-blocks $[i : i]$ and $[j : j]$ whose nodes occurred explicitly on some level $\ell < k$ in their respective trees, where they were S-connected, but at least one of them does no longer occur explicitly on level $k$. If for such a pair $k$ is the smallest number such that $s[i : n] \sim_k t[j : n']$, then we need to split them and also somehow store that $k$ was the level on which the split took place. To allow for a general treatment of all the cases, we will consider that if a pair of singleton-nodes $[i : i]$ and $[j : j]$ occurring on explicitly on level $\ell$ of the two trees, respectively, is in P-connection then it is a P-connected pair (with at least one of the nodes in the pair implicit) on all the levels $\ell > k$. This pair is also S-connected up to the level $k$ such that $s[i : n] \sim_k t[j : n']$; then it is explicitly removed from the S-connection.
In this way, we compute exactly the S-connection. The computation is simple for $k = 1$. Then, we proceed by considering the values of $k$, from 2 to $n$, and determining the pairs of $k$-blocks (that is, pairs of both explicit and implicit $k$-nodes) that need to be split, according to the pairs of $(k-1)$-blocks that were split. We emphasize that, in general, the $k$-blocks which we split are explicitly found on level $k$ of the two trees. However, we might also split pairs of nodes out of which at least one is implicit, representing singletons, both occurring explicitly on other levels closer to the root. For such a pair of nodes, which might be correctly S-connected up to a certain level and split afterwards, we store the level on which it is split.

A first step towards efficiently implementing the above idea is to formulate a lemma allowing us to both define the P-connection and define a splitting criterion for $k$-blocks.

We introduce first a notation. For $w \in \{s, t\}$, a position $j \in w$, and a letter $x$, we define $\text{next}_w(j, x)$ as the leftmost position where $x$ occurs in $w[j : |w|]$, or as $\infty$ 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 $\text{next}_w(a, x) = \text{next}(n_a, x)$. We generally omit the subscript $w$ as it is clear from the context to which we refer.

**Lemma 5.** 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 $a' \sim_{k+1} b'$ if and only if there exists a letter $x$ such that $\text{next}(a', x) + 1 : n] \sim_k t[\text{next}(b', x) + 1 : n']$.

**Proof.** Let us assume $a' \sim_{k+1} b'$ and show that there exists a letter $x$ such that $\text{next}(a', x) + 1 : n] \sim_k t[\text{next}(b', x) + 1 : n']$. As $a' \sim_{k+1} b'$, we can assume without loss of generality that there exists a word $w$, with $|w| = k + 1$, such that $w \in SF_{k+1}(n_{a'}, s) \setminus SF_{k+1}(n_{b'}, t)$. Let $x$ be the first letter of $w$, i.e., $w = xw'$ for some word $w'$. If $s[\text{next}(a', x) + 1 : n] \sim_k t[\text{next}(b', x) + 1 : n']$ would hold, then $w'$ would be a subsequence of length $k$, in both $s[\text{next}(a', x) + 1 : n]$ and $t[\text{next}(b', x) + 1 : n']$. Thus, $w = xw'$ would be a subsequence of both $s[\text{next}(a', x) : n]$ and $t[\text{next}(b', x) : n']$, so $w \in SF_{k+1}(n_{b'}, t)$. Thus, $s[\text{next}(a', x) + 1 : n] \sim_k t[\text{next}(b', x) + 1 : n']$

Now, let there be a letter $x$ such that $SF_k(\text{next}(n_{a'}, x) + 1, s) \neq SF_k(\text{next}(n_{b'}, x) + 1, t)$. Then we can assume without losing generality that there exists $w'$ of length $k$ in $SF_k(\text{next}(n_{b'}, x) + 1, s) \setminus SF_k(\text{next}(n_{b'}, x) + 1, t)$. Clearly, $w = xw'$ is in $SF_k(\text{next}(n_{a'}, x), s) \subseteq SF_k(\text{next}(n_{a'}, x) s)$ but $w = xw' \notin SF_k(\text{next}(n_{b'}, x), t)$, so $w \notin SF_k(n_{b'}, t)$. This means that $s[\text{next}(a', x) : n] \sim_{k+1} t[\text{next}(b', x) : n']$. \hfill \Box

The main idea of this lemma (pictured in the left part of Figure 1) is that two $(k+1)$-blocks $a'$ and $b'$ are not S-connected, although their parents were S-connected, iff we can find a letter $x$ such that $s[\text{next}(a', x) + 1 : n]$ and $t[\text{next}(b', x) + 1 : n']$ are not $k$-equivalent, but are $(k-1)$-equivalent. That is, $\text{next}(a', x) + 1$ and $\text{next}(b', x) + 1$ 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 $s[\text{next}(a', x) + 1 : n]$ and $t[\text{next}(b', x) + 1 : n']$.

The following lemma, strongly related to Lemma 5, is particularly useful to define the P-connection. Because Lemma 6 is not both necessary and sufficient (unlike, e.g., Lemma 3), it can only be used to define a relation shorter than the S-connection, so it cannot be used to characterise it. Recall that in Simon-trees the children of a node are numbered right to left.

**Lemma 6.** 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 $T_a$ can only be $S$-connected (but it is not necessarily connected) to the $i$th child of the node $b$ of $T_b$, for all $i \geq 1$. Moreover, for $i > 1$, if the $i$th child of the node $a$ is the node $a' = [m_{a'} : n_{a'}]$ and the $i$th child of the node $b$ is the node $b' = [m_{b'} : n_{b'}]$ and $a' \sim_{k+1} b'$, then $\text{alph}(s[n_{a'} : n_a - 1]) = \text{alph}(t[n_{b'} : n_b - 1])$.}

\hfill 12
Remark 10 which are not S-connected on this level, and also we split two $k_{a, b}$, we can decide that a pair of nodes $(a', b')$ is not S-connected. As our approach is to refine the P-connection till the S-connection is reached, we need to have $j = i$. 

The same proof actually shows the stronger result: if $a'$ and $b'$ are connected, then we need to have $\text{alph}(s[n_{a'} : n_{a} - 1]) = \text{alph}(t[n_{b'} : n_{b} - 1])$. 

Now we are able to formally define recursively the P-connection for the words $s$ and $t$:

**Definition 6.** 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 viceversa).

Remark 10. It is not hard to see that, in the spirit of Remark 9, the P-connection is non-crossing. Moreover, by Lemma 8 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. As our approach is to refine the P-connection till the S-connection is reached, we can decide that a pair of nodes $(a, b)$ is not in the S-connection when the pair consisting of the respective parent-nodes of $a$ and $b$ are not S-connected.

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. In 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. For simplicity of exposure, when we split two $k$-blocks, we say that we $k$-split them.

The 1-blocks (which all occur explicitly on level 1 of $T_s$ and respectively $T_t$) which are S-connected are easily 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 $\text{alph}(s[n_{a'} : n_{a}]) = \text{alph}(t[n_{b'} : n_{b}])$. All the other P-connected pairs of 1-blocks are not S-connected, so they are 1-split.

The identification of the pairs of $(k + 1)$-blocks and pairs of singletons which need to be $(k+1)$-split is based on Lemma 8. The idea is the following. A pair of P-connected $(k+1)$-blocks $a' = s[m_{a'} : n_{a'}]$ and $b' = t[m_{b'} : n_{b'}]$ is not S-connected if and only if there exists a letter $x$ such that $s[next(a', x) + 1 : n] \sim_k t[next(b', x) + 1 : n']$. 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 $j$, with $i = \text{next}(a', x) + 1$ and $j = \text{next}(b', x) + 1$ 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 block $[i : i]$. On the highest level where this block explicitly appeared, it was S-connected to a node $[j : j]$ representing a singleton too, according to Remark 8 and Lemma 8. 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 \( \ell \). 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 \( B = [m_B : n_B] \) 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 \( b = [m_b : n_b] \) the \( \ell \)th child of \( A \) and \( B \), respectively. Clearly, \( b \) might be explicit, implicit, or even empty. The following hold. All positions of \( a \) are \( k \)-split from the positions \([n_B : m_b - 1]\) 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 \( b \).

Taking \((i, j)\) to be each position of \( a \) paired with each the positions from which it is split, according to the above, might not be efficient. Lemma 7 explains how we can actually choose \( i \) inside the block \( a \) (and the corresponding \( j \)) efficiently in both cases (A) and (B).

But first we need a few more notations. For a block \( a = [m_a : n_a] \) of \( s \) or \( t \) and a letter \( x \), let \( \text{prev}(a, x) \) 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 \( \text{right}(a, x) \) 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 7 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 \( i = \text{next}(a', x) + 1, j \in [m_a : n_a], j = \text{next}(b', x) + 1, \) and \( i \sim_k j \).

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 two \( k \)-blocks which are S-connected. Otherwise, they would have already been split.

**Lemma 7.** For \( k \geq 1 \), a pair of P-connected \((k + 1)\)-blocks \( a' = [m_{a'} : n_{a'}] \) (occurring in \( s \), whose parent is the \( k \)-block \( a = [m_a : n_a] \)) and \( b' = [m_{b'} : n_{b'}] \) (occurring in \( t \), whose parent is the \( k \)-block \( b = [m_b : n_b] \)) is \((a, k + 1)\)-split if and only if there exists a letter \( x \) such that one of the following holds:

1. \( x \) is a letter occurring in \([m_a - 1 : n_a - 1]\), \( a ' \) ends strictly between \( \text{prev}(a, x) \) and \( m_a \) (i.e., \( \text{prev}(a, x) < n_{a'} < n_a \)), and \( b' \) ends to the left of \( \text{prev}(b, x) + 1 \) (i.e., \( n_{b'} \leq \text{prev}(b, x) \)).
2. \( x \) is a letter occurring in \([m_a - 1 : n_a - 1]\), \( a ' \) ends strictly between \( \text{prev}(a, x) \) and \( \text{right}(a, x) + 1 \) (i.e., \( \text{prev}(a, x) < n_{a'} \leq \text{right}(a, x) \)), and \( b' \) ends strictly between \( \text{right}(b, x) \) and \( n_b \) (i.e., \( \text{right}(b, x) < n_{b'} \leq n_b \)).
3. \( a \sim_k b, x \) is a letter occurring in \([m_a - 1 : n_a - 1]\), \( a ' \) ends strictly between \( \text{prev}(a, x) \) and \( \text{right}(a, x) + 1 \) (i.e., \( \text{prev}(a, x) < n_{a'} \leq \text{right}(a, x) \)), and \( b' \) ends between \( \text{prev}(b, x) + 1 \) and \( \text{right}(b, x) \) (i.e., \( \text{prev}(b, x) < n_{b'} \leq \text{right}(b, x) \)).

**Proof.** Assume \( A = [m_A : n_A] \) and \( B = [m_B, n_B] \) are two \((k - 1)\) (implicit or explicit) blocks from \( T_s \) and \( T_t \), respectively, and \( a = [m_a : n_a] \) and \( b = [m_b, n_b] \) the \( \ell \)th children of \( A \) and \( B \), if \( A \) or, respectively, \( B \) is implicit, then \( a \) or, respectively, \( b \) is also implicit.

Let us consider a pair of P-connected \((k + 1)\)-nodes \( a' = [m_{a'} : n_{a'}] \) in \( s \) and \( b' = [m_{b'} : n_{b'}] \) in \( t \) which is \((a, k + 1)\)-split. Then there exist two positions \( i \) and \( j \), with \( i = \text{next}(a', x) + 1 \) and \( j = \text{next}(b', x) + 1 \) which were \( k \)-split but whose \((k - 1)\)-blocks are S-connected, and, moreover, \( i \) is in \( a \). This means \( j \) must have been in \( B \). There are three cases to analyse.

Case 1 occurs when \( j = \text{next}(b', x) + 1 \) is a position of \([m_B : m_b - 1]\). As \( i = \text{next}(a', x) + 1 \) is in \( a \), we get that \( a' \) ends after \( \text{prev}(a, x) \). Also, \( b' \) is P-connected to \( a' \) and the P-connection is non-crossing, so \( a' \) should also end before \( m_a \). Moreover, \( b' \) ends before \( j = \text{next}(b', x) + 1 \), and, as \( \text{next}(b', x) \) returns the position of a letter \( x \) in the interval \([m_B - 1 : m_b]\), we get that \( j \leq \text{prev}(b, x) + 1 \).
Case 2 occurs when \( j = \text{next}(b', x) + 1 \) is a position of \([n_b + 1 : n_B]\). The analysis of where \( a' \) may occur is similar to the first case above: as \( i = \text{next}(a', x) + 1 \) is in \( a \), we get that \( a' \) ends after \( \text{prev}(a, x) \). As \( i \) is in \( a \), we get that \( a' \) cannot end to the right of the rightmost \( x \) in \( a \), so \( \text{prev}(a, x) < n_{a'} \leq \text{right}(a, x) \). Now, next \((b', x) + 1\) is in \([n_b + 1 : n_B]\), which means that next \((b', x)\) is not in \( b \), so next \((b', x) > \text{right}(b, x)\). As \( b' \) is \( P \)-connected to \( a \), and the \( P \)-connection is non-crossing, we get that \( b' \) cannot end to the right of \( n_b \).

Case 3 occurs when \( j \) is inside \( b \) and \( a \sim_k b \) (so that \( i \) and \( j \) are \( k \)-split). The analysis is, however, very similar to the above, and the conclusion follows in the same way.

We will now show the converse. In case 1, we have that next \((a', x) + 1\) is \( k \)-split from next \((b', x) + 1\), as the former is a position inside \( a \) and the latter is a position to the left of \( m_b - 1 \). In case 2, we have that next \((a', x) + 1\) is \( k \)-split from next \((b', x) + 1\), as the first is a position inside \( a \) and the second one is a position strictly to the right of \( n_b \). In case 3, we have that next \((a', x) + 1\) is \( k \)-split from next \((b', x) + 1\), as next \((a', x) + 1\) is a position inside \( a \) and next \((b', x) + 1\) is a position inside \( b \), and \( a \) and \( b \) are \( k \)-split. \( \square \)

Remark 11. In Lemma 7 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 alph \((s[n_{a'} : n])\) = alph \((t[n_{b'} : n']\)). This means, in particular, that next \((a', x)\) \( \neq \infty \) for some letter \( x \) if and only if next \((b', x)\) \( \neq \infty \).

Now, we can explain how to algorithmically find the pairs of \((k + 1)\)-blocks which should be split. For this, we computed in the previous step (when we computed the \( k \)-split pairs) the list \( L \) of pairs of singleton \( k \)-blocks which were \( k \)-split, as well as a list \( H \) of all the explicit \( k \)-nodes of \( T_s \) paired to the nodes of \( T_t \) to which they are \( P \)-connected.

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 \( x \in \text{alph}(a) \cup \{s|m_a - 1\} \) (note that the symbols \( x \in \text{alph}(a) \) can be identified as the first symbols of the \((k + 1)\)-blocks into which \( a \) is split, that is, the children of node \( a \)) we do the following:

1. If \( x \) is a letter occurring in \( s[m_a - 1 : n_a - 1] \), identify each \((k + 1)\)-block \( a' = [m_{a'} : n_{a'}] \) with \( \text{prev}(a, x) < n_{a'} < m_a \) and its pair \( b' = [m_{b'} : n_{b'}] \). Then \((a', b')\) is not in the \( S \)-connection if \( n_{b'} \leq \text{prev}(b, x) \) (i.e., \( a' \) and \( b' \) are \((a, k + 1)\)-split).

2. If \( x \) is a letter occurring in \( s[m_a - 1 : n_a - 1] \), identify each \((k + 1)\)-block \( a' = [m_{a'} : n_{a'}] \) with \( \text{prev}(a, x) < n_{a'} \leq \text{right}(a, x) \) and its pair \( b' = [m_{b'} : n_{b'}] \). Then \((a', b')\) is not in the \( S \)-connection if \( \text{right}(b, x) < n_{b'} \leq n_b \).

3. If \( a \sim_k b \) and \( x \) is a letter occurring in \( s[m_a - 1 : n_a - 1] \), identify each \((k + 1)\)-block \( a' \) with \( \text{prev}(a, x) < n_{a'} \leq \text{right}(a, x) \) and its pair \( b' = [m_{b'} : n_{b'}] \). Then \((a', b')\) is not in the \( S \)-connection if \( \text{prev}(b, x) < n_{b'} \leq \text{right}(b, x) \).
The blocks \( a' \) and their P-connections \( b' \), identified in each of the three cases above, form contiguous sequences of blocks in \( T_s \) and \( T_t \), respectively, since the P-connection is non-crossing.

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

In general, 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.

We now need to implement this idea efficiently, i.e., to describe how the \((k + 1)\)-blocks \( a' \) and \( b' \) are identified. We say that the 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 \( T_s \). The following technical tools show that we can efficiently compute all information needed to use Lemma 7.

The first lemma is not hard to show.

**Lemma 8.** 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 process the trees \( T_s \) and \( T_t \) in \( O(n + n') \) time such that the following information can be retrieved in \( O(1) \) time:

1. For an (explicit or implicit) node \( a \) of \( T_s \), the (explicit or implicit) node \( b \) of \( T_t \) to which it is P-connected.

2. The \( j^{th} \) (from left to right) explicit node on level \( k \) of \( T_s \) (respectively, \( T_t \)). Note that, because \( T_s \) and \( T_t \) are ordered trees, we can uniquely identify the \( j^{th} \) (from left to right) explicit node that occurs on level \( k \).

3. For each position \( i \) of \( s \), the unique position \( U[i] \) of \( t \) such that the singleton node \([i : i]\) is P-connected to the singleton node \([U[i] : U'[i]]\).

4. For each position \( i \) of \( t \), the unique position \( U'[i] \) of \( s \) such that the singleton node \([i : i]\) is P-connected to the singleton node \([U'[i] : U'[i]]\).

5. For each position \( i \) of \( s \) (respectively of \( t \)) and level \( k \), the node associated to the \( k \)-block that starts with \( i \), if such a node exists.

**Proof.** We can directly use Definition 6 to compute the P-connection on levels. The roots of the trees are P-connected. Then, we start the traversal of the trees and, when considering a pair of P-connected nodes, we connect their respective \( i^{th} \) children, for all \( i \). The only aspect that needs to be treated carefully is that each time we place a pair of explicit nodes \([(i : i), [j : j]]\) in the P-connection, we can set \( U[i] = j \) and \( U'[j] = i \), and note that the (implicit or explicit) nodes \([i : i]\) and \([j : j]\) will be P-connected on all lower levels. This solves items 1, 3, 4 of the list in the statement. For simplicity, we can also assume that the P-connection is implemented as a series of arrays \( P_k \) and \( P'_k \) (where \( k \) is a level of \( T_s \)) such that \( P_k[j] = j' \) and \( P'_k[j'] = j \) if and only if the \( j^{th} \) node on level \( k \) of \( T_s \) (from left to right) is connected to the \( j^{th} \) node on level \( k \) of \( T_t \) (also from left to right).

The rest of the proof is given for \( s \) and \( T_s \). An analogous approach works for \( t \) and \( T_t \).

By traversing the tree \( T_s \) on levels (left to right) we can also associate to each explicit node of the tree the pair \((k, j)\), where \( k \) is its level, and \( j - 1 \) is how many explicit nodes occur to the left of that node on its level. As such, we can construct an array for each level of the tree,
namely \( s_k \), such that \( s_k[j] \) is the explicit node associated to the pair \((k, j)\) from \( T_s \). For \( t \) and \( T_t \) we construct the array \( t_k \) for each level of the tree.

This solves item 2 of the above list in linear time.

During the traversal of \( T_s \), we can also compute for each level \( k \) an array \( N_{a,k} \) such that if \([p : q]\) is a \( k \)-block of \( T_s \), then \( N_{a,k}[p] \) is a pointer to the node associated to the block \([p : q]\) of \( T_s \). In \( T_t \), we define the arrays \( N_{t,k} \), where \( N_{t,k}[p] \) points to the node associated to \([p : q]\) from \( T_t \).

So, the whole process takes linear time, and all the desired information can be retrieved in \( O(1) \) using the arrays we constructed. The statement follows.

Lemma 9. Given two words \( s \) and \( t \), with \(|s| = n\) and \(|t| = n', n \geq n'\), and their Simon-trees \( T_s \) and \( T_t \), respectively, we can compute in \( O(n) \) time all the values:

- \( \text{prev}(a, x) \) and \( \text{right}(a, x) \) for \( a = [m_a : n_a] \) a block in \( T_s \) and \( x \in \text{alph}(a) \cup \{s[m_a - 1]\} \).
- \( \text{prev}(b, x) \) and \( \text{right}(b, x) \) for \( b = [m_b : n_b] \) a block in \( T_t \), which is \( P \)-connected to the block \( a = [m_a : n_a] \) of \( T_s \), and \( x \in \text{alph}(a) \cup \{s[m_a - 1]\} \).

Proof. The first observation is that \( \text{right}(a, x) \) for \( a = [m_a : n_a] \) a \( k \)-block in \( T_s \) and \( x \in \text{alph}(a) \cup \{s[m_a - 1]\} \) can actually be computed by looking at the splitting of the block \( a \) into \((k + 1)\)-blocks. By Lemma 8, each position \( \text{right}(a, x) \), with \( x \in \text{alph}(a) \cup \{s[m_a - 1]\} \), is a position where one such \((k + 1)\)-block starts. Similarly, we can compute \( \text{right}(b, x) \) for \( b = [m_b : n_b] \) a \( k \)-block in \( T_t \), which is \( P \)-connected to the block \( a = [m_a : n_a] \) of \( T_s \), and \( x \in \text{alph}(a) \cup \{s[m_a - 1]\} \). Again, these positions are among the positions that split \( b \) into \((k + 1)\)-blocks.

To efficiently manage the \( P \)-connection, we can first compute it using Lemma 8. To retrieve \( \text{right}(a, x) \) in \( O(1) \) time, we store it in the child of \( a \) representing a block whose last letter is \( x \).

To compute all the values \( \text{prev}(a, x) \) for all blocks \( a \) of \( T_s \) and \( x \in \text{alph}(a) \cup \{s[m_a - 1]\} \), we do the following:

1. We represent the query \( \text{prev}(a, x) \) for a block \( a = [m_a : n_a] \) of \( T_s \) and \( x \in \text{alph}(a) \cup \{s[m_a - 1]\} \) as the triple \((x, m_a, n_a)\). For each such triple, we store a pointer to the node \( a \).
2. We radix-sort the triples \( \{(x, m_a, n_a) | a = [m_a : n_a] \text{ is a block of } T_s \text{ and } x \in \text{alph}(a) \cup \{s[m_a - 1]\}\} \) and obtain a list \( L \).
3. For each \( x \in \Sigma \) we select in a list \( L_x \) the contiguous part of \( L \) consisting in all the triples of the form \((x, i, i')\).
4. We initialize an array \( Q \) with \(|\Sigma|\) elements, where \( Q[x] = 0 \) for all \( x \in \Sigma \).
5. We now go through the letters \( s[j] \) of the word \( s \), for \( j \) from 1 to \( n \). If \( s[j] = x \) holds, then we remove from the list \( L_x \) all the triples \((x, i, i')\) with \( i - 1 \leq j \); for each such triple \((x, i, i')\) and the block \( a = [i : i'] \), we set \( \text{prev}(a, x) = Q[x] \). Then, we set \( Q[x] = j \).

It is immediate that the above algorithm computes correctly \( \text{prev}(a, x) \) for all blocks \( a \) of \( T_s \) and \( x \in \text{alph}(a) \cup \{s[m_a - 1]\} \). In order to access these values efficiently, we store \( \text{prev}(a, x) \) in the children of \( a \) ending with \( x \), for \( x \in \text{alph}(a) \), and as a separate satellite value in the node \( a \) for \( x = s[m_a - 1] \). The complexity is clearly linear.

A similar algorithm can be used to compute the values \( \text{prev}(b, x) \) for \( b = [m_b : n_b] \) a block in \( T_t \), which is \( P \)-connected to the block \( a = [m_a : n_a] \) of \( T_s \), and \( x \in \text{alph}(a) \cup \{s[m_a - 1]\} \).
Lemma 10. 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 \( O(|\text{alph}(a)|) \) time the three lists, associated to the pair \((a, b)\), containing:

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

Proof. We first run the algorithm of Lemma 9 and then the required interval-pairs can be clearly computed in \( O(1) \) time per pair. This adds up to \( O(|\text{alph}(a)|) \) time for a pair of \( P \)-connected nodes \((a, b)\).

The final step before stating our main result is the following lemma.

Lemma 11. 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 \( \text{alph}(s[n_a : n]) = \text{alph}(t[n_b : n']) \).

Proof. Let \( \sigma = |\Sigma| \) be the size of the input alphabet. Clearly, we have \( \sigma \in O(n) \) and we can assume that \( \Sigma = [1 : \sigma] \).

Let \( g \) be the number of 1-nodes in \( T_s \) (recall that these nodes are numbered from right to left), and \( g' \) be the number of 1-nodes of \( T_t \). The partition of \( s \) (resp., \( t \)) into 1-blocks is done according to Lemma 3 and it can be retrieved from the first level of the tree \( T_s \) (and \( T_t \) respectively). Indeed, we split the interval \([1 : n]\) into the intervals/blocks corresponding to the children of the 0-node of \( T_s \). We also apply these splits in the split-find structure \( S \).

Let us see now how to synchronise the blocks of the two trees. We use an array \( H \) with \( \sigma \) elements, initially all set to 0. As an invariant, when processing the \( i \)-th block \( a = [m_a : n_a] \) from \( T_s \) and the \( j \)-th block \( b = [m_b : n_b] \) from \( T_t \) we have that, for each letter \( x \in \Sigma \):

- \( H[x] = 0 \) if \( x \notin \text{alph}(s[n_a : n]) \cup \text{alph}(t[n_b : n']) \),
- \( H[x] = 1 \) if \( x \in \text{alph}(s[n_a : n]) \setminus \text{alph}(t[n_b : n']) \),
- \( H[x] = 2 \) if \( x \in \text{alph}(s[n_a : n]) \cap \text{alph}(t[n_b : n']) \),
- \( H[x] = 3 \) if \( x \in \text{alph}(t[n_b : n']) \setminus \text{alph}(s[n_a : n]) \).

Also, we maintain a variable \( \text{check} \) that counts how many odd values are in \( H \). Clearly, \( \text{alph}(s[n_a : n]) = \text{alph}(t[n_b : n']) \) if and only if \( \text{check} \) is 0.

We maintain \( \text{check} \) and \( H \) as follows. For \( i \) from 1 to \( g \), assume that the \( i \)-th node on level 1 of \( T_s \) is \( a = [m_a : n_a] \), and the \( j \)-th node on level 1 of \( T_t \) is \( b = [m_b : n_b] \). If \( H[s[n_a]] = 0 \), then set \( H[s[n_a]] \leftarrow 1 \), and \( \text{check} \) is increased by 1. If \( H[s[n_a]] = 3 \), then set \( H[s[n_a]] \leftarrow 2 \), and \( \text{check} \) is decreased by 1. If \( H[t[n_b]] = 0 \), then set \( H[t[n_b]] \leftarrow 3 \), and \( \text{check} \) is increased by 1. If \( H[t[n_b]] = 1 \), then set \( H[t[n_b]] \leftarrow 2 \), and \( \text{check} \) is decreased by 1. In all other cases we do nothing. After this, if \( \text{check} \) is odd, then \( \text{alph}(s[n_a : n]) \neq \text{alph}(t[n_b : n']) \). Otherwise, \( \text{alph}(s[n_a : n]) = \text{alph}(t[n_b : n']) \). We then consider the next value of \( i \).

The algorithm described above can be clearly implemented in \( O(n + n') \) time.

Based on the previous lemmas, we can now show our main technical theorem. We use Lemma 11 to see which 1-nodes are not \( S \)-connected. 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 10. To do this efficiently, we maintain an interval union-find and an interval split-find structure for each word.

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 the node $[U[i] : U[i]]$.

**Proof.** Let $\sigma = |\Sigma|$ be the size of the input alphabet. Clearly, we have $\sigma \in O(n)$, and we can assume that $\Sigma = [1 : \sigma]$. We first present our algorithm and then show that it fulfils the desired properties.

§ Data structures and preprocessing.

We maintain an interval union-find data structure $\mathcal{U}$ and two interval split-find data structures: $\mathcal{S}_a$ over the universe $[1 : n]$ and $\mathcal{S}_t$ over the universe $[1 : n']$. Initially, $\mathcal{U}$ contains all the separate intervals $[i : i]$ for $i \in [1 : n]$, while $\mathcal{S}_a$ (respectively, $\mathcal{S}_t$) contains just the interval $[1 : n]$ (the interval $[1 : n']$). We assume that the $\text{find}_{\mathcal{U}}(x)$ (respectively, $\text{find}_{\mathcal{S}_a}(x)$ and $\text{find}_{\mathcal{S}_t}(x)$) operation returns the margins of the interval of $\mathcal{U}$ (respectively, $\mathcal{S}_a$ and $\mathcal{S}_t$) which contains $x$.

We will use $\mathcal{U}$ to keep track of the positions of $s$ which were split from the positions to which they are P-connected, while $\mathcal{S}_a$ and $\mathcal{S}_t$ are used to maintain the splitting of $s$ and, respectively, $t$ into $k$-blocks while $k$ is incremented during the algorithm.

We also maintain an array with $n$ components $\text{Level}_a$ and an array with $n'$ components $\text{Level}_t$. Initially, all components are set to $\infty$. At the end of the computation, these arrays will store, for each position $i$ of $s$ (respectively, of $t$), the highest level on which $i$ is contained in an implicit or explicit block of $s$ (respectively, $t$) which is not S-connected to any node of $T_t$ (respectively, $T_a$).

In this initial phase, we compute the Simon-trees $T_s$ and $T_t$ using the algorithm from Theorem 3. We also compute the data structures in Lemmas 8 and 9. Finally, we can also compute the lists of interval-pairs from Lemma 10 (stored, e.g., as three separate lists of interval-pairs) for each pair of P-connected nodes.

§ The main algorithm.

Now we move on to the main phase of the algorithm. The 0-nodes of the two trees are clearly S-connected. In the data structures $\mathcal{S}_a$ and $\mathcal{S}_t$ we split the respective 0-blocks into the corresponding 1-blocks (these are given by their children in $T_a$ and $T_t$).

- **The first step of our algorithm is computing the S-connection between 1-nodes.**

  For $k = 1$, we process $s$, $t$, and their corresponding Simon-trees according to Lemma 11.

  Now, we 1-split the pair $a = [m_a : n_a]$ and $b = [m_b : n_b]$ if and only if $\text{alph}(s[n_a : n]) \neq \text{alph}(t[n_b : n'])$. Further, for all $\ell \in [m_a : n_a]$, we set $\text{Level}_a[\ell] = 1$, as $1$ is the level on which position $\ell$ was split from its P-connection. Similarly, for $\ell' \in [m_b : n_b]$ we set $\text{Level}_t[\ell'] = 1$. We also update the union-find structure $\mathcal{U}$. First, we make the union of the singletons $[i : i]$, for $i$ from $m_a$ to $n_a$. Then, if $\text{Level}_a[m_a - 1] \neq \infty$, we make the union between the interval that contains $m_a - 1$ (returned by $\text{find}_{\mathcal{S}_a}(m_a - 1)$) and the interval that contains $m_a$ (namely, $[m_a : n_a]$). Further, if $\text{Level}_a[n_a + 1] \neq \infty$, then we make the union between the interval that contains $m_a$ (returned by $\text{find}_{\mathcal{S}_a}(m_a)$) and the interval that contains $n_a + 1$ (returned by $\text{find}_{\mathcal{S}_a}(n_a + 1)$). In this way, we ensure that each interval of consecutive positions $i$ of $s$, for which $\text{Level}_a[i] \neq \infty$ and which cannot be extended to the left nor to the right, corresponds to a single interval stored in the union-find structure $\mathcal{U}$.

  If $c = [m_c : n_c]$ is a 1-block of $s$ (or, respectively, $t$) which was not P-connected to a block of $s$ (respectively, $t$), then, for $\ell' \in [m_c : n_c]$, we set $\text{Level}_a[\ell'] = 1$ (respectively, $\text{Level}_t[\ell'] = 1$).
At the end of this step, we collect in a list \( L_1 \) the positions \( i \) for which the 1-blocks \([i : i]\) and \([U[i] : U[i]]\) were 1-split (so, the pairs of singleton-1-blocks which were split). Finally, we split the intervals \([1 : n]\) (and \([1 : n']\)) in the data structures \( S_s \) and \( S_t \) into the corresponding 2-blocks (i.e., each 1-block is split into its children).

We note that at the end of this step we will have that \( \text{Level}_s[i] \neq \infty \) if and only if the position \( i \) of \( s \) was contained in a 1-block of \( T_s \) which was split from the block of \( t \) to which it is P-connected (that is, the respective pair of blocks is not in the S-connection). Clearly, for any 2-block \( a = [m_a : n_a] \) of \( s \), a position \( i \in [m_a : n_a] \) fulfills \( \text{Level}_s[i] \neq \infty \) if and only if all positions \( j \in [m_a : n_a] \) fulfill \( \text{Level}_s[j] \neq \infty \).

The pairs of P-connected 1-blocks which were not split, are S-connected.

- The iterated step of our algorithm is computing the S-connection between \( k + 1 \)-nodes, for \( k \) from 1 to \( d - 1 \), where \( d \) is the last level of \( T_s \).

We can assume that we have the list \( L_k \) containing the pairs of (explicit and implicit) singleton-\( k \)-blocks which were \( k \)-split (that is, split in the previous iteration), as well as the list \( H_k \) of explicit nodes on level \( k \) of \( T_s \) paired with the nodes from level \( k \) of \( T_t \) to which they are P-connected.

So let \((a, b)\) be a pair of nodes from \( L_k \cup H_k \). That is, \( a = [m_a : n_a] \) is a \( k \)-block from \( T_s \) (explicit or implicit), \( b = [m_b : n_b] \) is a \( k \)-block from \( T_t \), and \( a \) and \( b \) are P-connected. Using Lemma 10 in the preprocessing phase, we have computed the following three lists of pairs of intervals:

1. For each letter \( x \) occurring in \( s[m_a - 1 : n_a - 1] \), the interval-pair \([\text{prev}(a, x) + 1 : m_a - 1], [0 : \text{prev}(b, x)]\).
2. For each letter \( x \) occurring in \( s[m_a - 1 : n_a - 1] \), the interval-pair \([\text{prev}(a, x) + 1 : \text{right}(a, x)], [\text{right}(b, x) + 1 : n_a]\).
3. For each letter \( x \) occurring in \( s[m_a - 1 : n_a - 1] \), the interval-pair \([\text{prev}(a, x) + 1 : \text{right}(a, x)], [\text{prev}(b, x) + 1 : \text{right}(b, x)]\).

Let \(([e_1 : e_2], [f_1 : f_2])\) be an interval-pair contained in one of the first two lists. We should \((a, k + 1)\)-split all the pairs of P-connected \((k + 1)\)-blocks \((a', b')\) (explicit or implicit) such that \( a' \) ends inside \([e_1 : e_2]\) and \( b' \) inside \([f_1 : f_2]\). For this, we will use our additional data structures \( \mathcal{U} \) and \( S \).

This is done as follows. Using \( \text{find}_S(e_1) \) and \( \text{find}_S(f_1) \), we compute the \((k + 1)\)-block \([p, q]\) of \( s \) that contains \( e_1 \) and, respectively, the \((k + 1)\)-block \([p', q']\) of \( t \) that contains \( f_1 \). Let \([r : o]\) be the \((k + 1)\)-block of \( s \) which is P-connected to \([p', q']\). We can obtain \([r : o]\) according to Lemma 5 which we already used in the preprocessing phase. If \( r \geq p \), then we set \( p \leftarrow r \) and \( q \leftarrow o \) to ensure that \([p : q]\) is the lefthemost \((k + 1)\)-block of \( s \) that ends to the right or on \( e_1 \), which is P-connected to a block which ends to right or on \( f_1 \).

Further, if \( \text{Level}_s[p] \neq \infty \) then we use \( \text{find}_p(s) \) to identify the interval \([d_1 : d_2]\) of positions of \( s \) which contains \( p \) and has the property that if \( i \in [d_1 : d_2] \), then \( \text{Level}_s[i] \neq \infty \). We then set \( p \leftarrow d_2 + 1 \) and \( q \) as the right margin of the interval \( \text{find}_p(p) \) (i.e., the right margin of the \((k + 1)\)-block starting with \( p \)). Now, we have \( \text{Level}_s[p] = \infty \), and we can continue as described below.

The main part of the processing we do for \( k + 1 \) consists in the following loop.

If \( \text{Level}_s[p] \neq \infty \), then we check whether the \((k + 1)\)-block \([p : q]\) ends in \([e_1 : e_2]\). If yes, we compute the block \([p' : q']\) to which it is P-connected and see if it ends inside \([f_1 : f_2]\). If any of the previous checks is false, then we stop the execution of the loop. If both checks are true, then we decide that \([p : q]\) and \([p' : q']\) are not S-connected (the reason for this being that \( s[\text{next}([p : q], x) + 1 : n] \approx_k t[\text{next}([p' : q'], x) + 1 : n'] \)). Thus, we set \( \text{Level}_s[i] = k + 1 \) for all
$i \in [p : q]$ and Level$_s[i] = k + 1$ for all $i \in [p' : q']$. We also make in $\mathcal{U}$ the union of all the intervals $[i : i]$ with $i \in [p : q]$. Then we make the union of the interval $[p : q]$ and the interval of $p - 1$ if Level$_s[p - 1] \neq \infty$, and the union of the interval of $q$ and the interval of $q + 1$ if Level$_s[q + 1] \neq \infty$.

Then, we consider the $(k + 1)$-block starting with $p + 1$, returned by find$_S(p + 1)$, and repeat the loop for this block in the role of $[p : q]$.

The loop stops when we reached a $(k + 1)$-block $[p : q]$ that ends outside of $[e_1 : e_2]$ or it is P-connected to a block $[p' : q']$ that ends outside $[f_1 : f_2]$. In both cases, we consider a new interval-pair ($[e_1 : e_2]$, $[f_1 : f_2]$) from our lists and repeat the same process described above.

Moving on to the third list, we will process it exactly as the first two lists, but only in the case when it corresponds to a pair $(a, b)$ of $k$-blocks that are not $S$-connected.

Finally, if $a = [m_a : n_a]$ is a block of $s$ (respectively, of $t$) which did not have a pair in the P-connection (due to Lemma 8), we set Level$_s[i] = k + 1$ (respectively, Level$_t[i] = k + 1$) for all $i \in [m_a : n_a]$. If $a$ is a block of $s$, we also make in $\mathcal{U}$ the union of all the intervals $[i : i]$ with $i \in [m_a : n_a]$. Then we make the union of the interval $[m_a : n_a]$ and the interval of $m_a - 1$ if Level$_s[p - 1] \neq \infty$, and the union of the interval of $n_a$ and the interval of $n_a + 1$ if Level$_s[n_a + 1] \neq \infty$. Once more, we ensure that each interval of consecutive positions $i$ of $s$, for which Level$_s[i] \neq \infty$ and which cannot be extended to the left nor to the right, corresponds to a single interval stored in the interval union-find structure $\mathcal{U}$.

Similarly to the first step, at the end of this step, we collect in a list $L_{k + 1}$ the positions $i$ for which the P-connection between the $(k + 1)$-blocks $[i : i]$ and $[U[i] : U[i]]$ were split, and we split the non-singleton $k$-blocks of $s$ and $t$, and the corresponding intervals of $S_s$ and $S_t$, into their $(k + 1)$-blocks (i.e., each $k$-block is split into its children).

As an invariant, at the end of the execution of the iteration for $k$, we will have that Level$_s[i] = j \neq \infty$ if and only if the position $i$ of $s$ was contained in an explicit or implicit $(k + 1)$-block of $T_s$ which was split in the iteration $j - 1$ from the block of $T_s$ to which it is P-connected. Clearly, for any $g$-block $a = [m_a : n_a]$ of $s$, with $g > k + 1$, a position $i \in [m_a : n_a]$ fulfils Level$_s[i] \neq \infty$ if and only if all positions $j \in [m_a : n_a]$ fulfil Level$_s[j] \neq \infty$.

The pairs of P-connected $(k + 1)$-blocks (implicit or explicit) which were not split, are $S$-connected.

§ The output of the algorithm and its correctness.

Our algorithm outputs the $S$-connection we computed as well as the arrays Level$_s$ and Level$_t$; these store for each position $i$ (of $s$ or $t$) the highest level on which the position $i$ is part of a block in its respective tree which is not $S$-connected to any node of the other tree. The correctness of our algorithm follows from Lemmas 5, 6 and 7; we split all pairs of P-connected nodes which are not $S$-connected. Moreover, we do this considering the nodes of the trees in increasing order of their levels, which proves that the arrays Level$_s$ and Level$_t$ are correctly computed.

§ The complexity of the algorithm.

According to Lemma 1 we can assume that constructing the data structures $\mathcal{U}$ and $\mathcal{S}$ takes linear time, and the time needed to execute all the operation on these structures is linear in their number, i.e., each operation takes $O(1)$ amortized time. Therefore, according to the Lemmas 8, 9 and 10 the preprocessing phase takes linear time. In particular, by Lemma 10 we need $O(\alpha(n))$ time to compute the interval-pair lists for two P-connected nodes $a$ and $b$; this is proportional to the number of children of $a$, so summing this up over all pairs results in a time proportional to the number of nodes of $T_s$, so $O(n)$ time. The running time of both the first step and each of the iterations of the iterated step is linear in the sum of the number of positions $i$ of $s$ and $t$ for which we set Level$_s[i]$ (respectively, Level$_t[i]$) to a value different from $\infty$ and the total number of elements contained in the three lists for the nodes of the lists $L_k$ and $H_k$. 

21
for all \( k \). Because for each position \( i \) we change the value of \( \text{Level}_s[i] \) (or \( \text{Level}_t[i] \)) exactly once from \( \infty \) to some \( k \), then the number of positions \( i \) of \( s \) and \( t \), for which we set \( \text{Level}_s[i] \) (respectively, \( \text{Level}_t[i] \)) to a value different from \( \infty \), is \( O(n + n') \). Then, for each pair \((a, b)\) of \( H_k \), the total number of interval-pairs in the three lists associated to \((a, b)\) is linear in the number of children of the node \( a \). Summing up for all explicit nodes \( a \) of \( T_k \) the total number of interval-pairs occurring in the lists associated to \((a, b)\) (where \( a \) and \( b \) are P-connected) is clearly linear in the number of nodes of \( T_k \) (as each child is counted once). Then, for each pair \((a, b)\) of \( L_k \), the total number of interval-pairs in the three lists is \( O(1) \). Thus, the total number of elements contained in the three lists for the nodes of the lists \( L_k \) and \( H_k \), for all \( k \), is \( O(n) \).

We can now conclude that the overall time needed to perform the first step and all iterated steps is \( O(n + n') \). Thus, the whole algorithm runs in linear time, so the statement of the theorem is correct. \( \square \)

As a consequence of Theorem 4 we can show our main result. 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.

**Proof.** It is not hard to see that 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 Theorem 4 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 \( T_t \).

To find a distinguishing word of minimum length we proceed as follows.

We first run the algorithm of Theorem 4 and store all its additional data structures, including those produced in the preprocessing phase. We also use an additional array \( Y \) with \(|\Sigma|\) elements, all initialised to 0.

We will use this algorithm to show the following general claim.

\[ \text{Claim.} \]

Let \( i \) be a position of \( s \) and \( j \) be a position of \( t \), and let \( k > 1 \). Assume \( s[i : n] \sim_k t[j : n'] \), but \( s[i : n] \sim_{k-1} t[j : n'] \), and that \( i \) is included in the \((k-1)\)-block \( a = [m_a : n_a] \) and \( j \) is included in the \((k-1)\)-block \( b = [m_b : n_b] \). According to Lemma 5, there exists a letter \( x \) such that \( s[\text{next}(i, x) + 1 : n] \sim_{k-1} t[\text{next}(j, x) + 1 : n'] \); moreover, a word of length \( k \) that distinguishes \( i \) and \( j \) starts with \( x \). We can find the letter \( x \) and its position in time \( O(|\text{alph}(a)| + |\text{alph}(b)| + (\text{next}(i, x) - i) + (\text{next}(j, x) - j)) \).

Note that \( x \) occurs always after both \( i \) and \( j \) in their respective words, or \( s[i : n] \sim_{k-1} t[j : n'] \) would hold.

**Proof of the Claim.**

Indeed, we can find this \( x \) and \( \text{next}(i, x) \) and \( \text{next}(j, x) \) as follows.

Assume first that \( i \) and \( j \) are included, respectively, in a pair of P-connected \( k \)-blocks \( a' = [m_{a'} : n_{a'}] \) and \( b' = [m_{b'} : n_{b'}] \). Then, in the iterated step of the algorithm from Theorem 4 we have identified a letter \( x \) such that \( s[\text{next}(a', x) + 1 : n] \sim_{k-1} t[\text{next}(b', x) + 1 : n'] \); this letter can be stored as a satellite information in the \( \text{Level}_s \) and \( \text{Level}_t \) arrays. It is easy to note that this letter \( x \) appears in at most one of the factors \( s[i : n_a] \) and \( t[j : n_b] \). Clearly, in \( O((\text{next}(i, x) - i) + (\text{next}(j, x) - j)) \) we can search letter by letter for the first occurrence of \( x \) after \( i \) and \( j \), respectively; that is, we compute \( \text{next}(i, x) \) and \( \text{next}(j, x) \), respectively.

If \( i \) and \( j \) are included, respectively, in a pair of \( k \)-blocks \( a' = [m_{a'} : n_{a'}], b' = [m_{b'} : n_{b'}] \) which are not P-connected, then there exists a letter \( x \) that occurs in \( a \) after \( i \) but does not occur in \( b \) after \( j \), or vice versa. This is found as follows: we produce the lists \( L_a \) and \( L_b \) of last letters of the \( k \)-blocks which partition the \((k - 1)\) blocks \( a \) and \( b \), respectively, and remove those that occur to the left of \( i \) and \( j \), respectively. In \( Y \), we set \( Y[y] \leftarrow 1 \) for all \( y \) that occur in \( L_a \).
Then, we set $Y[y] \leftarrow Y[y] - 1$ for all $y \in L_b$. If there is a letter $y$ of $L_a$ such that $Y[y] = 1$, then we choose the letter $x$ which we were searching as $y$. Otherwise, there must be a letter $y$ of $L_b$ such that $Y[y] = -1$, and we choose the letter $x$ which we were searching as $y$. Furthermore, we reset $Y[y] = 0$ for all $y \in L_a \cup L_b$. Finding the letter $x$ as above takes $O(|\text{alph}(a)| + |\text{alph}(b)|)$ time. Furthermore, finding $\text{next}(i, x)$ and $\text{next}(j, x)$ takes $O((\text{next}(i, x) - i) + (\text{next}(j, x) - j))$ time.

Our claim follows. \hfill \Box

Now, we show the main statement. Running Theorem 4 we compute a value $k$ such that $s[1 : n] \approx_k t[1 : n']$, but $s[1 : n] \sim_{k-1} t[1 : n']$. Let us assume that $k > 1$. Using our claim, we find a letter $x_1$ such that $s[\text{next}(1, x_1) + 1 : n] \approx_{k-1} t[\text{next}(1, x_1) + 1 : n']$. Then, $s$ and $t$ will be distinguished by a word $w = x_1 w'$, with $|w'| = k - 1$. This takes $O(|\text{alph}(a_{k-1})| + |\text{alph}(b_{k-1})|) + (\text{next}(1, x_1) - 1) + (\text{next}(1, x_1) - 1))$ time, where $a_{k-1}$ is the $(k-1)$-block of $s$ in which $1$ is included, and $b_{k-1}$ is the $(k-1)$-block of $t$ in which $1$ is included.

We continue by searching the first letter $x_2$ of $w'$ as the first letter of a word distinguishing $s[\text{next}(1, x_1) + 1 : n]$ and $t[\text{next}(1, x_1) + 1 : n']$. Because we have $s[1 : n] \approx_k t[1 : n']$, but $s[1 : n] \sim_{k-1} t[1 : n']$, it follows that $s[\text{next}(1, x_1) + 1 : n] \sim_{k-1} t[\text{next}(1, x_1) + 1 : n']$ but $s[\text{next}(1, x_1) + 1 : n] \sim_{k-2} t[\text{next}(1, x_1) + 1 : n']$ (otherwise, $s[1 : n]$ would be distinguished by a word of length $k - 1$ from $t[1 : n']$).

This means we can apply the claim for $i = \text{next}(1, x_1) + 1$ (included in the $k - 2$ block $a_{k-2}$ of $s$) and $j = \text{next}(1, x_1) + 1$ (included in the $k - 2$ block $b_{k-2}$ of $t$) and $k - 1$ instead of $k$. We repeat this until we reach two positions $i$ and $j$, and we need to find a single letter $x_k$ that distinguishes $s[i : n]$ and $t[j : n']$ (that is, we reached $k = 1$, and we need to find the last letter of the word $w$ which distinguishes $s[1 : n]$ and $t[1 : n']$). This can be easily found in $O(n)$ by applying, e.g., a similar strategy as the one in the proof of the claim.

The overall time complexity is $O(n)$ as the time we use is:

\[
O(n) + O(|\text{alph}(a_{k-1})| + |\text{alph}(b_{k-1})|) + (\text{next}(1, x_1) - 1) + (\text{next}(1, x_1) - 1)) + O(|\text{alph}(a_{k-2})| + |\text{alph}(b_{k-2})|) + (\text{next}(\text{next}(1, x_1) + 1, x_2) - \text{next}(1, x_1)) + (\text{next}(\text{next}(1, x_1) + 1, x_2) - \text{next}(1, x_1)) + \ldots
\]

Adding this up, we get that the overall time we use is $O(|T_s| + |T_t| + n) = O(n)$. \hfill \Box

Finally, one can show the following extension of Theorem 4. Given $\ell$ words $s_1, \ldots, s_\ell$, we can compute in $O(\sum_{1 \leq i \leq \ell} |s_i|)$ time the largest $k$ such that for all $i, j \leq \ell$ we have $s_i \sim_k s_j$.

Proof. We can run the algorithm in Theorem 4 to solve MaxSimK for the pairs of input words $s_i$ and $s_{i+1}$ for $i$ from 1 to $\ell - 1$. Then we take the minimum from the values we computed in the previous step. This clearly works, as $\sim_k$ is a congruence relation. \hfill \Box

5 Conclusions and future work

In this paper, we presented an algorithm solving MaxSimK in optimal time, based on the construction of the Simon-trees associated to the two input words, and then on establishing a connection between their nodes. One way our solution can be further improved is to see if the rather heavy data structures toolbox we use to show Theorem 4 can be simplified without
losing complexity, and to reach a more elegant and efficient algorithm similar to, e.g., the one constructing the Simon-tree of a word. Other problems one could consider are variants of MaxSimK. For instance, how to compute, for two words \( s \) and \( t \), what is the largest \( k \) such that \( \mathcal{SF}_{\leq k}(s) \subseteq \mathcal{SF}_{\leq k}(t) \).

An interesting direction in which we can extend our results is to consider more general input alphabets. In particular, a computational model used in string algorithms assumes that the input is over general ordered alphabets (see [12, 13, 20] and the references therein). More precisely, the input is a sequence of elements from a totally ordered set \( \mathcal{U} \) (i.e., string over \( \mathcal{U} \)). The operations allowed in this model are those of the standard Word RAM model, with one important restriction: the elements of the input cannot be directly accessed nor stored in the memory used by the algorithms; instead, we are only allowed to compare (w.r.t. the order in \( \mathcal{U} \)) any two elements of the input, and the answer to such a comparison-query is retrieved in \( O(1) \) time. In this model, it still holds that sorting the elements of an input sequence as well as checking the equality of two sets of size \( O(n) \) require at least \( \Omega(n \log n) \) comparisons. So, following the arguments presented in the Introduction, solving MaxSimK would take at least \( \Omega(n \log n) \) time. An implementation of our algorithms, where we first sort the letters of the input words, map them to words over \( \{1, \ldots, n + n'\} \), and then use the same strategy as the one described for the case of integer alphabets, runs in \( O(n \log n) \) time. So, from the matching upper and lower bounds, we get also in this case an optimal algorithm for solving MaxSimK. This complexity is still achieved when we use a less-complicated lighter-weight implementation of the union-find and split-find data structures, e.g., the one in [33].

Another interesting extension of Theorem 4 would be the efficient construction of succinct data structures allowing us to retrieve in sublinear time the answer to MaxSimK for any two words belonging to a given set.
References

[1] Laura Barker, Pamela Fleischmann, Katharina Harwardt, Florin Manea, and Dirk Nowotka. Scattered factor-universality of words. To appear in Proc. DLT 2020, CoRR, abs/2003.04629, 2020. arXiv:2003.04629

[2] M. A. Bender and M. Farach-Colton. The LCA problem revisited. LATIN 2000: Theoretical Informatics, 1776:88–94, 2000.

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

[4] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. Algorithms on strings. Cambridge University Press, 2007.

[5] 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.

[6] David P. Dobkin and Richard J. Lipton. On the complexity of computations under varying sets of primitives. J. Comput. Syst. Sci., 18(1):86–91, 1979.

[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. CoRR, abs/1509.00622, 2015. 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] Pawel Gawrychowski, Tomasz Kociumaka, Wojciech Rytter, and Tomasz Walen. Faster longest common extension queries in strings over general alphabets. In Roberto Grossi and Moshe Lewenstein, editors, 27th CPM 2016., volume 54 of LIPIcs, pages 5:1–5:13, 2016.

[13] 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.

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

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

[16] 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.
[17] 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.

[18] 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.

[19] Dmitry Kosolobov. Computing runs on a general alphabet. Inf. Process. Lett., 116(3):241–244, 2016.

[20] Dmitry Kosolobov. Finding the leftmost critical factorization on unordered alphabet. Theor. Comput. Sci., 636:56–65, 2016.

[21] Dietrich Kuske and Georg Zetzsche. Languages ordered by the subword order. In Proc. FOS-SACS 2019, volume 11425 of Lecture Notes in Computer Science, pages 348–364. Springer, 2019.

[22] Marie Lejeune, Julien Leroy, and Michel Rigo. Computing the k-binomial complexity of the Thue-Morse word. CoRR, abs/1812.07330, 2018. arXiv:1812.07330.

[23] Julien Leroy, Michel Rigo, and Manon Stipulanti. Generalized Pascal triangle for binomial coefficients of words. CoRR, abs/1705.08270, 2017. arXiv:1705.08270.

[24] M. Lothaire. Combinatorics on Words. Cambridge University Press, 1997.

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

[26] Alexandru Mateescu, Arto Salomaa, and Sheng Yu. Subword histories and Parikh matrices. Journal of Computer and System Sciences, 68(1):1–21, 2004.

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

[28] Arto Salomaa. Connections between subwords and certain matrix mappings. Theoretical Computer Science, 340(2):188–203, 2005.

[29] Shinnosuke Seki. Absoluteness of subword inequality is undecidable. Theor. Comput. Sci., 418:116–120, 2012.

[30] Imre Simon. An algorithm to distinguish words efficiently by their subwords. unpublished.

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

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

[33] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. J. ACM, 22(2):215?225, 1975.

[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.