A Fast Algorithm for the Inexact Characteristic String Problem

Moritz G. Maaß*
Fakultät für Informatik, TU München
Boltzmannstr. 3, D-85748 Garching, Germany
maass@informatik.tu-muenchen.de

August 4, 2003

Abstract

We present a new algorithm to solve the INEXACT CHARACTERISTIC STRING PROBLEM using Hamming distance instead of Levenshtein distance as a measure. We embed our new algorithm and the previously known algorithm for Levenshtein distance in a common framework which reveals an additional improvement to the Levenshtein distance algorithm. The INEXACT CHARACTERISTIC STRING PROBLEM can thus be solved in time $\mathcal{O}(||T|| + l \cdot ||S \setminus T||)$ for Hamming distance and in time $\mathcal{O}(||T|| + k \cdot l \cdot ||S \setminus T||)$ for Levenshtein distance, where $S \subseteq \Sigma^*$, $T \subsetneq S$ ($T \neq \emptyset$) is the target set, and l is the length of a shortest string in T. The INEXACT CHARACTERISTIC STRING PROBLEM has applications in probe and primer design.

Both algorithms need to solve the COMMON SUBSTRING PROBLEM for more than two strings. We present an improved algorithm for this problem being simpler and faster in practice by a constant factor than the previous algorithm.

Keywords: Algorithms and Data Structures, Pattern Matching, Computational Biology

1 Introduction

Solving the CHARACTERISTIC STRING PROBLEM requires to find a string that matches all strings of a selected subset (the target set) and that does not match any string in the remainder of the given set of strings (the distance set). The solution is then characteristic for the strings in the target set. Given an arbitrary string out of the set we can decide whether it belongs to the target set by checking whether the characteristic string matches it.

The problem is motivated by applications in computational biology [6]. In DNA sequencing, the techniques described here can be used to select a primer that bonds somewhere in a defined region of a DNA strand, while excluding the possibility that the string hybridizes in another region. PCR can then be used to replicate the substrings of the DNA starting at the position where the primer hybridized. This is used in a technique called "chromosome walking" to close gaps in DNA sequencing.

*Research supported by DFG, grant Ma 870/5-1 (Leibnizpreis Ernst W. Mayr).

Another usage is the design of probes. A probe can be used to verify the presence (or, better, absence) of a certain genome in a defined environment. A (short) complementary subsequence (the probe) of the target DNA is synthesized that does not match any other DNA occurring in the environment (e.g., the target might be a harmful one among a set of harmless bacteria). If the probe does not hybridize, the target does not appear. If the probe hybridizes, the target might or might not appear. Therefore, the probe needs to be carefully selected so that it does not hybridize to other DNA occurring frequently in the defined environment. In both problems, the primer or probe is selected to match some target DNA while not matching other DNA from the environment.

The hybridization process does not require an exact match. A probe might hybridize even if some base pairs do no match. This motivates the definition of the INEXACT CHARACTERISTIC STRING PROBLEM. To further reduce false hybridization, the probe is selected such that it does not match non-target DNA (which must be known a priori) even allowing some errors.

There are multiple notions of matching with errors, most prominently Hamming distance and LEVENSHTEIN DISTANCE[1]. Ito et al. [9] have presented an algorithm to solve the INEXACT CHARACTERISTIC STRING PROBLEM in time $\mathcal{O}(||T|| + l^2 \cdot |S \setminus T| + k \cdot l \cdot ||S \setminus T||)$, where $S \subseteq \Sigma^*$, $T \subsetneq S$ ($T \neq \emptyset$) is the target set, $S \setminus T$ the distance set, and l is the length of a shortest string in T. Here $|S|$ is the cardinality of the set S and $||S||$ is the size of all elements of S. Levenshtein distance is used as a measure, but the algorithm is easily adapted to solve the problem for Hamming distance in the same asymptotic time bound.

In this paper we present a new algorithm for efficiently solving the INEXACT CHARACTERISTIC STRING PROBLEM for Hamming distance. The new algorithm is faster and more space efficient than the above algorithm – it runs in time $\mathcal{O}(||T|| + l \cdot ||S \setminus T||)$ – but it works only for Hamming distance and not for Levenshtein distance. We believe that this is not a severe restriction because Hamming distance seems to be a very natural measure with respect to DNA hybridization. Lanctot et al. [10] argue that gaps are much more destabilizing than substitutions, thus making Hamming distance the measure of choice for the design of short oligomers (e.g., probes).

We take a second look at the Levenshtein distance version of the INEXACT CHARACTERISTIC STRING PROBLEM, which can be be solved more efficiently in time $\mathcal{O}(||T|| + k \cdot l \cdot ||S \setminus T||)$ when embedded in the same framework.

Furthermore we present a practical and very efficient algorithm for the COMMON SUBSTRING PROBLEM which improves over previous algorithms [8] by a constant factor in speed and a much smaller memory overhead.

2 Overview and Previous Work

For our algorithm we introduce a four step framework which structures the solution into different parts. The two major parts of the algorithm consist of computing substrings that do not match a string from the distance set ("difference part"), and of solving the COMMON SUBSTRING PROBLEM for the strings in the target set. Both parts use a shortest string $v \in T$ of length l as a reference.

The same framework can be used for Levenshtein distance as distance measure and we show how to reduce the running time by carefully embedding the core of the algorithm of Ito et al. [9] in our framework. The computation of the substrings not matching the non-target strings (we will

[1] also known as edit distance

call them k-distant substrings) contributes the main term in the running time. Thus, the running time depends solely on the choice between Levenshtein distance or Hamming distance in the difference part. As a result, the Hamming distance version of the INEXACT CHARACTERISTIC STRING PROBLEM can be solved by a factor of k faster than the improved Levenshtein distance version. Experimental results indicate an additional performance yield for the Levenshtein distance version even for $k = 1$ because no suffix trees and no constant lowest common ancestor queries are needed. If k is not independent but related to an error rate α, then there is a linear dependency between k and the length of the characteristic string. This length can be bounded from below by the size of a 0-characteristic string. Szpankowski [17] has shown that the size of a minimal string that does not occur as a substring in a string of size n converges almost sure to $\frac{1}{h}\log n$ as $n \longrightarrow \infty$, where $0 < h < \infty$ is related to the entropy[2]. So for a fixed error rate α, one should choose $k = \Omega(\log ||S \setminus T||)$.

To avoid confusion, we will say "v is a substring of S", if v is a substring of each string in S. We will say that "v is a substring in S", if there is a string in S that has v as a substring.

A linear algorithm to solve the CHARACTERISTIC STRING PROBLEM has been published by Nakanishi et al. [15]. The algorithm relies on generalized suffix trees and runs in time and space $\mathcal{O}(||S||)$. The generalized suffix tree for all strings in S is first traversed to mark all nodes which represent a prefix of a string v in $S \setminus T$ (they have a leaf representing a suffix of v as a child). In a second run for each node n that is unmarked and a child of a marked node p, it is checked (by a linear time subtree traversal) whether the node represents a substring u of T. If a leaf representing a suffix of a string v is found below n for every $v \in T$, then u is a substring of T. Since u is not a substring in $S \setminus T$ it is a characteristic string. Every prefix of u that is longer than the string represented by p is a characteristic string. Among all strings thus found, the shortest one is returned.

Ito et al. [9] have presented an algorithm to solve the INEXACT CHARACTERISTIC STRING PROBLEM in time $\mathcal{O}(||T|| + l^2 \cdot |S \setminus T| + k \cdot l \cdot ||S \setminus T||)$, where l is again the length of a shortest string in T. The key idea here is to use the diagonal method of dynamic programming to match every suffix of a shortest string v in T against all strings of $S \setminus T$ with k errors. (This method was introduced by Landau and Vishkin [11] and, at the same time, Myers [14].) If for a suffix of v the end of the string is not reached in any such match an appropriately sized prefix of the suffix is a k-distant v-substring. Every such substring is matched against the remaining strings in T. If every string has a match, the substring is a k-characteristic substring. Among all these the shortest one is selected. The algorithm can easily be improved to run in time $\mathcal{O}(||T|| + k \cdot l \cdot ||S \setminus T||)$.

Our solution to the INEXACT CHARACTERISTIC STRING PROBLEM for Hamming distance makes use of the fact that overlapping substrings share common matches and mismatches. For instance, consider the strings `GTTAGGATTA` and `GTTAGATTA`. The substring `TTAGG` matches the substring `TTAGA` with one error. The overlapping substrings `AGGATT` and `AGATTA` match with three errors (`AGG` and `AGA` match with one error and the additional substrings `ATT` and `TTA` match with two errors). For Levenshtein distance we can match the substrings `AGGATT` and `AGATTA` with distance two, which does not depend on the previous errors made when matching `TTAGG` and `TTAGA` with one error.

For each string u from the distance set $S \setminus T$ we find for every suffix $v[i]$ of the shortest string v from T the length of a longest substring of u matching a prefix of $v[i]$. For each starting position of v, we thus calculate the length of a longest match with distance k. Instead of calculating the lengths for each position in v at once, we calculate the maximal matching lengths for each of the $|u| + |v|$ different alignments of v against u, using information from overlapping substrings.

[2]For a memoryless Source, $h = -\log(p_{min})$.

The difference part of the algorithm can be implemented very efficiently without the use of suffix trees or other complex data structures. Only for the solution of the COMMON SUBSTRING PROBLEM we do need suffix trees.

We also present a much simpler and (by a constant factor) faster algorithm to solve the COMMON SUBSTRING PROBLEM for more than two strings. Lucas Hui has given a linear time solution to this problem that relies on constant time lowest common ancestor computation [8] (see also [6]). The algorithm builds the generalized suffix tree for the given set of strings and augments the tree with information to calculate lowest common ancestor queries (see for instance [16]) in constant time. The resulting data structure is linear in space but with a large constant.

Our algorithm constructs the suffix tree for each string in the set one by one, at any time keeping only one suffix tree in memory and without the overhead for lowest common ancestor queries. Experimental results (for an alphabet of size four and from two to one hundred strings of sizes up to a hundred thousand characters) indicate a better performance of the new algorithm by a factor of four to five. An even more space efficient version of the algorithm constructs only the suffix tree for the shortest string in the set. In the computational biology domain the much lower amount of space needed might enable researchers to tackle previously unreachable problems.

A problem related to the INEXACT CHARACTERISTIC STRING PROBLEM is the INVERSE PATTERN MATCHING PROBLEM. To solve the INVERSE PATTERN MATCHING PROBLEM for a given string a short pattern is searched that maximizes (minimizes) the sum of all character mismatches when the pattern is aligned at all positions of the given string (which is the sum of the Hamming distance between each corresponding substring and the pattern). There are variants of the problem depending on whether a pattern is sought to come from the string itself (internal inverse pattern matching) or not (external inverse pattern matching). Amir et al. [1] have studied the problem and have given first algorithms to solve the INVERSE PATTERN MATCHING PROBLEM and its variants. Later, Gasieniec et al. [5] have improved the solution to the external INVERSE PATTERN MATCHING PROBLEM. The internal INVERSE PATTERN MATCHING PROBLEM can be solved in time $\mathcal{O}(n\sqrt{m}\log^2 m)$, where m is the size of the desired pattern. The INVERSE PATTERN MATCHING PROBLEM and its external variant can be solved in time $\mathcal{O}(n)$ for fixed alphabet size.

The main difference between the INVERSE PATTERN MATCHING PROBLEM and the INEXACT CHARACTERISTIC STRING PROBLEM lies in the fact that the solution string for the INEXACT CHARACTERISTIC STRING PROBLEM is guaranteed not to match anywhere even with k errors, while the solution string for the INVERSE PATTERN MATCHING PROBLEM might have the highest amount of mismatches but still match at a lot of places with few (or no for the non-external versions) errors. The solution to the INVERSE PATTERN MATCHING PROBLEM is a string that has a maximal average distance, while the solution to the INEXACT CHARACTERISTIC STRING PROBLEM guarantees a minimal distance for each position. Hence, the INVERSE PATTERN MATCHING PROBLEM solution pattern is not well suited for sharp classifications. The applications of the INVERSE PATTERN MATCHING PROBLEM are such that sharp distinctions are not necessary (see [1] for some examples).

On the other hand, the INVERSE PATTERN MATCHING PROBLEM does not require a target set. When no target set is given for the INEXACT CHARACTERISTIC STRING PROBLEM, the given algorithms do not work. A straightforward solution supplying an adequate de Bruijn sequence of order m (with length $|\Sigma|^m$) would lead to a running time of $\mathcal{O}(|\Sigma|^m \cdot ||S||)$ for Hamming distance. It is unlikely that an algorithm with a polynomial running time exists. When there is no target set the problem is, given a set $S \subseteq \Sigma^*$ and an integer k, to find a string $v \in \Sigma^*$ such no substring in S is within Hamming distance k of v. Frances and Litman [4] have shown that, given a set $C \subseteq \{0,1\}^n$ and an integer k, the MAXIMUM COVERING RADIUS PROBLEM of deciding wether there exists a string $v \in \Sigma^n$ with Hamming distance greater k to every element of C is NP-complete. Lanctot et al. [10] show that several related problems such as the DISTINGUISHING STRING SELECTION PROBLEM and the FARTHEST STRING PROBLEM are also NP-complete. For the FARTHEST STRING PROBLEM a string $v \in \Sigma^n$ with Hamming distance at least k to all strings of a set $S \subseteq \Sigma^n$ is sought, a generalization of MAXIMUM COVERING RADIUS PROBLEM. Lanctot et al. also present a PTAS for that problem. The input of the DISTINGUISHING STRING SELECTION PROBLEM is a set $B \subseteq \Sigma^n \times \Sigma^*$, a set $G \subseteq \Sigma^n$ and two thresholds d_b, d_g. The objective is to find a string v with Hamming distance at most d_b to each substring of length n in B and Hamming distance at least d_g to each string in G.

3 Preliminaries

Let Σ be an arbitrary finite alphabet, let Σ^* denote the set of all finite strings over Σ (including the empty string ϵ), let $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$ denote the set of all non-empty strings over Σ. Let $t = t_1 t_2 t_3 \ldots t_n$ be a string with characters $t_i \in \Sigma$, we define $|t| = n$ to be its length. By $t[i]$ we will refer to the i-th suffix of t: $t[i] = t_i \ldots t_{|t|}$. Given a (finite) set of strings $S \subseteq \Sigma^*$, we denote by $|S|$ the number of elements of S and by $||S||$ the size of S ($||S|| = \sum_{u \in S} |u|$).

If c is a condition, then we will let the expression $[c]$ denote 1 if c is true and 0 otherwise ($[\cdot]$ is sometimes called the indicator function). For instance, $[a = b]$ is 1 if a and b represent the same characters and it is 0 otherwise.

Definition 1 (CHARACTERISTIC STRING PROBLEM). *Given a set S of strings and a non-empty set $T \subsetneq S$, find a shortest string u, s.t. u is a substring of T and u is not a substring in $S \setminus T$.*

The idea behind this definition is that the string u characterizes the set T in S.

In inexact pattern matching there are two classical measures for an approximate match, these are Levenshtein distance and Hamming distance. They are defined as follows:

Definition 2 (Levenshtein distance). *Let $u, v \in \Sigma^*$ be two arbitrary strings. The Levenshtein distance $\mathrm{dist}_{lev}(u,v)$ is defined as the minimal number of insertions, deletions and replacements of characters needed to transfer u into v. Ignoring the boundry cases, the Levenshtein distance of $u = u_1 \ldots u_n$ and $v = v_1 \ldots v_m$ can be defined recursively as*

$$\mathrm{dist}_{lev}(u_1 \ldots u_n, v_1 \ldots v_m) = \min \left\{ \begin{array}{l} \mathrm{dist}_{lev}(u_2 \ldots u_n, v_2 \ldots v_m) + [u_1 \neq v_1], \\ \mathrm{dist}_{lev}(u_1 \ldots u_n, v_2 \ldots v_m) + 1, \\ \mathrm{dist}_{lev}(u_2 \ldots u_n, v_1 \ldots v_m) + 1 \end{array} \right\},$$

where the first line represents the match/replacement case, the second line represents the deletion case, and the third line represents the insertion case.

The Hamming distance is only defined for strings of the same length.

Definition 3 (Hamming distance). *For $n > 0$, let $u, v \in \Sigma^n$ be two strings. The Hamming distance is defined as the number of characters that do not match between u and v:*

$$\mathrm{dist}_{ham}(u,v) = \sum_{1 \leq i \leq n} [u_i \neq v_i]$$

With the use of a distance measure $\operatorname{dist}(u, v)$ we can define the inexact version of the CHARACTERISTIC STRING PROBLEM:

Definition 4 (INEXACT CHARACTERISTIC STRING PROBLEM). *Given a set S of strings, a non-empty set $T \subsetneq S$, and a number k, find a shortest string u, s.t. u is a substring of T and u matches no substring in $S \setminus T$ with k or less errors. Formally, for all $w \in S \setminus T$, if w' is a substring of w then $\operatorname{dist}(w', u) > k$.*

For convenience, we introduce some further notions. Let $u \in \Sigma^*$ be such that all substrings w' of all strings $w \in S \setminus T$ have $\operatorname{dist}(w', u) > k$. Then u is called a "**k-distant string**" (with respect to $S \setminus T$). If u is a substring of v and u is a k-distant string with respect to $S \setminus T$ we will call u a "**k-distant v-substring**". We call a string with property P "**right minimal**", if $u_1 \dots u_{|u|-1}$ does not have property P. Analogously, a substring $u_i \dots u_j$ with property P is called "**right maximal**", if $u_i \dots u_{j+1}$ does not have property P.

4 A Faster Solution for the Hamming distance INEXACT CHARACTERISTIC STRING PROBLEM

We will present an $\mathcal{O}(||T|| + l \cdot ||S \setminus T||)$ algorithm, where l is the length of some shortest string $v \in T$. The input of the algorithm consists of a set of strings $S \subseteq \Sigma^*$ and a target set $T \subsetneq S$.

The algorithm consists of four phases:

1. Find a shortest string v in T.
2. For i from 1 to $|v|$ find the right minimal k-distant v-substring $cand_i$ (if it exists). $cand_i$ occurs at position i in v.
3. For each candidate $cand_i$ check whether it is a common substring of all strings in T and hence a k-characteristic string.
4. Select the shortest among all k-characteristic strings.

Observation 5. *Let $v \in T$ be a shortest string in T. Each characteristic string is a substring of v.*

We will therefore only need to consider the shortest string $v \in T$ when comparing it with strings from $S \setminus T$.

Lemma 6. *Every shortest k-characteristic string is a right minimal k-distant v-substring with respect to $S \setminus T$.*

Proof. Let s be a shortest k-characteristic string. Obviously, s must be a substring of T and hence also of v. By definition, s must be a k-distant string with respect to $S \setminus T$. Suppose, s were not right minimal, then $s' = s_1 \dots s_{|s|-1}$ would be a k-distant string. Since s is a common substring of T, so would be s'. Hence, s' would be a shorter k-characteristic string, contradicting the assumption. □

As a consequence, by looking at all right minimal k-distant v-substrings with respect to $S \setminus T$ we will also see every shortest k-characteristic string. Hence, the above scheme is correct. A key idea to the algorithm is the following observation:

Lemma 7. *$cand_i = v_i \dots v_{i+l+1}$ is a right minimal k-distant v-substring, iff $cand'_i = v_i \dots v_{i+l}$ is a right maximal substring of v, such that for any substring u in $S \setminus T$ at least $\operatorname{dist}(cand'_i, u) \geq k$ and there is a substring u in $S \setminus T$ with $\operatorname{dist}(cand'_i, u) = k$.*

Proof. $v_i \dots v_{i+l+1}$ must have distance at least $k+1$ from any substring in $S \setminus T$, otherwise $v_i \dots v_{i+l}$ would not be right maximal. Therefore, $v_i \dots v_{i+l+1}$ is a k-distant string. On the other hand, there is a substring in $S \setminus T$ such that $v_i \dots v_{i+l}$ matches with distance k. Hence, $v_i \dots v_{i+l+1}$ is right minimal. □

With Lemma 7 we can find the candidates by finding maximal length matches.

Phase 1 can easily be implemented to run in time $\mathcal{O}(||T||)$. Let $l = |v|$ be the length of the shortest string in T. By Lemma 6, there are at most l candidates after Phase 2 and Phase 3. Hence, Phase 4 can easily be implemented in time $\mathcal{O}(l)$.

In the sequel we will show how to implement Phase 2 in time $\mathcal{O}(l \cdot ||S \setminus T||)$ and Phase 3 in time $\mathcal{O}(||T||)$. This will lead to a total running time of $\mathcal{O}(||T|| + l \cdot ||S \setminus T||)$.

4.1 A Simple Algorithm for Finding All Common Substrings of a Set of Strings in Linear Time

Definition 8 (COMMON SUBSTRING PROBLEM). *Let T be a set of strings. Find all strings v, s.t. v is a substring of all strings in T.*

Since there may be $\Omega(||T||^2)$ many common substrings, one has to be careful about the output representation. The output can be represented in linear space[3], if we pick a string $v \in T$ and store the length of the longest common substring starting at position $i \in \{1, \dots, |v|\}$ in v.

Lucas Hui has given a linear solution to this problem that relies on constant time lowest common ancestor computation [8] (see also [6]). Hui's algorithm is able to find at the same time all substrings common to a fixed number of $k \leq |T|$ strings in T, while we will only find substrings common to all strings in T. Our algorithm is much simpler and faster.

The algorithm is based on matching statistics (see [6], respectively [2]). The matching statistic $ms(i)$ for a pattern p and a text t gives the length of the longest substring starting at position i in p that is also a substring of t. All values of $ms(i)$, $i \in \{1, \dots, |p|\}$, can be calculated in time $\mathcal{O}(|p| + |t|)$ with the use of a suffix tree for t. The linear time algorithm constructs the suffix tree for t (with suffix links) in time $\mathcal{O}(|t|)$. Then it finds a canonical reference pair (see the construction of suffix trees by Ukkonen [18] for a detailed description of reference pairs, suffix links, canonizing etc.) for the longest prefix w of p that matches a substring of t by a simple search in the suffix tree. A canonical reference pair for string w is a node n representing a prefix of w in the suffix tree and a length denoting a prefix of an edge starting at n representing the remaining part of w. The length of the prefix w of p is the value of $ms(1)$. The reference pair and the length for $ms(i+1)$ is computed from the reference pair for $ms(i)$ by "shortening" the reference pair at the front. This is done by replacing the node n with its suffix link parent and afterwards canonizing and lengthening the reference pair as far as possible. The suffix links are a byproduct of the suffix tree construction (see [13, 18]). For a node n representing a string $u = u_1 u_2 \dots u_m$, its suffix link leads to a node n' representing the first suffix $u_2 \dots u_m$ of u. The whole traversal takes amortized time $\mathcal{O}(|p|)$.

To find all common substrings of T in time $\mathcal{O}(||T||)$ we proceed as shown in Algorithm 1.

Theorem 9. *Algorithm 1 correctly solves the* COMMON SUBSTRING PROBLEM.

Proof. Let w be a common substring of all strings in T. Then w is also a substring of the shortest string v in T. W.l.o.g. let w appear at position i in v ($w = v_i \dots v_{i+|w|}$). Since w is common to all

[3] Only in the uniform cost model.

Algorithm 1 Common Substring

1: Let v be the shortest string in T.
2: **for** $i = 1$ **to** $|v|$ **do**
3: $\quad ms_{min}(i) = |v| - i + 1$
4: **for all** $u \in T, u \neq v$ **do**
5: $\quad$ calculate matching statistics $ms(i)$ (i from 1 to $|v|$) for v and u.
6: $\quad$ **for** $i = 1$ **to** $|v|$ **do**
7: $\quad\quad ms_{min}(i) = \min\{ms_{min}(i), ms(i)\}$

strings $u \in T$, in every iteration of the loop in line 4 $ms(i) \geq |w|$ and therefore $ms_{min}(i) \geq |w|$. This is especially true for the longest common substring starting at position i.

Suppose, after Algorithm 1 $ms_{min}(i) = j$. For every iteration of the loop in line 4 we have $ms(i) \geq j$ and there is a substring of length j or greater in starting at i in v that is also a substring of the current string u. Therefore $v_i \dots v_{i+j}$ is a common substring of all strings in T. □

Theorem 10. *Algorithm 1 runs in time $\mathcal{O}(||T||)$*

Proof. To find the shortest string v in line 1 of the algorithm it takes time $\mathcal{O}(||T||)$. The for-loop of lines 2–3 takes time $\mathcal{O}(|v|)$. Lines 5–7 are is executed for each string u in $T \setminus \{v\}$. Line 5 takes time $\mathcal{O}(|u| + |v|) = \mathcal{O}(|u|)$ (because $|u| \geq |v|$) and lines 6 – 7 take time $\mathcal{O}(|v|) = \mathcal{O}(|u|)$. Hence, Lines 4–7 take total time $\sum_{u \in T \setminus \{v\}} \mathcal{O}(|u|) = \mathcal{O}(||T \setminus \{v\}||)$. As a result the total running time is $\mathcal{O}(||T \setminus \{v\}|| + |v| + ||T||) = \mathcal{O}(||T||)$. □

An even more space efficient implementation uses only one suffix tree for the shortest string v to calculate the matching statistics for v and $u \in T \setminus \{v\}$. This is achieved by storing a "high water mark" for each edge as the deepest position reached by a traversal with a reference pair as described above with string u. A first subsequent traversal sets the "high water mark" to maximum at all ancestors of marked edges. In a final traversal for each leaf the maximal mark of any parent edge is stored as the corresponding value of the matching statistics (for the leaf representing the i-th suffix, the value is stored as $ms(i)$). The traversal for string u costs $\mathcal{O}(|u|)$, while the two subsequent traversals (and the building of the suffix tree) cost $\mathcal{O}(|v|)$. Hence, the total time is also $\mathcal{O}(|u| + |v|)$. If v is smaller than the other strings in the set, this approach will save even more space. Additional space to store the high water marks is needed, so the algorithm needs $2n$ more integers for the suffix tree.

Compared with Hui's algorithm either version of this algorithm saves a considerable amount of space. For a large number of huge strings as they are encountered in DNA databases, space requirements make computations possible with our new algorithm which would have used a prohibitive amount of memory with the previous algorithm.

4.2 Finding All Right Minimal k-Distant v-Substrings with Respect to $S \setminus T$

This section describes how Phase 2 of the algorithm works for Hamming distance as distance measure. In Phase 2, for each position i of the shortest string $v \in T$, the right minimal k-distant v-substrings $cand_i$ is searched for. In order to find $cand_i$, we must align the i-th suffix $v[i]$ against every suffix $u[j]$ of a string in $u \in S \setminus T$ and calculate the shortest prefixes of $u[j]$ and $v[i]$ that have a distance greater then k. We use Lemma 7 to calculate the right minimal k-distant v-substrings from the right maximal prefixes with distance k.

Algorithm 2 Minimal Prefixes with Distance k

Let $m[]$ be an array of size $|v|$, initialized to 0
Let $e[]$ be an array of size $k+1$, organized as a ring buffer, initialized to 0
for $(pos_v, pos_u) \in \{(1,l)|l = 1 \ldots |u|\} \cup \{(l,1)|l = 1 \ldots |v|\}$ **do**
  $len_{match} = 0$
  $h = 0$
  $pos_{err} = 0$
  while $pos_{err} \leq k$ and $pos_v + h \leq |v|$ and $pos_u + h \leq |u|$ **do**
    $l = \mathrm{lce}(pos_v + h, pos_u + h)$
    $e[pos_{err}] = l$
    $len_{match} = len_{match} + l + 1$
    $h = h + l + 1$
    $pos_{err} = pos_{err} + 1$
  while $pos_v + h - 1 \leq |v|$ and $pos_u + h - 1 \leq |u|$ **do**
    for $r = 0$ **to** $e[pos_{err} - k - 1]$ **do**
      $m[pos_v + h - len_{match} + r] = \max\{m[pos_v + h - len_{match} + r], len_{match} - r\}$
    $len_{match} = len_{match} - e[pos_{err} - k - 1] - 1$
    $l = \mathrm{lce}(pos_v + h, pos_u + h)$
    $e[pos_{err}] = l$
    $len_{match} = len_{match} + l + 1$
    $h = h + l + 1$
    $pos_{err} = pos_{err} + 1$
  if $pos_v + h - 1 = |v| + 1$ **then**
    for $h' = |v| + 2 - len_{match}$ **to** $|v|$ **do**
      $m[h'] = |v| + 1$
  else if $pos_u + h - 1 = |u| + 1$ **then**
    for $l' = 0$ **to** len_{match} **do**
      if $m[pos_v + h - l'] < l'$ **then**
        $m[pos_v + h - l'] = l'$

The *longest common extension* (lce) of two strings u and v is defined as the longest prefix of u that is also a prefix of v (or vice versa). Using a generalized suffix tree for v and u we can calculate the lce between any two suffixes in constant time by means of constant time *lowest common ancestor* (lca) queries (for lca see [16]). The lce of two suffixes $v[i]$ and $u[j]$ is the string represented by the lca of the leaves (which represent these suffixes) in the generalized suffix tree. In particular we only need its length, which we can store with each node.

The maximal prefixes of $u[j]$ and $v[i]$ with distance k can thus be calculated in time $\mathcal{O}(k)$. But this simple approach would lead to an overall time $\mathcal{O}(k \cdot ||S \setminus T|| \cdot l + l \cdot |S \setminus T|)$. We will improve this basic scheme by reducing redundant comparisons and work in preparing lce queries.

Let u be an arbitrary string in $S \setminus T$. We define the alignment offset of the i-th suffix $v[i]$ of v and the j-th suffix $u[j]$ of u as $o_{align} = i - j$. In order to speed up the distance calculations we will calculate the lengths of all maximal prefixes with distance k of all suffixes with the same alignment offset together. This can be done by aligning the $(o_{align}+1)$-th character of v with the first character of u, respectively the $(-o_{align}+1)$-th character of u with the first character of v (if $o_{align} < 0$).

For each such alignment of v against u let r be the number of aligned characters (if $o_{align} \geq 0$ then $v[o_{align}+1]$ is aligned against $u[1]$ and $r = \min\{|u|, |v| - o_{align}\}$, otherwise $v[1]$ is aligned against $u[-o_{align}+1]$ and $r = \min\{|u| + o_{align}, |v|\}$). Let $r(u, v)$ be the number of characters of all alignments of v and u (for all possible values of o_{align}):

$$r(u,v) = \sum_{o_{align}=0}^{|v|} \min\{|u|, |v| - o_{align}\} + \sum_{o_{align}=-|u|}^{-1} \min\{|u| + o_{align}, |v|\}$$

Lemma 11. $r(u, v) = |u| \cdot |v|$

Proof.

$$\begin{aligned}
r(u,v) &= \sum_{o_{align}=0}^{|v|} \min\{|u|, |v| - o_{align}\} + \sum_{o_{align}=-|u|}^{-1} \min\{|u| + o_{align}, |v|\} \\
&= \sum_{i=0}^{|v|} \min\{|u|, |v| - i\} + \sum_{i=0}^{|u|} \min\{|u| - i, |v|\} - \min\{|u|, |v|\}
\end{aligned}$$

W.o.l.g. $|v| \geq |u|$:

$$\begin{aligned}
&= \sum_{i=0}^{|v|-|u|} |u| + \sum_{i=|v|-|u|}^{|v|} (|v| - i) + \sum_{i=0}^{|u|} (|u| - i) - |u| \\
&= (|v| - |u|) \cdot |u| + \sum_{i=0}^{|u|} (|u| - i) + \sum_{i=0}^{|u|} (|u| - i) - |u| \\
&= (|v| - |u|) \cdot |u| + 2 \cdot \frac{1}{2} \cdot |u| \cdot (|u| + 1) - |u| \\
&= |v| \cdot |u| - |u|^2 + |u|^2 + |u| - |u| \\
&= |v| \cdot |u|
\end{aligned}$$

□

Lemma 12. *There are $\mathcal{O}(l \cdot |S \setminus T| + ||S \setminus T||)$ different alignments of the pattern v against the strings in $S \setminus T$. The total number of aligned characters is $l \cdot ||S \setminus T||$.*

Proof. For each pair $v, u \in S \setminus T$, there are $|v|$ alignments with positive o_{align} and $|u|$ alignments with negative o_{align}. Hence, we have $\sum_{u \in S \setminus T}(|v| + |u| + 1) = \mathcal{O}(|v| \cdot |S \setminus T| + ||S \setminus T||)$ different alignments in total.

The number of alignments is $\sum_{u \in S \setminus T} |v| \cdot |u| = l \cdot \sum_{u \in S \setminus T} |u| = l \cdot ||S \setminus T||$ □

Figure 1 gives a schematic view of the way the algorithm works. Informally, for each value of o_{align} the strings u and v are aligned. A right maximal substring *cand′* with k mismatches is calculated. The next character will be a mismatch. Hence, the inclusion of this character leads to the desired right minimal k-distant v-substring *cand*. If *cand′* (or *cand*) does not start with a mismatch, the first suffix of *cand* is also a candidate (and possibly its next suffix and so on). If *cand′* (or *cand*) starts with a mismatch, the mismatch is dropped and the candidate is extended to the next mismatch with another lce query.

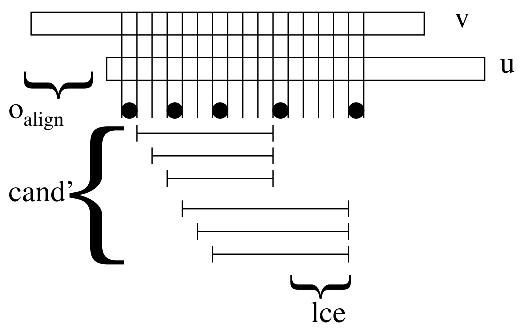

Figure 1: Schematic view of an alignment between v and $u \in S \setminus T$ and the way previous information is reused in the algorithm. (A black bullet denotes a mismatch, no bullet denotes a match.)

Calculating the lengths of the minimal prefixes of all suffixes $u[j]$ and $v[i]$ that match with distance k can be done in time $\mathcal{O}(|v| \cdot |u|)$. Algorithm 2 correctly computes right minimal k-distant v-substrings with respect to $\{u\}$ in the array $m[]$ in time $\mathcal{O}(|v| \cdot |u|)$, given that $\operatorname{lce}(i, j)$ can be computed in time $\mathcal{O}(\operatorname{lce}(i, j))$.

Theorem 13 (Correctness). *At the end of Algorithm 2,*

$$\forall i, 0 < i \leq |v| \quad m[i] = \begin{cases} l & \textit{if a right minimal } k\textit{-distant } v\textit{-substring of} \\ & \textit{length } l \textit{ starts at } i \textit{ in } v, \\ |v| + 1 & \textit{if no } k\textit{-distant } v\textit{-substring starts at } i \textit{ in } v \end{cases}$$

Proof. For the first case, suppose $v_i \ldots v_{i+l-1}$ is a right minimal k-distant v-substring. Then $v_i \ldots v_{i+l-1}$ matches any l-length substring w of u with $\operatorname{dist}_{ham}(v_i \ldots v_{i+l-1}, w) > k$. Since $v_i \ldots v_{i+l-1}$ is right minimal, there is a substring w' of length $l-1$, s.t. $\operatorname{dist}_{ham}(v_i \ldots v_{i+l-2}, w') = k$ (Lemma 7). W.l.o.g let $w' = u_j \ldots u_{j+l-2}$, furthermore let $o_{align} = i - j > 0$. Since $\operatorname{dist}_{ham}(v_i \ldots v_{i+l-1}, u_j \ldots u_{j+l-1}) > k$, either $\operatorname{dist}_{ham}(v_i \ldots v_{i+l-1}, u_j \ldots u_{j+l-1}) = k + 1$ and $v_{i+l-1} \neq u_{j+l-1}$ or $j + l - 2 = |u|$ and w' is a suffix of u.

Lines 8–12 and lines 17–21 are identical. If this block is entered with the condition

$$v_{pos_v+h-1} \neq u_{pos_u+h-1} \text{ ornot } (pos_v + h - 1 \in \{1, \ldots, |v|\} \wedge pos_u + h - 1 \in \{1, \ldots, |u|\}), \quad (1)$$

then afterwards condition (1) will also be true. Condition (1) is valid at the beginning because either $pos_v = 1$ or $pos_u = 1$ and $h = 0$. Hence, (1) is invariant throughout the execution of the algorithm.

Let h_b be the value of h before the block is entered and h_a the value of h afterwards. Then $u_{pos_u+h_b} \ldots u_{pos_u+h_a-2}$ matches $v_{pos_v+h_b} \ldots v_{pos_v+h_a-2}$. The length of the match $l = h_a - h_b - 1$ is stored in the next position of the array $e[]$ and pos_{err} is increased by one. The block of lines 8–12 (respectively lines 17–21) is passed through once for each mismatch between $v_{pos_v} \ldots v_{pos_v+h-1}$ and $u_{pos_u} \ldots u_{pos_u+h-1}$ (if $pos_v + h - 1 \in \{1, \ldots, |v|\}$ and $pos_u + h - 1 \in \{1, \ldots, |u|\}$) and once when the end of one of the strings is reached.

Let $pos_v - pos_u = o_{align} = i - j$ be chosen in line 3. If $v_{i+l-1} \neq u_{j+l-1}$ and $\operatorname{dist}_{ham}(v_i \ldots v_{i+l-1}, u_j \ldots u_{j+l-1}) = k + 1$, then the block of lines 8–12 (respectively lines 17–21) has been passed at least $k+1$ times when $pos_v + h - 1 = i + l - 1$ and $pos_u + h - 1 = j + l - 1$ is reached either after the first or after the second block. The algorithm enters the for-loop in lines 14–15 with $len_{match} = k + 1 + \sum_{t=1}^{k+1} e[pos_{err} - t]$ and condition (1) holds for $h = h - len_{match} - 1$. Hence, either u or v start at $h - len_{match}$, or there is a mismatch $u_{pos_u+h-len_{match}-1} \neq v_{pos_v+h-len_{match}-1}$. Therefore, $i > pos_v + h - len_{match} - 1$ and $j > pos_u + h - len_{match} - 1$. On the other hand, $i < pos_v + h - len_{match} + e[pos_{err} - k - 1] + 1$ and $j < pos_u + h - len_{match} + e[pos_{err} - k - 1] + 1$ because $u_{pos_u+h-len_{match}+e[pos_{err}-k-1]+1} \dots u_{pos_u+h-1}$ and $v_{pos_v+h-len_{match}+e[pos_{err}-k-1]+1} \dots v_{pos_v+h-1}$ match with k errors. Thus, there exists an $r \in \{0, \dots, e[pos_{err} - k - 1]\}$, s.t. $pos_v + h - len_{match} + r = i$ and $pos_u + h - len_{match} + r = j$ and $l = len_{match} - r$ will correctly be stored in $m[i]$ (line 15).

If $j + l - 2 = |u|$ and w' is a suffix of u, then the block of lines 8–12 (respectively lines 17–21) has been passed at least $k+1$ times when $pos_v + h - 1 = i + l - 1$ and $pos_u + h - 1 = j + l - 1$ is reached either after the first or after the second block. Since $pos_u + h - 1 = j + l - 2 + 1 = |u| + 1$ the second block of the if-statement in lines 26–28 will be entered. By the same argument as above, $i > pos_v + h - len_{match} - 1$ and $j > pos_u + h - len_{match} - 1$. Hence, there exists an $l' \in \{0, \dots, len_{match}]\}$, s.t. $i = pos_v + h - l' = i + l - l'$ and $l = l'$ will correctly be stored in $m[i]$ (line 28).

For the second case, suppose no k-distant v-substring starts at position i in v. Then the i-th suffix of v matches with distance less than k any substring of u, i.e. even the longest substring of v starting at i is matched somewhere in u with k or less mismatches. W.l.o.g. $\mathrm{dist}_{ham}(v_i \dots v_{|v|}, u_j \dots u_{j+|v|-i}) \leq k$ and $o_{align} = i - j > 0$.

Let $pos_v - pos_u = o_{align} = i - j$ be chosen in line 3. At some point after the block of lines 8–12 (respectively lines 17–21) has been passed, we will have $pos_v + h - 1 \notin \{1, \dots, |v|\}$ by condition (1). At the beginning $pos_v + h \in \{1, \dots, |v|\}$ and for each call to $\mathrm{lce}(\cdot, \cdot)$ we have that $pos_v + h + \mathrm{lce}(pos_v + h, \cdot) - 1 \leq |v|$. Therefore, $pos_v + h - 1 \leq |v| + 1$. It follows that $pos_v + h - 1 = |v| + 1$ and the first block of the if-statement in lines 23–24 will be entered.

In lines 4 and 5, len_{match} and h are set to zero. Throughout the first while-loop both values are increased synchronously. Only after at least $k + 1$ passes through the block of lines 8–12 len_{match} is decreased by the length of the $(k+1)$-th last mismatch each time a new mismatch or the end of the string is reached. Hence, if $h \neq len_{match}$, there are at least k mismatches between $u_{pos_u+h-len_match} \dots u_{pos_u+h-1}$ and $v_{pos_v+h-len_match} \dots v_{pos_v+h-1}$.

Since $v_i \dots v_{|v|}$ matches $u_j \dots u_{j+|v|-i}$ with no more than k errors, we have either $h = len_{match}$ and $pos_v = i$ or $pos_v + h - 1 - (len_{match} - 1) \leq i$. Hence, we always have $pos_v + h - len_{match} \leq i$ and therefore $|v| + 2 - len_{match} \leq i$ (since $pos_v + h - 1 = |v| + 1$).

As a result, h will run from $|v| + 2 - len_{match} \leq i$ to $|v| \geq i$ and $m[i]$ will be set to $|v| + 1$ in line 24. ☐

Theorem 14 (Complexity). *If* $\mathrm{lce}(i, j)$ *can be computed in time* $\mathcal{O}(\mathrm{lce}(i, j))$, *Algorithm 2 has a running time of* $\mathcal{O}(|u| \cdot |v|)$.

Proof. During each iteration of the while loops in lines 7 and 13, h is increased by at least one and the length $l = \mathrm{lce}(i, j)$ of the lce calculated in line 8, respectively line 17. The for loop of line 14 is iterated l times for each lce of length l. Hence the total number of iterations of any line between 7 and 21 is at most h. The for loops of line 23 and 26 are also iterated at most h times. Since h never exceeds the number of characters aligned when $v[pos_v]$ is aligned to $u[pos_u]$, the total number of iterations of any line of Algorithm 2 is at most $r(u, v) = \mathcal{O}(|u| \cdot |v|)$. ☐

Executing Algorithm 2 for all strings in $S \setminus T$ will therefore lead to a running time of $\mathcal{O}(l \cdot ||S \setminus T||)$.

The longest common extension of the i-th and j-suffix $\mathrm{lce}(i, j)$ can either be calculated directly in $\mathcal{O}(\mathrm{lce}(i, j))$ or with the use of a generalized suffix tree and lca queries in constant time.

In practice, calculating the lce by direct comparison of characters performs much better. Without constant time longest common extension queries no suffix tree needs to be build and no matching statistics need to be calculated. The expected number of characters to compare for finding the longest common extension under the assumption that the strings are generated by a memoryless source with probabilities $Pr[x_j = i] = p_i$ for $i \in \Sigma$ is constant:

$$\mathbb{E}[l : x_l \neq y_l \wedge \forall j < l \ \ x_j = y_j] = \frac{1}{1 - \sum_i p_i^2}.$$

For a uniform distribution over a four letter alphabet, the expected length is $4/3$. Hence, it is no surprise that this approach outperforms the constant time longest common ancestor calculations by far in practice.

5 An Improvement for the Levenshtein distance INEXACT CHARACTERISTIC STRING PROBLEM

The algorithm presented by Ito et al. [9] uses the diagonal method of dynamic programming (see [11]) to calculate the Levenshtein distance of a suffix v_i of the shortest string $v \in T$ against any suffix (and thereby substring) of a string $u \in S \setminus T$ with the help of constant time lce queries in time $\mathcal{O}(|u| \cdot k + |v|)$, where time $\mathcal{O}(|u| + |v|)$ is needed for the suffix tree construction and time $\mathcal{O}(|u| \cdot k)$ for calculating the Levenshtein distance of the $\mathcal{O}(|u|)$ suffixes of u against v_i. (This is possible since only distances of at $k + 1$ are considered.) The generalized suffix tree for u and v can be used for all suffixes v_i. Hence, the Levenshtein distance of all suffixes of v against all suffixes of v can be calculated in $\mathcal{O}(|v| \cdot |u| \cdot k)$. The second phase of calculating all right minimal k-distant v-substrings with respect to $S \setminus T$ can thus be implemented in time $\mathcal{O}(k \cdot l \cdot ||S \setminus T||)$, where l is the length of the shortest string in T. The Phases 1, 3, and 4 described in Section 4 will be the same. The modified algorithm then has a running time of $\mathcal{O}(||T|| + k \cdot l \cdot ||S \setminus T||)$. The term $l^2 \cdot |S \setminus T|$ of Ito et al. disappears because suffix trees are reused in the construction.

6 Conclusion

We have developed an efficient algorithm for the INEXACT CHARACTERISTIC STRING PROBLEM for Hamming distance. It improves over previous results by a factor of k, the number of mismatches allowed in matching with the non-target set. Our algorithm is faster even for $k = 1$ because we do not need suffix trees and constant time lca queries in the main part of the algorithm. The possibility to use different weights depending on the kind of mismatch allows a broader field of application. We have also improved the known INEXACT CHARACTERISTIC STRING PROBLEM algorithm by Ito et al. [9] by incorporating both algorithms for Levenshtein distance and Hamming distance in a common framework, thus pointing out the main differences.

The presented algorithm can be easily adapted to use arbitrary weights depending on the kind of mismatch. For applications in biology this can be used to reflect different base pair bonding strengths, thus allowing a more realistic probe design. The advantage of our algorithm for probe design is that a certain distance to every substring in the model is guaranteed.

The running time depends highly on the shortest string v in the target set T. For short sequences the algorithm is still feasible. Experiments with the small subunit database from the ARB project have shown that a single run with a 1500 bp sequence target (TmgMar22) and distance set of roughly twelve thousand sequences of total size 20 MB takes about 31 minutes on an Athlon XP1800+. The work can easily be distributed among multiple workstations scaling almost linearly to about 4 minutes on 10 machines.

Additionally, we presented a new practical, space efficient, and fast algorithm to solve the COMMON SUBSTRING PROBLEM, which outperforms previously known algorithms.

The NP-completeness results for the strongly related problems DISTINGUISHING STRING SELECTION PROBLEM and FARTHEST STRING PROBLEM make it unlikely that there exist efficient algorithms for versions of the CHARACTERISTIC STRING PROBLEM without a target set or that allow errors in matching the target set.

References

[1] AMIR, A., APOSTOLICO, A., AND LEWENSTEIN, M. Inverse pattern matching. *J. Algorithms 24*, 2 (1997), 325–339.

[2] CHANG, W. I., AND LAWLER, E. L. Sublinear approximate string matching and biological applications. *Algorithmica 12* (1994), 327–344.

[3] DE BRUIJN, N. G. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen* (1946), 758–764.

[4] FRANCES, M., AND LITMAN, A. On covering problems of codes. *Theory of Computing Systems 30* (1997), 113–119.

[5] GASIENIEC, L., INDYK, P., AND KRYSTA, P. External inverse pattern matching. In *Combinatorial Pattern Matching* (1997), pp. 90–101.

[6] GUSFIELD, D. *Algorithms on Strings, Trees, and Sequences – Computer Science and Computational Biology*. Press Syndicate of the University of Cambridge, 1997.

[7] HAMMING, R. W. Error detecting and error correcting codes. *Bell Syst. Tech. J.* (1950), 147–160.

[8] HUI, L. Color set size problem with applications to string matching. In *CPM: Proceedings of the 3rd Symposium on Combinatorial Pattern Matching* (1992), vol. 644 of LNCS, Springer, pp. 230–243.

[9] ITO, M., SHIMIZU, K., NAKANISHI, M., AND HASHIMOTO, A. Polynomial-time algorithms for computing characteristic strings. In *CPM: Proceedings of the 5th Symposium on Combinatorial Pattern Matching* (1994), vol. 807 of LNCS, Springer, pp. 274–288.

[10] LANCTOT, J. K., LI, M., MA, B., WANG, S., AND ZHANG, L. Distinguishing string selection problems. In *Proc. of the 10th SIAM-ACM Symposium on Discrete Algorithms* (1999), SIAM,ACM, pp. 633–642.

[11] LANDAU, G. M., AND VISHKIN, U. Introducing efficient parallelism into approximate string matching and a new serial algorithm. In *Proceedings of the 8th Annual ACM Symposium on Theory of Computing* (May 1986), ACM, ACM, pp. 220–230.

[12] LEVENSHTEIN, V. I. Binary codes capable of correcting deletions, insertions and reversals. *Doklady Akademii Nauk SSSR 163*, 4 (1965), 845–848.

[13] McCreight, E. M. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM 23*, 2 (April 1976), 262–272.

[14] Myers, E. W. An $o(nd)$ difference algorithm and its variations. *Algorithmica 1* (1986), 251–266.

[15] Nakanishi, M., Hasidume, M., Ito, M., and Hashimoto, A. A linear-time algorithm for computing characteristic strings. In *Proceddings of the 5th International Symposium on Algorithms and Computation* (1994), vol. 834 of LNCS, Springer, pp. 315–323.

[16] Schieber, B., and Vishkin, U. On finding lowest common ancestors: Simplification and parallelization. *SIAM J. Comput. 17*, 6 (December 1988), 1253–1262.

[17] Szpankowski, W. A generalized suffix tree and its (un)expected asymptotic behaviors. *SIAM J. Computing 22* (1993), 1176–1198.

[18] Ukkonen, E. On-Line Construction of Suffix Trees. *Algorithmica 14* (1995), 249–260.