Eötvös Loránd University

Faculty of Science

Bálint Márk Vásárhelyi

Suffix Trees and Their Applications

MSc Thesis

Supervisor:

Kristóf Bérczi
Department of Operations Research

Budapest, 2013

Acknowledgements

I would like to express my gratitude to my supervisor, Kristóf Bérczi, whose dedicated and devoted work helped and provided me with enough support to write this thesis.

I would also like to thank my family that they were behind me through my entire life and without whom I could not have finished my studies.

Soli Deo Gloria.

Contents

Introduction

Bioinformatics is a very promising and dynamic field of mathematics, in which a large scale of biological problems can be investigated using mathematical tools.

One of the crucial questions in bioinformatics is analysing large data sequences provided by several sequencers. A certain number of problems arise here, like finding a short sample sequence in another, usually longer sequence (in general, the longer sequence is called the string and the shorter sequence is the pattern) or comparing two sequences according to some distance function. More algorithms exist to give an answer for these problems.

When a long sequence is examined from more aspects, e.g. more patterns should be found in that, one might not want to read the long sequence more than once, as it consumes a lot of time. Rather, with a small extra effort, a handful data structure can be built in which the time to find the sample is proportional only to the length of the pattern and not of the long sequence.

This data structure is the suffix tree, which basically represents all the suffixes of a string in $\mathcal{O}(n)$ space and can be built in $\mathcal{O}(n)$ time, where n is the length of the string. In this thesis, we deal with an algorithm to create the suffix tree, review some applications and discuss a few related questions. The main source of the thesis is [10].

In Chapter 1, we introduce some basic definitions and examples. We also detail Ukkonen's linear time construction of a compact suffix tree and define the generalized suffix tree for more strings.

In Chapter 2, we mention some applications of the suffix tree, and we give a detailed description for some of them. Exact and inexact matching are very important to identify several characteristic parts of the genome. The problem of the longest common substring occurs frequently when the similarity of two sequences is in question. A related problem is the longest common subsequence and the "reverse" question is the super-string problem. Circular string linearisation is a task when one have to investigate bacterial genome. Repetitive structures often appear in the genome, suffix trees are a good tool for finding them. DNA-contamination can be also examined with the aid of suffix trees. Furthermore, in some genome-scale projects suffix trees have been used.

In Chapter 3, a related topic is in focus, namely, the matching statistics, which is a similar question to the size of a (non-compact) suffix tree. A few distance functions are introduced, and bounds are given for the edit distance.

In Chapter 4 We show some new results on the size of a suffix tree using a recursive formula for the number of aperiodic strings. In this chapter, another problem is also examined. Our question is that the length of the longest prefix of a string, which is also a suffix of its last $n-1$ characters. Although the question is easy in terms, the correct answer has not been found yet, and it can be the issue of further researches.

Chapter 1

Preliminaries

Suffix tree is a special data structure, which is useful for several combinatorial problems involving strings. The notion of suffix trees was introduced by Weiner in [18], although he did not use this name for them. Suffix trees have wide-range applications, including exact and inexact matching problems, substring problems, data compression and circular strings [10]. The motivation of this thesis is that in computational biology, molecular biology and bioinformatics these problems are crucially important. Numerous questions raise about sequencing and investigating DNA or RNA, such as looking for the longest common substring of two DNA sequences, finding exact and inexact matchings of a sample in a long sequence etc. It should be mentioned that there are some bioinformatical software tools based on suffix trees, such as Multiple Genome Aligner [11] or MUMmer [4]. The interested reader is also referred to [19], where one can find a useful overview of suffix trees.

1.1 Suffix tree

Following [10], we will introduce the following notations. Let S denote a string, the length of which is n. Let $S[i, j]$ denote the substring of S from position i to position j. Before constructing the suffix tree, we concatenate a new character, \$ to S. The importance of this character is twofold. First, by adding it to the string, one can avoid that a suffix is a prefix of another suffix, which is undesirable. Second, the generalization is also made easier by this operation. Now, we will define the suffix tree of a string. We always consider a fixed size alphabet; in [5] the case of unbounded alphabets is discussed.) A **suffix tree** is a rooted, directed tree. It has n leaves labelled from 1 to n, and its edges are labelled by characters of the alphabet. The label of an edge e is denoted by $\ell(e)$. On a path from the root to the leaf j one can read the suffix $S[j,n]$ of the string and a \$ sign. Such a path is denoted by $\mathcal{P}(j)$ and its label is referred as the **path label** of j and denoted by $\mathcal{L}(j)$. We call a leaf w **reachable** from the node v, if there is a directed path from v to w. An illustration of a suffix tree is shown on Figure 1.1.

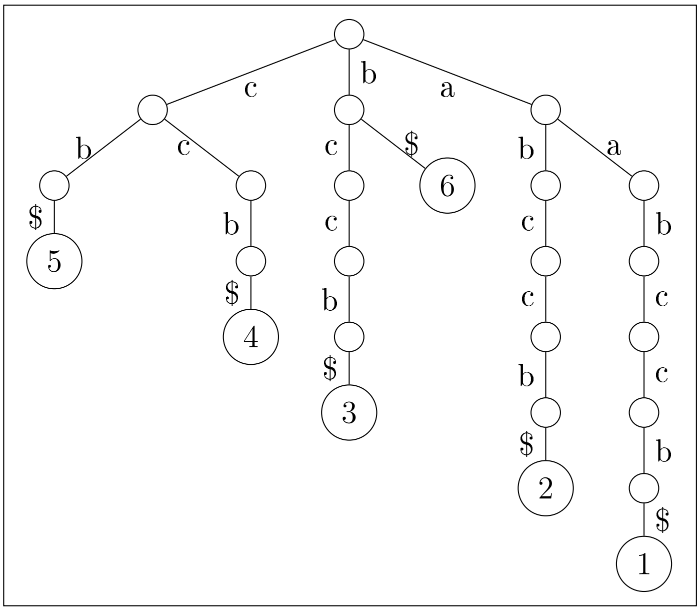

Figure 1.1: Suffix tree for the text aabccb

Suffix trees can use quite a lot of space. There are long branches which could be compressed. By compressing long branches one will receive the **compact suffix tree** of the string. Formally, a compact suffix tree of S is a rooted directed tree with n leaves. Each internal node has at least two children (the root is not an internal node). Each edge has a label with the property that if uv and uw are edges, then the first characters of the label of uv and of uw are distinct. The label of a path is the concatenation of the labels on its edges. An illustration of a compact suffix tree is shown on Figure 1.2.

1.2 A naive algorithm for constructing a suffix tree

Now, we will give a naive recursive algorithm for building the suffix tree of a string. $\mathcal{T}_i$ stands for the suffix tree built in phase i. As initiation, take a root and add an edge labelled by $S[1,n]\$$. In phase i, $\mathcal{T}_{i-1}$ is already built. First, add leaf i to the tree. Take the suffix $S[i,n]$ and find the longest path from the root whose label

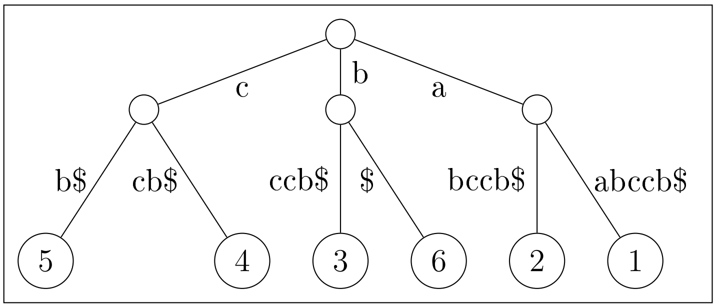

Figure 1.2: Compact suffix tree for the text aabccb

is its prefix. Suppose this label is $S[i,k]$. If the end of this label does not end in a node, create a new internal node and add a new edge from this node to leaf i labelled by $S[k+1,n]\$$.

This algorithm takes $\mathcal{O}(n^2)$ time.

1.3 Construct a compact suffix tree in linear time

Faster algorithms exist, such as Ukkonen's linear-time suffix tree algorithm ([16]), Weiner's linear-time suffix tree algorithm ([18]) or McCreight's suffix tree algorithm ([14]) (see also [10]).

The most widely used algorithm is Ukkonen's algorithm, which is for building the compact suffix tree. The main idea is that there are n *phases*, in each of them we construct a new tree from the previous one. The tree of phase i is a suffix tree for $S[1,i]$. The tree at the end of phase n is the suffix tree of S.

Now, we will give a detailed description of Ukkonen's algorithm according to [10]. The **implicit suffix tree** is obtained from the compact suffix tree by removing all \$ characters, then removing all edges with no label and at last, removing all nodes with at most one child. The implicit suffix tree encodes all the suffixes such that each suffix can be found by read the labels of a path from the root. However, if a path does not end in a leaf, it can correspond to a suffix, and in this case, there is no marker of the end of this path.

1.3.1 Outline of the algorithm

In Ukkonen's algorithm, we will build an implicit suffix tree $\mathcal{T}_i$ for each prefix $S[1,i]$ of S and finally, we construct the compact suffix tree from $\mathcal{T}_m$. The sketch of the algorithm is as it follows:

First, construct tree $\mathcal{T}_1$. In phase i ($i = 2, \ldots, n$) and extension j ($j = 1, \ldots, i$) find the path corresponding to $S[j, i-1]$ in the actual tree. If $S[j, i]$ does not occur in the tree, extend the path with this character, so that $S[j, i]$ is assured to be found in the implicit suffix tree of phase i.

1.3.2 Details of the algorithm

Nevertheless, it has to be specified how to extend the path $S[j, i-1]$, which we call a **suffix extension**. Let $\beta = S[j, i-1]$. First, find the last character of β and then extend it to make sure that $\beta S[i]$ occurs in the tree. For the extension, the following rules are to be applied:

Rule 1 If the end of β is a leaf, then append $S[i]$ to the end of the label on the last edge of the path of β. See Figure 1.3.

Rule 2 If at the end of β there is no path started with $S[i]$, create a new edge from this position to a new leaf, and label it with $S[i]$. If this position is inside an edge, then create here a new node and divide the label of the original edge. See Figure 1.4 and 1.5.

Rule 3 If $\beta S[i]$ is already in the tree, do nothing.

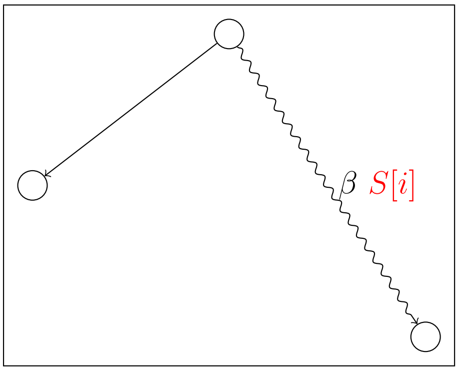

Figure 1.3: Extension rule 1

The extension takes only constant time. The key step is to find the end of β. A naive approach gives $\mathcal{O}(|\beta|)$ time by walking through $\mathcal{P}(\beta)$, where $|\beta|$ is the length of β. In this algorithm, extension j of phase i will take $\mathcal{O}(i-j)$ time, which means $\mathcal{T}_i$ can be created from $\mathcal{T}_{i-1}$ in $\mathcal{O}(i^2)$ time. This algorithm takes $\mathcal{O}(n^3)$ time.

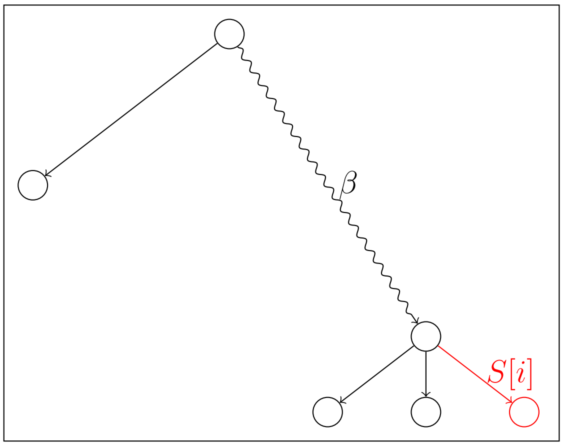

Figure 1.4: Extension rule 2

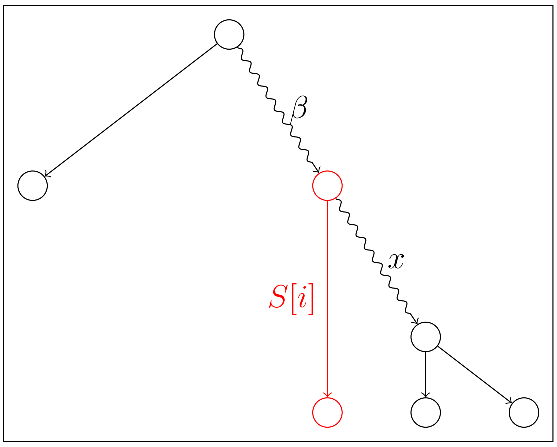

Figure 1.5: Extension rule 2

1.3.3 Speed-up of the algorithm

Ukkonen's linear time algorithm is a speed-up of algorithm introduced in Subsection 1.3.2 by applying a few tricks.

Suffix links

For introducing the first speed-up, we will define **suffix links**. An ordered pair $(v, s(v))$ of an internal and of an arbitrary node is a suffix link, if the label of $\mathcal{P}(v)$ is $x\alpha$ and the label of $\mathcal{P}(s(v))$ is α for a character x and a possibly empty substring α. See also Figure 1.6. The proof of this theorem can be found in [10].

Theorem 1. *If in extension j of phase i a new internal node is added and the label of $\mathcal{P}(v)$ is $x\alpha$, then either there is a suffix link from v to another node or there will*

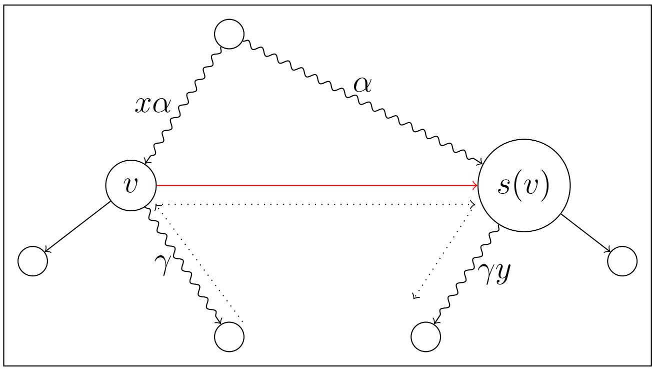

Figure 1.6: Suffix link

be created a new internal node w in extension $j+1$ such that the label of $\mathcal{P}(w)$ is α.

Theorem 1 implies that any new internal node will have a suffix link after the next extension.

Theorem 2. *In an implicit suffix tree $\mathcal{T}_i$ there is a suffix link from each internal node.*

Proof. Consequence of Theorem 1. □

This suffix link structure gives a short-cut to find the suffix $S[j,i]$ of $S[1,i]$ in extension j of phase i, which was previously determined by walking in the current tree. The application of suffix links is the following: in extension 1 of phase i, let $S[1,i] = x\alpha$ and $(v,1)$ is the edge to leaf 1. Now, the end of $S[2,i]$ has to be found in the current tree. If v is the root, just follow the path labelled α. If v is an internal node, there is a suffix link to $s(v)$, the path label of which node is α, and the algorithm can follow this suffix link. In the second extension of a phase, we take the edge $(v,1)$ as in extension 1. Let γ denote its edge label. In order to find α, walk back from leaf 1 to v, follow the suffix link to $s(v)$ and then follow the path labelled by γ. In the following extensions, start at the end of $S[j-1,i]$ (which is a leaf), walk back to the first node v on an edge labelled γ, follow the suffix link from it to $s(v)$ (if it is the root, walk through α to find its end) and walk down following the path which is also labelled γ.

The algorithm above using suffix links is called **Single extension algorithm**. However, the time bound of the Single extension algorithm is the same as the algorithm without suffix links, but this trick will be useful in the followings.

Skip/count trick

In the Single extension algorithm, there is a step when we walk down following a path labelled γ. This step takes $g = |\gamma|$ time, i.e. the number of characters. The skip/count trick can reduce this to the number of nodes on the path, which implies that the time of all the walk downs in a phase is $\mathcal{O}(n)$.

The main idea of this trick is that first, we find that edge e from $s(v)$ where $\ell(e)[1] = \gamma[1]$, and so, we can jump to the end of this edge without looking at the other characters of this edge.

Formally, let $g = |\gamma|$, $h = 1$ and $u = s(v)$. First, find the edge uw where $\ell(uw)[1] = \gamma[h]$. The length of $\ell(uw)$ is g'. If $g' < g$, then set $g = g - g'$, $h = g + g'$ and $u = w$. If $g' \geq g$, jump to character g on this edge and finish the algorithm, resulting in a path with label γ. Repeat these steps until reaching the end of γ.

We define the **node-depth** of a node u as the number of nodes on $\mathcal{P}(u)$, and denote this value with $\delta(u)$. The following theorem is shown in [10].

Theorem 3. *If $(v, s(v))$ is a suffix link, then $\delta(v) \leq \delta(s(v)) + 1$.*

If the last visited node is u, then the **current node-depth** of the algorithm is $\Delta = \delta(u)$. The skip/count trick ensures that any phase of Ukkonen's algorithm takes $\mathcal{O}(n)$ time, implying that the whole algorithm now runs in $\mathcal{O}(n^2)$ time.

Edge-label compression

For an $\mathcal{O}(n)$ algorithm, it is necessary to find an alternative representation of the edge labels, as if the edge labels contain all the characters, $\mathcal{O}(n^2)$ space is needed. The edge-label compression is a very simple idea: on each edge e, we only store a pair of indices, which are the indices in S of the first and the last character of $\ell(e)$. To find the corresponding characters in S takes only constant time, as the algorithm has a copy of S. Since there are at most $2n - 1$ edges, the suffix tree implemented in this way will use only $\mathcal{O}(n)$ symbols.

Stop if Rule 3 applies

If in phase i and extension j Rule 3 applies, it will apply in all further extensions in that phase, since if rule 3 applies, $S[j, i]$ is continued with $S[i + 1]$, therefore $S[j + 1, i]$ is also continued with $S[i + 1]$. The algorithm can be speeded up here such that if rule 3 applies, the end of $S[k, i]$ does not have to be found explicitly (for all $k > j$).

Once a leaf, always a leaf

If the algorithm creates a new leaf, there is no rule for changing it to an internal node. For any leaf edge, the edge label is of form (p, i), if we are in phase i. The label (p, i) will be changed to $(p, i+1)$ in the next phase. The trick here is to use a global index e meaning *the current end* and only changing e instead of changing all the leaf edge labels. In Rule 1, we do not change the label of the leaf edge, but we write the index (p, e) here instead of $(p, i+1)$. This replaces a certain number of extensions in phase $i+1$.

Single phase algorithm

Summarizing all the implementation tricks mentioned above, the algorithm will do the following in phase i: Increment index e to i, then compute extensions explicitly one by one, until extension rule 3 applies, in which case all extensions are done.

Ukkonen's algorithm implemented this way will take $\mathcal{O}(n)$ time for creating all the implicit suffix trees.

Creating the compact suffix tree

$\mathcal{T}_n$ can be converted to a compact suffix tree by adding character \$ to the end of S. We also have to replace the global index e with $n+1$ which stands for the extra \$ character. These modifications take $\mathcal{O}(n)$ time.

1.4 Generalized suffix trees

It is useful to construct one single suffix tree for more strings. First, take m strings, $S_1, \ldots, S_m$ with lengths $n_1, \ldots, n_m$, respectively. Add characters $\$_i$ to S_i, for $i = 1, \ldots, m$. The generalized suffix tree has $\sum\limits_{j=1}^{m} n_j$ leaves. Each leaf is labelled by the number j of the string and a number between 1 and n_j. The edges are labelled similarly to the edges of the simple suffix tree. The label of the path from the root to the leaf (i, j) represents the suffix $S_j[i, n_j]$ of the string j and a \$ character. One can define the compact generalized suffix tree similarly to the compact suffix tree.

Generalized suffix trees can be constructed in two ways. The first method is to take the concatenation of all strings $C = S_1\$_1 \ldots S_m\$_m$, and build the suffix tree of C. The leaves are numbered by the start position of the corresponding suffix in

C, which can be converted to the correct labels. The synthetic suffixes containing a \$ symbol inside can be quickly eliminated.

Nevertheless, it is also possible to add the strings one by one by extending Ukkonen's algorithm. When the implicit suffix tree of $S_1, \ldots, S_j$ is built, search the longest path matching to a prefix of S_{j+1}. If its length is i, then all suffixes of $S_{j+1}[1, i]$ are encoded, which means the algorithm can skip to phase $i+1$.

Chapter 2

Applications

In this section, some applications of the suffix tree will be given with an outlook to the biological applications. In a suffix tree one can find the occurrence of one or more patterns the total length of which is p in $\mathcal{O}(p)$ time. In molecular biology this is very useful when looking for special short sequences in a long DNA sequence. Generalized suffix trees are useful for determine some properties of strings. The longest common substring of two or more strings is an important element of the homology (similarity) tests between two different sequences. The shortest substrings occurring only once play a very important role in gene mapping, as these can be easily determined in a long sequence. The assembly of multiple strings, i.e. finding the shortest string containing each of them as a substring can be approximated by using suffix trees. Suffix trees can be also used for find all palindromes, tandem repeats or inexact matching. In the following, we will introduce some applications in details.

2.1 Matching

2.1.1 Exact matching

As it was already mentioned, suffix trees are useful when solving the exact matching problem. The **exact matching problem** is the following: we are given a **string** and a **pattern**, which are sequences over an alphabet, and we have to find all the occurrences of the pattern in the string. With a suffix tree, the solution of this problem is very easy (i.e. in linear time in the size of the string). Let n and m denote the length of the string S and of the pattern P, respectively.

First of all, observe that the pattern occurs in the string if it is the prefix of a suffix of the string. Therefore, taking the suffix tree of the string, we should search the first character of the pattern among the edges from the root. If this character does not appear, then the pattern does not occur in the string. If the first k characters of the pattern are found, then the edge labelled with the character $k+1$ of the pattern should be chosen. If there is no such edge, the pattern does not occur in the string. If all characters were found in the tree in the appropriate order, an occurrence of the pattern is found and the algorithm stops.

The **position** of an occurrence is the first character of the pattern in the string. Suppose that the algorithm above found an occurrence of the pattern and stopped at node v. Now, one would like to have all the positions of the occurrences of the pattern. Determine the leaves which are reachable from v on a directed path. Let the indices of these leaves be contained in the set J. If $j \in J$, then P is a prefix of the suffix $S[j, n]$, so the pattern occurs at the position j. If the pattern is the prefix of the suffix $S[j, n]$, then j will be obviously in J. Now, we found all the occurrences of the pattern, thus we solved the exact matching problem.

One can easily generalize the exact matching for more strings. First, the generalized suffix tree has to be built for all the strings, and then the algorithm above can be repeated. Now, the leaves have two labels. The pattern occurs in string i at position j if and only if the algorithm stops at an internal node from which the leave labelled by (i, j) is reachable on a directed path.

Complexity of the algorithm

Now, we will compare suffix trees and other linear-time algorithms (Knuth-Morris-Pratt, Boyer-Moore, see [17]) for exact matching. The linear-time algorithms take $\mathcal{O}(n) + \mathcal{O}(m)$ time for each matching problem. If we search k patterns in the same string, the running time is $k\mathcal{O}(n) + \sum_{i=1}^{k} \mathcal{O}(m_i)$, where m_i is the length of the pattern i.

Constructing the suffix tree takes $\mathcal{O}(n)$ time with Ukkonen's algorithm. Finding an instance of P in S takes $\mathcal{O}(m)$ time. If we search k patterns in the same string, the running time is $\mathcal{O}(n) + \sum_{i=1}^{k} \mathcal{O}(m_i)$.

For single searches, linear-time algorithms perform better, but if more patterns are to be found, the algorithm using suffix trees is faster.

2.1.2 Inexact matching

In molecular biology, the k-mismatch problem is a central topic. The nature of the genetic code allows some mismatches in a DNA sequence. The properties of two protein molecules can be similar even if there are several differences between their amino acid sequences. In several experiments RNA sequences have to be planned such that there are at least k mismatches between them.

Formally, the k-mismatch problem can be defined as it follows: given a pattern, a string and a number k, find those all substrings of the string matching the pattern with not more than k mismatches. (See [10].) For finding all k-mismatches of a pattern, the fastest way is to generate every string which mismatches the pattern at most k positions, and find all the occurrences of them using the exact matching algorithm. If k and the size of the alphabet is small enough, this gives a fast algorithm. For multiple strings, one can use the generalized suffix tree instead of the simple suffix tree.

2.2 Longest common substring

When multiple strings are given, an interesting question is to find the longest common substring of them. If the generalized suffix tree is built, this becomes an easy question.

Formally, the problem is to find the longest common substring of m given strings $S_1, \ldots, S_m$.

To solve the problem, build the generalized suffix tree as it is mentioned in Section 1.4. Then add a label to each internal node. An internal node v is labelled by $(i_1, i_2, \ldots, i_k)$, if for each j there is a leaf reachable from v labelled by the character i_j. Now, an internal node which is labelled by all the strings represents a common substring the length of which is the depth of the node. It is easy to see that all common substrings are represented in this way, therefore in order to find the longest substring it suffices to find the node with highest depth.

This algorithm is polynomial, so the longest common substring can be found in polynomial time.

2.3 Longest common subsequence

A related question is the longest common subsequence problem. One can get a **subsequence** of a string by leaving out arbitrary many characters of the string.

For two strings S_1 and S_2, a simple dynamic programming approach gives a good solution. Let $f(i, j)$ be the length of the longest common subsequence of $S_1[1, i]$ and $S_2[1, j]$. If the length of S_1 is n_1 and the length of S_2 is n_2, then the goal is to determine $f(n_1, n_2)$.

$$f(0, i) = f(j, 0) = 0 \quad \text{for all } i, j\,. \tag{2.1}$$

The recursion is the following:

$$f(i, j) = \begin{cases} f(i-1, j-1) + 1, & \text{if } S_1[i] = S_2[j], \\ \max\{f(i-1, j), f(i, j-1)\}, & \text{otherwise.} \end{cases} \tag{2.2}$$

For m strings $(S_1, \ldots, S_m)$, if the number of the strings is fixed, a similar dynamic programming algorithm can be easily constructed. Let $f(i_1, \ldots, i_m)$ be the longest common subsequence of $S_1[1, i_1], \ldots, S_m[1, i_m]$.

$$f(0, \ldots, 0, i) = \cdots = f(0, \ldots, 0, i, 0, \ldots, 0) = \cdots = f(i, 0, \ldots, 0) = 0 \quad \text{for all } i \tag{2.3}$$

The recursion is as follows:

$$f(i_1, \ldots, i_m) = \\ \begin{cases} f(i_1 - 1, \ldots, i_m - 1) + 1, & \text{if } S_1[i_1] = \cdots = S_m[i_m], \\ \max\{f(i_1 - 1, i_2, \ldots, i_m), \ldots, f(i_1, \ldots, i_{m-1}, i_m - 1)\}, & \text{otherwise.} \end{cases} \tag{2.4}$$

However, if the number of strings is unfixed, the problem becomes NP-hard. Gorbenko and Popov gave an explicit reduction to SAT and to 3-SAT in [9] and in [8].

2.4 Super-string

In the analysis of DNA, usually a large number of short sequences are read, which should be put together to find the original sequence. As a general approach, the shortest sequence containing each sequence as substring needs to be found. The **Super-string problem** is formulated as follows: $s_1, \ldots, s_n$ are given, and the task is to find S such that for all i, s_i is a substring of S. The super-string problem is NP-hard, as the Hamiltonian path problem can be reduced to it(see [6]).

For this problem, a 4-approximation heuristic is given in [12]. The main idea is to take two strings s_i and s_j such that a prefix t of s_i is a suffix of s_j, and then put these two strings together in a new string s_{ij}.

The greedy heuristic is the following: Let $A = \{s_1, \ldots, s_n\}$ be the set of strings. If s_i is a substring of s_j, remove s_i. If there are at least two identical strings, keep exactly one of them. Find two strings s_i, s_j in A such that their overlap is maximal, and let $A' = A \setminus \{s_i, s_j\} + s_i \cup s_j$, where $s_i \cup s_j$ is the shortest string containing s_i and s_j. Repeat the last step (which is called **blending**) until $|A|= 1$. For this algorithm, the generalized suffix tree for $s_1, \ldots, s_n$ can be used.

Removing the strings which are substrings of the others can be done in the following way: mark the parent nodes of the leaves with the same label as with the leaves are labelled, then throw away all the leaves. Now, a string s_i is a substring of another string if and only if label $(i, 1)$ appears on an internal node.

A **blended chain** is a set of p strings such that s_{i_k} starts after the beginning of s_{i_j}, and ends after the beginning of s_{i_l} for $j < k < l$. A blended chain has to appear in the order $s_{i_1}, \ldots, s_{i_p}$ in the super-string.

The blending can be done by maintaining two array sets: *chain* and *wrap*. For a blended chain, we maintain $chain(i_j) = i_{j+1}$ $(j < p)$, $chain(i_p) = 0$, $wrap(i_1) = i_p$ and $wrap(i_p) = i_1$.

We also need two index sets for each nodes. One for the strings with available suffixes (S_u) and one for the strings with available prefixes (P_u).

$$S_u = \{i|\exists d > 1 : (i, d)\text{is a label on a leaf of } u\text{'s subtree}\},$$

$$P_u = \{i|(i, 1)\text{is a label on a leaf of } u\text{'s subtree}\}.$$

The suffix labels of unavailable suffixes and prefixes are removed only when they are processed at the corresponding nodes. The set of available suffixes is $S_u \cap \{i|chain(i) = 0\}$. If $wrap(i) = j$, then $s_i \cup s_j$ is not feasible. If $i \in S_u$, $chain(i) = 0$, pick a j from P_u such that $s_i \cup s_j$ is feasible. If it is infeasible, take another i' from S_u with $chain(i') = 0$. For that, $s_{i'} \cup s_j$ must be feasible. Finally, set $chain(i) = j$, $wrap(wrap(i)) = j$ and $wrap(wrap(j)) = i$.

The algorithm is finished when no more pair of strings is feasible. Now, the array *chain* contains an order of the strings in which an approximately good superstring is achieved. For a slight modification of this algorithm, the 4-approximation is shown in [2].

2.5 Circular string linearisation

In biochemistry, circular molecules occur in many problems. In bacteria and in mitochondria, the DNA itself forms a circular molecule, but other circular structures are also known. One might would like to store the sequence of a circular molecule as a linear string, which is easier to store and search.

We call S a **circular string** if $S[n]$ precedes $S[1]$. If $j > i$, then a substring $S[j, i]$ is $S[j, n] + S[1, i]$, where $+$ stands for concatenation of the strings. We say S_1 is **lexicographically smaller** than S_2 ($S_1 \prec S_2$) if S_1 is shorter than S_2 or if they are equally long, S_1 precedes S_2 in alphabetical order.

A **cut** of S at i is a linear string containing the same characters as S, and started at position i of S, i.e. the concatenation of $S[i, n]$ and $S[1, i - 1$.

The **circular string linearisation problem** is to find the lexicographically smallest cut of S (see [10]). A naive algorithm for that is to take the lexicographical order of all cuts, which takes $\mathcal{O}(n^2 \log n)$ time. With suffix trees, this can be reduced to linear time in length of S.

First, take an arbitrary cut C of S and construct the suffix tree $\mathcal{T}$ of CC. Take and order of the alphabet, and let \$ greater than any other characters. Find a path in $\mathcal{T}$ with the following rule: at every node, the path takes the edge whose label starts with the smallest possible character. The label of this path is at least n long, since we cannot get into a leaf in less than n steps (as \$ is greater than any other character). If the label-length of the path reached n, stop and check the leaves in the subtree of the current node. Cut the string by using a leaf in the subtree. If leaf l is used for the cut and $1 < l \leq n$, cut S between character $l - 1$ and l. If $l = 0$ or $l = n + 1$, cut S between character n and 1. All cuts in the subtree will give the same linear string, which is the lexicographically smallest one.

2.6 Repetitive structures

In human and other eukaryote genomes repeated substrings often occur. Various models exist to explain the reason and the purpose of these repeats. Repetitive structures are very important with regard to DNA sequencing, gene mapping and other uses. Though in DNA usually inexact repeats occur, we discuss only exact repeats (for further details, see [10]).

We call a pair of identical S_1, S_2 substrings of S a **maximal pair** if the extension of them in any direction would make the two strings different. A maximal pair can be represented by a tuple (i_1, i_2, k), where i_1 and i_2 are the starting positions and k is the length of S_1 and S_2. The string $\alpha = S_1 = S_2$ is a **maximal repeat**. $\mathcal{R}(S)$ denotes the set of maximal repeats in S. A **super-maximal repeat** is a maximal repeat which is not a substring of any other maximal repeats.

For example, in the string $z\alpha xz\alpha y$ α is a maximal repeat, but not a super-maximal repeat, whereas $z\alpha$ is super-maximal.

2.6.1 Maximal Repeats

An obvious observation is that if $\alpha \in \mathcal{R}(S)$, then α is the path label of an internal node v of the compact suffix tree $\mathcal{T}$, as α occurs (at least) twice in S, and it is followed by different characters. Furthermore, this observation and the bound for the size of a compact suffix tree imply that the number of maximal repeats is at most n.

Let $S[i-1]$ be called the **left character** of position i. We say that a node v of $\mathcal{T}$ is **left diverse** if in the subtree of v at least two leaves exist with distinct left characters. Note that a leaf cannot be left diverse.

The following theorem is shown in [10].

Theorem 4. *A node is left diverse if and only if its path label α is a maximal repeat in S.*

Proof. If α is a maximal repeat, then v is obviously left diverse, as α must occur at least twice with distinct left characters.

If a node is left diverse, then there is an $x \neq y$ such that $x\alpha$ and $y\alpha$ are substrings in S. If the two occurrences are $x\alpha p$ and $y\alpha q$ (with $p \neq q$), then α is a maximal repeat. Suppose that the two occurrences are $x\alpha p$ and $y\alpha p$. As v is an internal node, there is a substring αq in S, where $q \neq p$. Now, if αq is not preceded by x, α is a maximal repeat with maximal pair $z\alpha q$, $y\alpha p$ (where $z \neq x$). Similarly, if αq is not preceded by y, α is again a maximal repeat. □

According to 4, finding the left diverse nodes in a compact suffix tree $\mathcal{T}$ is sufficient to find all the maximal repeats. An $\mathcal{O}(n)$ algorithm for this question is given in [10]. We are going through $\mathcal{T}$ from the bottom to the top. First, record the left characters of the leaves. In a general step, check all children of v. If any children is left diverse, v is also left diverse. If none of the children is left diverse, consider the recorded characters of them. If all of them are the same, mark v with this character. If there are at least two distinct characters among them, v is left diverse. Finally by deleting all nodes from $\mathcal{T}$ which are not left diverse, the tree of the left diverse nodes will be obtained.

2.6.2 Super-maximal Repeats

We call α a **near-super-maximal repeat** if there is an occurrence of α which is not included in another maximal repeat. Such an occurrence of α is called a **witness** for the near-super-maximality. Obviously, all super-maximal repeats are maximal repeats. For example, in $a\alpha bx\alpha ya\alpha b$ α is near-super-maximal, the witness of which is its second occurrence, though it is not super-maximal.

Let α be a maximal repeat, v be a node with path-label α and w be one of v's children. In the subtree of w the leaves correspond to some locations of α in S. Let the set of these occurrences be $L(w)$.

The following theorem [10] gives an approach to find all near-super-maximal repeats in linear time.

Theorem 5. *A near-super-maximal repeat is represented by a left diverse internal node v if and only if one of v's children is a leaf such that its left character is distinct from the left characters of all other leaves in v's subtree. A super-maximal repeat is represented by a left diverse node v if and only if all of its children are leaves with pairwise different left characters.*

Proof. Suppose that γ is the label of the edge vw. Now, the indices of $L(w)$ correspond to an occurrence $\alpha\gamma$.

If w is internal, $|L(w)|> 1$, therefore $\alpha\gamma$ occurs twice in S, which means these positions are not witnesses for the near-super-maximality of α.

If w is a leaf, let x be its left character. If x is the left character of any other leaf below v, $x\alpha$ occurs twice implying this occurrence of α is contained in a super-maximal repeat. If x is not the left character of any other leaf below v, $x\alpha$ occurs only once. If the first character of γ is y, then αy occurs in S only once, since w is a leaf. Thus a witness for the near-super-maximality of α found.

If each child of v is a leaf having pairwise different left characters, α is super-maximal, as the left character of any occurrence is different and the labels on the edges from v to the leaves are started with distinct characters. □

This theorem gives an approach to find all near-super-maximal repeats in linear time.

2.7 DNA contamination

When processing DNA in a laboratory, foreign DNA sequences often contaminate the sequence of interest, like the DNA of a vector or of the host organism. Even a very small amount of contamination can be inserted into the DNA or can be copied by the polymerase chain reaction (PCR), which is used to amplify the DNA sequences. DNA contamination is a very serious problem to be solved, as it can falsify the whole experiment. Usually, many potential contaminating DNA sequences are known, like cloning vectors, PCR primers, whole genome of the host etc.

This problem can be modelled in the following way (see [10]): we are given a string S (the DNA string of interest) and a string set $\mathcal{C}$ (the known possible contaminators). The goal is to find all substrings of all $T \in \mathcal{C}$ which occur in S and longer than a certain length ℓ. These are the substrings which are likely to be contaminators of the DNA string of interest. With suffix trees, the solution is to build the generalized suffix tree of S and $\mathcal{C}$, and mark all internal nodes v which is part of a path representing a suffix of S and of another one representing a suffix of an element of $\mathcal{C}$. Among the marked nodes, take all with path length at least ℓ. These will be the parts which are likely to be contaminated.

2.8 Genome-scale projects

Suffix trees have been applied in several genome projects. Three of these projects are the mapping of the *Arabidopsis thaliana*, the *Saccharomyces cerevisiae* (yeast) and the *Borrelia burgdorferi* genomes.

In the *Arabidopsis* project suffix trees were used in three ways (see [1]). First, they searched the contaminations by known vector DNA sequences. Here, a generalized suffix tree was created for the vector sequences. As a second step, all sequenced fragments were checked to find duplications. The fragment sequences were also kept in a generalized suffix tree. Third, suffix trees were also used to find biologically known and important sequences in the found sequences. Patterns were represented as regular expressions. The problem was formulated as an inexact matching problem with a small number of errors. They used suffix trees to give an answer to the question.

In the yeast and the *Borrelia* projects the suffix tree was the main data structure, and was used to solve the fragment assembly problem which is the following:

we are given a large number of fragments which are partially overlapping, and we have to find the full sequence.

Chapter 3

String matching statistics

3.1 Motivation

Inexact matching is a very important problem in bioinformatics. A frequent application is to find a mutated pattern in a mutated DNA sequence. Inexact matching is also useful when searching in a database, in a large text file or on the Internet due to mistype errors or in coding theory.

As we have mentioned before, inexact matching can be handled with the aid of suffix trees. Nevertheless, there is a large number of other methods to solve inexact matching. In [7] a wide range of these methods are introduced and analysed.

3.2 Distance of two strings

A good approach to inexact matching is to introduce a distance function of two strings such that the difference of two strings is smaller if the distance is smaller. However, the complexity of the problem can be linear but also NP-hard, depending on the distance function used. A family of the error models is when some substrings can be replaced to others for a certain cost. In this family, the goal is to minimize the cost of transforming the pattern to a substring of the text of interest.

The set of **operations** is a finite number of rules which are of the form $c(x, y) = t$, where x and y are different strings and $t \geq 0$ is the cost.

We call $d(S_1, S_2)$ the **distance** of the strings S_1 and S_2 if $d(S_1, S_2)$ is the minimal cost of the sequence of operations which transform S_1 to S_2. $d(S_1, S_2)$ is ∞ if S_1 cannot be transformed to S_2. If a substring was converted by an operation, it cannot be changed by any other operations.

The most widely used distance function is the **edit distance**. Generally, different costs are assigned to the insertion, deletion or replacement of a character, depending on the involved character (the first occurrence of this distance function is in [13]). The most simple of this type is called **Levenshtein-distance**, when all costs are 1.

A lot of interesting problems can be formulated with edit distance. E.g. the longest common subsequence problem can be solved by minimizing the edit distance when the costs of insertion and deletion are 1, and the cost of replacement is a very large number (practically infinity). If only replacements are allowed, we get back the Hamming distance. Edit distance can be also extended by allowing the transposition as a new operation, which is swapping to adjacent character of the string. The edit distance and the Hamming distance are symmetric (i.e. $d(S_1, S_2) = d(S_2, S_1)$). The episode distance allows only insertions, hence it is not symmetric.

3.3 Matching statistics

One might be interested about the probability of a match, or the expected value of the number of matches. Suffix trees are also concerned at this point, as the number of the internal nodes of a suffix tree basically depends on the maximal length of such a prefix of S which is a substring of $S[2, n]$ at the same time (see 4).

For the edit distance model, a simple model is given in [7]. We suppose that each character occurs with probability $\frac{1}{\sigma}$, where σ is the size of the alphabet. In the referred paper, it is shown that the edit distance $e(S_1, S_2)$ of two strings with length n is between $n - l(S_1, S_2)$ and $2(n - l(S_1, S_2))$, where $l(S_1, S_2)$ stands for their longest common subsequence. In [3] it is shown that $l(S_1, S_2)$ is bounded by $\frac{n}{\sqrt{\sigma}}$ and $\frac{n}{\sqrt{\sigma}}\mathrm{e}$. In [15] it is proved that if σ is large enough, $e(S_1, S_2) \approx n\left(1 - \frac{1}{\sqrt{\sigma}}\right)$.

The probability of a matching within k errors is another important question. Let $f(m, k)$ denote the probability that an m-length pattern matches the text at a certain position with at most k errors. The analytical bounds for $f(m, k)$ are the followings: $\frac{\delta^m}{\sqrt{m}} \leq f(m, k) = \mathcal{O}(\gamma^m)$, where $\delta = \left(\frac{1}{(1-\alpha)\sigma}\right)^{1-\alpha}$ and $\gamma = \left(\frac{1}{\sigma\alpha^{\frac{2\alpha}{1-\alpha}})(1-\alpha)^2}\right)^{1-\alpha}$ and $\alpha \leq 1 - \frac{\mathrm{e}}{\sqrt{\sigma}}$ is the maximum error level [7].

Theorem 6. *The average edit distance is at least*

$$n\left(1 - \frac{e}{\sqrt{\sigma}}\right). \tag{3.1}$$

Proof. Let $p(n,k)$ be the probability that $e(S_1,S_2) \leq k$ for the strings S_1, S_2 of length n. $p(n,k)$ is obviously at most $f(n,k)$. Let ed denote the value of the edit distance of two random strings of length n.

The average distance is formulated as

$$\begin{aligned}\sum_{k\,=0}^{n} k \cdot P(ed = k) = \sum_{k\,=0}^{n} P(ed > k) = \sum_{k=0}^{n} (1 - p(n,k)) \\ \geq n - \sum_{k=0}^{n} f(n,k) \geq n - n\left(1 - \frac{\mathrm{e}}{\sqrt{\sigma}}\right),\end{aligned} \tag{3.2}$$

using the analytical bounds for $f(m,k)$. □

3.4 Dynamic programming approach

The edit distance can be computed with an algorithm based on dynamic programming. Let $C_{i,j} = e(S_1[1,i], S_2[1,j])$ if $i \in \{1, \ldots, |S_1|\}$ and $j \in \{1, \ldots, |S_2|\}$. Let $C_{0,j} = j$ and $C_{i,0} = i$.

$$C_{i,j} = \begin{cases} C_{i-1,j-1} & \text{if } x_i = y_j \\ \min\{C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1}\} & \text{otherwise.} \end{cases}$$

For a detailed description of the algorithm, see [10] or [17].

3.5 Adaptation for text searching

The algorithm of 3.4 can be adapted to find a pattern in a string. Now, the initialization is slightly different. As any position of the string can be the start of an occurrence of the pattern, $C_{0,j}$ should be defined as 0 for all $j \in \{0, \ldots, n\}$, where n is the length of the string. The rest remains the same as it was in the algorithm for computing the edit distance.

Chapter 4

Size of a suffix tree

When constructing a suffix tree, an interesting question is the number of the internal nodes, which are all the nodes excluding the leaves and the root. A suffix tree of an n-long string has n leaves, but the number of internal nodes is not always the same. A trivial upper bound is $\frac{n(n+1)}{2}$, which one might get by adding to the suffix tree for each suffix one branch with the length of the suffix. A trivial lower bound is $2n + 1$, which one might get from a string which is of n same characters.

If we consider the compact suffix tree, a trivial upper bound for the number of internal nodes is $n - 1$, as there are exactly n leaves, and each internal node has at least two children.

4.1 The growth of the suffix tree

If one's goal is to give an estimation on the size of a suffix tree, a possible approach is a recursive way. If the suffix tree of the last $n - 1$ character has been already constructed, only the suffix $S[1, n]$ have to appear in the suffix tree. The question is how many new nodes will be created.

If $S[1, n]$ has a prefix which is also the prefix of $S[j, n]$ (where $1 < j$), the tree can be branched where this common prefix finishes. If it is at $S[1, k]$, then $n - k$ new nodes are created. Therefore the goal is to find the longest substring of $S[2, n]$ which is a prefix of $S[1, n]$, whose length we call the **overlap** and denote by $\omega(S)$. If the overlap is k, then $n - k$ new nodes are created. $\gamma(S)$ stands for the number of new nodes. $\omega(S) + \gamma(S) = n$ by definition.

$\Omega(n, k)$ denotes the number of n-length strings S with $\gamma(S) = k$. (I.e. $\Omega(n, k) = |\{S : |S|= n,\ \gamma(S) = k\}|$).

We used a program written in Python (see 5.1) to determine $\Omega(n, k)$ for $n = 1, \ldots, 20$ over an alphabet of two characters. The results are shown in Table 4.1.

We will prove the following theorem:

Theorem 7. $\forall k\ \exists \phi(k) :\ \forall n \geq 2k - 1\ \ \Omega(n, k) = \phi(k)$.

Observation 1. *For any value of k there are but only a few possibilities to create a string with $\gamma(S) = k$. First, we decide how the overlap will occur in the string, i.e. we set an index $j \leq k$ such that $S[1, n-k] = S[j+1, j+1+n-k]$. We call the positions $1, \ldots, j$* ***first-type quasi-free*** *and $j+n-k+2, \ldots, n$* ***second-type quasi-free*** *positions. Obviously, if the first-type quasi-free positions are given, $S[j+1, j+1+n-k]$ is determined due to the condition $S[1, n-k] = S[j+1, j+1+n-k]$. The first of the second-type quasi-free characters cannot be arbitrary: $S[1, j+1+n-k]$ can be continued in exactly one way to make the overlap longer, therefore we have $\sigma - 1$ ways to ensure that the overlap is k. The rest of the second-type quasi-free characters can be chosen arbitrarily, which gives $(\sigma - 1)\sigma^{k-j-2}$ possibilities for a fix j.*

The possibilities for the first-type quasi-free characters are slightly more difficult to count. We should be aware of the periods which can occur in this part of the string; if periodicity appears, the overlap becomes longer.

We have to count the number of j-length aperiodic strings, which we denote with $\mu(j)$.

Theorem 8. $\mu(j) = \sigma^j - \sum\limits_{d|j, d\neq j} \mu(d)$

Proof. $\mu(1) = \sigma$ is trivial.

There are σ^j strings of length j. Suppose that a string is periodic of period exactly d. This implies that its first d characters form an aperiodic string of length d, and there are $\mu(d)$ such strings. This finishes the proof. □

Theorem 9. *If p is prime, then $\mu(p) = \sigma^p - \sigma$.*

Proof. It is a clear consequence of Theorem 8. □

Theorem 10. *If $q = p^t$, where p is prime and $t \in \mathbf{N}$, then $\mu(p^t) = \sigma^{p^t} - \sigma^{p^{t-1}}$.*

Proof.

$$
\begin{aligned}
\mu(q) &= \sigma^q - \sum_{0\le s<t} \mu(p^s) \\
&= \sigma^q - \mu(p^{t-1}) - \sum_{0\le s<t-1} \mu(p^s) \\
&= \sigma^q - \sigma^{p^{t-1}} + \sum_{0\le s<t-1} \mu(p^s) - \sum_{0\le s<t-1} \mu(p^s) \\
&= \sigma^{p^t} - \sigma^{p^{t-1}}.
\end{aligned}
\tag{4.1}
$$

□

Theorem 11. *If $n \ge 2k-1$, then $\phi(k) = \sum_{j=1}^{k-1} \mu(j)(\sigma-1)\sigma^{k-j-1} + \mu(k)$.*

Proof. This theorem is the consequence of Observation 1. □

Proof. (Theorem 7) In Theorem 11, it is clear that $\phi(k)$ is independent of n. □

We verified the results of Theorem 7 for the first few values of k and for any alphabet of size σ (I.e. we counted all possible strings with overlap k for any j and checked whether they are in agreement with the results of the simulation).

For $\underline{k=1}$, the only possibility is that all characters of the string are the same, as $S[1,n-1] = S[2,n]$, therefore $S[1] = S[2] = \cdots = S[n]$. This gives σ strings, so $\phi(1) = \sigma$. If $\sigma = 2$, it is 2.

Using Theorem 8 and Observation 1, we got the following results:

$\underline{k=2}$:

$j = 1$: $\sigma(\sigma-1)$

$j = 2$: $\sigma^2 - \sigma$

As a total, we have $\phi(2) = 2\sigma(\sigma-1)$. If $\sigma = 2$, it is 4.

$\underline{k=3}$:

$j = 1$: $\sigma(\sigma-1)\sigma$

$j = 2$: $(\sigma^2-\sigma)(\sigma-1)$

$j = 3$: $\sigma^3 - \sigma$

As a total, we have $\phi(3) = 3\sigma^2(\sigma-1)$. If $\sigma = 2$, it is 12.

$\underline{k=4}$:

$j = 1$: $\sigma(\sigma-1)\sigma^2$

$j = 2$: $(\sigma^2 - \sigma)(\sigma - 1)\sigma$

$j = 3$: $(\sigma^3 - \sigma)(\sigma - 1)$

$j = 4$: $\sigma^4 - \sigma^2$

As a total, we have $\phi(4) = 4\sigma^3(\sigma - 1) - \sigma(\sigma - 1)$. If $\sigma = 2$, it is 30.

$\underline{k = 5}$:

$j = 1$: $\sigma(\sigma - 1)\sigma^3$

$j = 2$: $(\sigma^2 - \sigma)(\sigma - 1)\sigma^2$

$j = 3$: $(\sigma^3 - \sigma)(\sigma - 1)\sigma$

$j = 4$: $(\sigma^4 - \sigma^2)(\sigma - 1)$

$j = 5$: $\sigma^5 - \sigma$

As a total, we have $\phi(5) = 5\sigma^4(\sigma - 1) - \sigma(\sigma - 1)^2$. If $\sigma = 2$, it is 78.

4.2 Lower bound for the expectation of the growth

Now, we will give a lower bound on the expected value of $\gamma(S)$.

Theorem 12. *For all $j > 1$ $\mu(j) \leq \sigma^j - \sigma$.*

Proof. $\mu(j) = \sigma^j - \sum\limits_{d|j} \mu(d)$

Considering $\mu(d) \geq 0$ and $\mu(1) = \sigma$, we have $\mu(j) \geq \sigma^j - \sigma$. □

Theorem 13. *For all $j > 1$ $\mu(j) \geq \sigma^{j-1}$.*

Proof. Given a string of length $j-1$, there is at most one character which makes it periodic if we append it to the end, so there is at least one character which makes it aperiodic. □

Theorem 14. *If $n \to \infty$, $\mathbb{E}(\gamma(S)) \geq \frac{n}{2}$)*

Proof. According to Theorem 12, $\mu(j) \leq \sigma^j - \sigma$ (if $j > 1$).

By Theorem 11, if $k > 1$,

$$\begin{aligned}\phi(k) &= \mu(k) + \sum_{j=1}^{k-1} \mu(j)(\sigma-1)\sigma^{k-j-1} \\ &\leq \sigma^k - \sigma + \sigma + \sum_{j=2}^{k-1} (\sigma^j - \sigma)(\sigma-1)\sigma^{k-j-1} \\ &= (k-1)(\sigma-1)\sigma^{k-1} + \sigma.\end{aligned} \tag{4.2}$$

[1]

Using Theorem 13 and Theorem 11,

$$\begin{aligned}\phi(k) &= \mu(k) + \mu(1) + \sum_{j=2}^{k-1} \mu(j)(\sigma-1)\sigma^{k-j-1} \\ &\geq \sigma^k - \sigma + \sigma + \sum_{j=2}^{k-1} \sigma^{j-1}(\sigma-1)\sigma^{k-j-1} \\ &= \sigma^k + \sum_{j=2}^{k-1} (\sigma-1)\sigma^{k-2} \\ &= (k-2)(\sigma-1)\sigma^{k-2} + \sigma^k\end{aligned} \tag{4.3}$$

Let $m = \lceil \frac{n}{2} \rceil$.

Now,

$$\begin{aligned}\sum_{k\,=1}^{m} \phi(k) &\leq \frac{(m-1)\sigma^{m+1} - m\sigma^m + m\sigma^2 - m\sigma - \sigma^2 + 2\sigma}{\sigma - 1} + \sigma \\ &\leq m\sigma^m,\end{aligned} \tag{4.4}$$

if m is large enough. As $\sigma^n \gg \frac{n}{2}\sigma^{\frac{n}{2}}$, this implies that in most cases, the suffix tree will expanded by more than $\frac{n}{2}$ new internal nodes.

A lower bound on the expectation of $\gamma(S)$ is

$$\begin{aligned}\frac{1}{\sigma^n}\left(\frac{n}{2}\sigma^{\frac{n}{2}} + \left(\sigma^n - \frac{n}{2}\sigma^{\frac{n}{2}}\right)\left(\frac{n}{2}+1\right)\right) &= \frac{1}{\sigma^n}\left(\frac{n+2}{2}\sigma^n + \left(\frac{n+2}{2} - \frac{n(n+2)}{4}\right)\sigma^{\frac{n}{2}}\right) \\ &= \frac{n}{2} + 1 + \mathcal{O}\left(\frac{1}{\sigma^{\frac{n}{2}}}\right).\end{aligned} \tag{4.5}$$

The right hand side of 4.5 tends to $\frac{n}{2} + 1$.

□

[1] The computations are performed by Wolfram Alpha.

Using the result of Theorem 14, it can be shown that the expectation of the size of a (not compact) suffix tree is very large, as it is larger than $\sum_{m=1}^{n} \mathbb{E}(\gamma(S_m))$, where S_m is a string of length m. (Remember that the trivial upper bound for the number of internal nodes was n^2.)

$$\begin{aligned}\sum_{m\,=1}^{n} \mathbb{E}(\gamma(S_m)) &\geq \sum_{m=1}^{n} \frac{m}{2} \\ &= \frac{n(n+1)}{4}.\end{aligned} \tag{4.6}$$

This result illustrates that the (not compact) suffix trees use too much space in nearly all cases, which supports the usage of compact suffix trees.

4.3 Autocorrelation of a String

An interesting related question is the following: what is the length of the longest prefix of $S[1,n]$ which is a suffix of $S[2,n]$. This we call the **autocorrelation** of the string S and denote by $\alpha(S)$. The autocorrelation is easy to determine for one string, but we were curious about the expected value of the autocorrelation of an n-long string, i.e. the quotient of the sum of all autocorrelations and the number of all n-long strings. If the size of the alphabet is z, then there are z^n different strings. The goal is to determine a formula for the sum of the autocorrelations of all n-length strings.

We used a program written in Python (see 5.2) to determine the expected value of the autocorrelation of an n-long string. For this purpose, we determined the sum of autocorrelations of all n-long strings over an alphabet with size σ, for more values of n and σ. The number of n-long strings over an alphabet with size σ is σ^n. The results are shown in Table 4.3. In Table 4.4 the expected values are shown.

$\Omega(n, \gamma(S))$		Number of new internal nodes ($\gamma(S)$)											
		1	2	3	4	5	6	7	8	9	10	11	12
Length of string (n)	1	2											
	2	2	2										
	3	2	4	2									
	4	2	4	8	2								
	5	2	4	12	12	2							
	6	2	4	12	26	18	2						
	7	2	4	12	30	52	26	2					
	8	2	4	12	30	70	98	38	2				
	9	2	4	12	30	78	150	178	56	2			
	10	2	4	12	30	78	176	320	316	84	2		
	11	2	4	12	30	78	180	394	662	556	128	2	
	12	2	4	12	30	78	180	420	856	1342	972	198	2
	13	2	4	12	30	78	180	432	928	1844	2676	1694	310
	14	2	4	12	30	78	180	432	970	2056	3896	5282	2950
	15	2	4	12	30	78	180	432	978	2174	4466	8156	10334
	16	2	4	12	30	78	180	432	978	2208	4764	9636	16920
	17	2	4	12	30	78	180	432	978	2220	4868	10394	20562
	18	2	4	12	30	78	180	432	978	2220	4918	10688	22450
	19	2	4	12	30	78	180	432	978	2220	4926	10838	23212
	20	2	4	12	30	78	180	432	978	2220	4926	10888	23596

Table 4.1: Number of strings with fixed length and fixed $\gamma(S)$, with an alphabet of two characters

$\Omega(n, \gamma(S))$		Number of new internal nodes							
		13	14	15	16	17	18	19	20
Length of string	13	2							
	14	490	2						
	15	5140	780	2					
	16	20074	8968	1248	2				
	17	34856	38774	15676	2004	2			
	18	43552	71358	74554	27460	3226	2		
	19	48156	91656	145344	142804	48212	5202	2	
	20	50110	102608	191896	294700	272672	84844	8398	2

Table 4.2: Number of strings with fixed length and fixed $\gamma(S)$ (continuation)

$\sum \alpha(S)$		Size of alphabet			
		2	**3**	**4**	**5**
Length of word	**2**	2	3	4	5
	3	6	12	20	30
	4	16	45	96	175
	5	36	144	400	900
	6	82	465	1676	4645
	7	176	1434	6792	23390
	8	372	4395	27448	117615
	9	768	13296	110120	588840
	10	1582	40185	441636	2947585
	11	3224	120930	1767952	
	12	6534	363603	7076108	
	13	13166	1091784		
	14	26504	3277803		
	15	53244	9836580		
	16	106824			
	17	214060			
	18	428764			
	19	858400			
	20	1718056			
	21	3437734			
	22	6877896			
	23	13759154			
	24	27523128			

Table 4.3: Sums of autocorrelations

$\frac{\sum \alpha(S)}{n}$		Size of alphabet			
		2	**3**	**4**	**5**
Length of word	2	0,5	0,333333	0,25	0,2
	3	0,75	0,444444	0,3125	0,24
	4	1	0,555556	0,375	0,28
	5	1,125	0,592593	0,390625	0,288
	6	1,28125	0,63786	0,40918	0,29728
	7	1,375	0,655693	0,414551	0,299392
	8	1,453125	0,669867	0,418823	0,301094
	9	1,5	0,675507	0,420074	0,301486
	10	1,544921875	0,680537	0,421177	0,301833
	11	1,57421875	0,682653	0,421513	
	12	1,595214844	0,684183	0,421769	
	13	1,607177734	0,684795		
	14	1,617675781	0,685307		
	15	1,62487793	0,685528		
	16	1,630004883			
	17	1,633148193			
	18	1,635604858			
	19	1,637268066			
	20	1,638465881			
	21	1,639239311			
	22	1,639818192			
	23	1,640218973			
	24	1,640506268			

Table 4.4: Expected values of autocorrelations

Chapter 5

Appendix

5.1 The Program Used for Examining the Size of the Suffix Tree

```
import sets
import itertools

def szoveggyar(hossz,abc):
    ls = [''.join(x) for x in itertools.product(abc, repeat=hossz)]
    return ls

def korrel(s1,s2):
    hossz = min(len(s1),len(s2))
    for i in range(0,hossz+1):
        j = hossz-i
        if s1[0:j] == s2[0:j] :
            return j

def egyezes(s):
    aktual = 0
    for i in range(1,len(s)):
        k = korrel(s,s[i:])
        if aktual < k:
            aktual = k
        if aktual > len(s)-i:
            return aktual
    return aktual

def newinternalnodes(s):
    return len(s)-egyezes(s)

for i in range(1,21):
    negyes = szoveggyar(i,'ab')
    eredmenyek = []
    for elem in negyes:
         eredmenyek.append(newinternalnodes(elem))
    darabok = []
    for j in range(1,i+1):
        x = eredmenyek.count(j)
        darabok.append(str(j) + ': ' + str(x))
    print darabok
```

5.2 The Program Used for Determining the Autocorrelation

```
import sets
import itertools
def szoveggyar(hossz,abc):
    ls = [''.join(x) for x in itertools.product(abc, repeat=hossz)]
    return ls

def korrelacio(s1,s2):
    hossz1 = len(s1)
    hossz2 = len(s2)
    hossz = min(hossz1,hossz2)
    match = [0]
    for i in range(1,hossz+1):
        if s1[:i] == s2[-i:]:
            match.append(i)
    return match[-1]
```

```
def autokorrelacio(s1):
    return korrelacio(s1,s1[1:])

def szamol(j,n):
    abc = ’’
    for s in range(0,n):
        abc+=(str(s))
    adatsor = szoveggyar(j,abc)
    egyez = []
    for szoveg in adatsor:
        egyez.append(autokorrelacio(szoveg))
    egyez.sort()
    egyez.reverse()
    exp_value = [sum(egyez),len(egyez)]
    return exp_value
```

Summary

In this thesis we gave a short review of suffix trees and their uses. This data structure proved to be an efficient tool in biomathematics. Large sequences can be analysed and compared faster by algorithms using suffix trees.

The matching and the longest common substring problems among other questions can be solved efficiently, and other problems, like the super-string problem, can be approximated. Suffix trees were also used in genome scale projects.

A close issue is the matching statistics of strings, where a wide scale of open questions raises. Although the distance of two strings can be estimated for some kinds of distances, the autocorrelation of a string or the size of a suffix tree are hard to approximate. The answers for these questions can be the goal of further researches.

Bibliography

[1] P. Bieganski. *Genetic sequence data retrieval and manipulation based on generalized suffix trees.* PhD thesis, University of Minnesota, 1995.

[2] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. *Proceedings of the 23th Annual ACM Symposium on the Theory of Computing*, pages 329–336, 1991.

[3] V. Chvátal and D. Sankoff. Longest common subsequences of two random sequences. *Journal of Applied Probability*, 12:306–315, 1975.

[4] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[5] M. Farach. Optimal suffix tree construction with large alphabets. *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 137–143, 1997.

[6] J. Gallant, D. Maier, and J. A. Storer. On finding minimal length superstrings. *Journal of Computer and System Sciences*, 20:50–58, 1980.

[7] G.Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33:31–88, 2001.

[8] A. Gorbenko and V. Popov. The longest common subsequence problem. *Advanced studies in Biology*, 4:373–380, 2012.

[9] A. Gorbenko and V. Popov. On the longest common subsequence problem. *Applied Mathematical Sciences*, 6:5781–5787, 2012.

[10] D. Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1997.

[11] M. Höhl, S. Kurtz, and E. Ohlebusch. Efficient multiple genome alignment. *Bioinformatics*, 18:312–320, 2002.

[12] S. R. Kosaraju and A. L. Delcher. Large-scale assembly of DNA strings and space-efficient construction of suffix trees. *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing*, pages 169–177, 1995.

[13] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10:707–710, 1966.

[14] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23:262–272, 1976.

[15] D. Sankoff and S. Mainville. Common subsequences and monotone subsequences. In D. Sankoff and J. B. Kruskal, editors, *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, pages 363–365. Addison–Wesley, 1983.

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

[17] B. M. Vásárhelyi. *Mathematical methods in DNA sequence analysis.* BSc diploma thesis, Eötvös Loránd University, 2011.

[18] P. Weiner. Linear pattern matching algorithms. *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[19] Wikipedia. Suffix tree–wikipedia, the free encyclopedia, 2013. `http://en.wikipedia.org/w/index.php?title=Suffix_tree&oldid=543856414`, Online; accessed 9-April-2013.