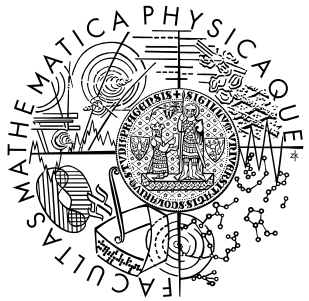
Charles University in Prague
Faculty of Mathematics and Physics

MASTER'S THESIS



Peter Bašista

Suffix tree construction with minimized branching

Department of Software and Computer Science Education

Supervisor: RNDr. Tomáš Dvořák, CSc.
Study program: Informatics
Specialization: Theoretical Computer Science (ITI)

Prague 2012

I would like to thank my supervisor RNDr. Tomáš Dvořák, CSc. for his guidance, support and feedback during the entire time of working on this thesis.

I thereby declare that I carried out this master's thesis independently and used exclusively the cited resources.
I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague August 3, 2012 Peter Bašista

Title: Suffix tree construction with minimized branching
Author: Peter Bašista
Department: Department of Software and Computer Science Education
Supervisor of the master's thesis: RNDr. Tomáš Dvořák, CSc.

Abstract: Suffix tree is a data structure which enables the performing of fast search-like operations on the text. In order to be used efficiently, it must be created quickly. In this thesis, we focus on the new kind of suffix link simulation called "minimized branching", which aims to increase the speed of suffix tree construction by reducing the number of branching operations. Our main goal is to present a comparison of the currently used methods for the suffix tree construction and to point out some advantages and disadvantages of the individual approaches. We introduce, implement and practically evaluate multiple variations of the standard McCreight's and Ukkonen's algorithms, as well as Partition and Write Only Top Down (PWOTD) algorithm, originally developed for disk-based construction. Our main result is the integrated description and implementation of these algorithms, which are both well-suited to be further built upon. We also present a simple recommendations on when it is advisable to use a particular algorithm's variation and why.

Keywords: suffix tree, construction algorithm, minimization of branching

Název práce: Konstrukce suffixového stromu s minimalizaci větvení
Autor: Peter Bašista
Katedra: Kabinet software a výuky informatiky
Vedoucí diplomové práce: RNDr. Tomáš Dvořák, CSc.

Abstrakt: Sufixový strom je datová struktura, která v textu umožňuje rychle vykonávat operace podobné vyhledávání. Aby ji bylo možné používat efektivně, musí být vytvořená rychle. V této práci se zaměříme na nový způsob simulace sufixových hran nazývaný "minimalizace větvení", který se snaží zvýšit rychlost konstrukce sufixového stromu pomocí znížení počtu větvících operací. Naš hlavní cíl je předvést porovnání současných metod pro konstrukci sufixového stromu a poukázat na některé výhody a nevýhody jednotlivých postupů. Představíme, implementujeme a prakticky posoudíme několik variant standardních algoritmů jako jsou McCreightův a Ukkonenův, stejně tak jako algoritmu PWOTD, který byl původně navržen pro diskově orientovanou konstrukci. Náším hlavním výsledkem je ucelený popis a implementace těchto algoritmů, na kterých se dá dále stavět. Také předložíme jednoduchá doporučení ohledně toho kdy je vhodné použít konkrétní algoritmus a proč.

Klíčová slova: suffixový strom, konstrukční algoritmus, minimalizace větvení

Contents

Introduction **11**

1 Overview **12**
1.1 Features of the suffix tree . . . 13
1.2 Convenience enhancements . . . 14
1.3 The sliding window . . . 16
1.4 Construction algorithms . . . 18

2 Terminology **20**
2.1 Basics . . . 20
2.2 Suffix tree . . . 26
2.3 Sliding window . . . 34

3 Construction algorithms **35**
Suffix tree representation . . . 35
Common terms and functions . . . 36
3.1 McCreight's algorithm . . . 40
3.1.1 Insertion point adjustment . . . 43
3.1.2 Suffix link simulation . . . 44
3.2 Ukkonen's algorithm . . . 47
3.2.1 Open edges . . . 49
3.3 Sliding window . . . 53
3.3.1 Edge label maintenance . . . 56
3.4 PWOTD . . . 58
3.4.1 Partitioning . . . 60
3.4.2 Evaluating the partitions . . . 67

4 Implementation details **70**
4.1 Simple linked list . . . 71
4.2 Simple hash table . . . 75
4.3 Sliding window . . . 80
4.4 Simple linear array . . . 82

5 Usage recommendations **85**

6 Benchmarks **90**
6.1 Input files . . . 91
6.2 Results . . . 94

7 Conclusions **111**

A Attached DVD **113**

Bibliography **115**

Introduction

This thesis provides a review, comparison and benchmarks of some of the most commonly used algorithms for the suffix tree construction. We analyze several variations and implementation techniques of these algorithms. Our focus is on the bottom-up suffix link simulation technique, recently published by Senft and Dvořák [SD12] and its performance in comparison with the other suffix link simulation techniques.

Two types of algorithms are examined: The construction of the entire suffix tree in the main memory and the construction using the so-called *sliding window*. Based on our analysis, we provide some conclusions on which algorithm is the most suitable for some of the typical usage scenarios. The following is a brief review of each chapter's content.

The first chapter provides a simple, nontechnical overview of this work. In the second chapter, we formally introduce the basic terminology related to the suffix trees.

The third chapter explains the reviewed algorithms for the suffix tree construction. Only the basic ideas of the examined algorithms are presented. The implementation details are the main topic of the fourth chapter. Here we address some of the issues, which arise when implementing these algorithms in practice.

The fifth chapter tries to provide a theoretical reasoning on which algorithm is the best for which purpose. This is the main result of this work. Here, we theoretically analyze the examined algorithms and try to provide some conclusions. The sixth chapter presents and analyzes the experimental results, which should support this theory. If the theory is different from the results, we explain why and either correct the theory or improve the experiments.

The last, seventh chapter gives the brief conclusion of the results from the whole work.

Chapter 1

Overview

In this initial chapter, we try to present a nontechnical overview of the discussed topics.

The most important term used in this thesis is the *suffix tree*. We can describe it as a data structure designed for storing the text and allowing fast search-like operations on it. Exact definition is presented in Chapter 2.

Gusfield [Gus97] gives an overview of some of the text operations, which can be performed particularly fast using the suffix tree. They include, but are not limited to the following:

- exact pattern matching
- finding the repetitive structures like maximal pairs, maximal repeats, super-maximal repeats or tandem repeats
- pattern matching using wildcards or regular expressions
- approximate pattern matching with a certain number of mismatches allowed
- finding the longest common substring of more strings, the longest palindrome, performing the circular string linearization or computing the matching statistics

When talking about *fast* operations, we need to clarify exactly *how fast* these operations in fact are.

Take for example the exact pattern matching problem. Using the algorithms like the ones introduced by Knuth, Morris, and Pratt [KMP77] or Boyer and Moore [BM77], we can achieve the time complexity linear with respect to the length of the text. In many applications, it might be fast enough.

But when some requirements are met, e.g. the size of the alphabet in use is constant, the suffix tree allows this operation to be performed in the time linear with respect to the length of the search *pattern*. In most cases, this pattern is much shorter than the whole text, which dramatically decreases the time complexity.

This gives the pattern matching using the suffix tree a great advantage over the previously mentioned algorithms. On the other hand, its memory requirements are rather high. To our knowledge, the most space-efficient implementation technique has been presented by Kurtz [Kur99] and it consumes up to 20 bytes per text character in the worst case. In practice, this space disadvantage is usually well balanced by the speed of most operations. As a conclusion, it might look like the pattern matching can usually be performed much faster using the suffix tree. Unfortunately, that is not entirely true.

Using a suffix tree introduces some additional time overhead. It is caused by the extra time necessary for the suffix tree *construction*. Fast and memory-efficient construction methods are therefore very important for the overall usability of the suffix tree.

1.1 Features of the suffix tree

Now we describe what the suffix tree is and what are its most important properties.

Note 1.1 *A **suffix tree** is a* tree-like *data structure, which stores* all *the suffixes of a given text and allows fast search-like operations on them.*

Being more precise, we could say it is a *rooted* tree. This just means that one of the vertices is called the *root* and has a special role. We can talk about the other vertices in terms of its *ancestors*. Consistently with the graph terminology, we call the non-root vertices of degree 1 the *leaves*. The other vertices, which are not leaves and are different from the root are usually called the *branching* vertices. Some authors prefer to use the term *node* instead of the term vertex.

This is how a simple suffix tree can look like:

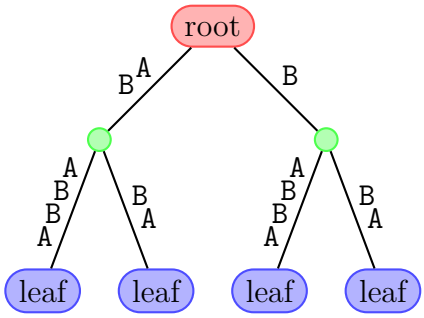


Figure 1.1: A simple suffix tree on top of the text `ABABBA`

Note that the edge labels need to be read literally from top to bottom. The initial characters of the edge label are displayed at the top of the edge while the last characters of the edge label are displayed at the bottom of the edge. We have decided to use such a labelling, because it provides more expressive rendering of the characters' positions in the edge's label.

The edges between the vertices are labelled by *substrings* of the text on top of which the suffix tree is built. Each vertex can be described by a *unique* path from the *root* to itself. The text obtained by concatenation of the edge labels on this path is a *substring* corresponding to the respective vertex. Just like we can say that the suffix tree *contains* a specific vertex, in a similar fashion we can say that the suffix tree *contains* the respective substring.

When the edge label is longer than a single character, it is usually convenient to divide such an edge using the *implicit vertices*. Their presence is only implied, which means they are *not* the actual part of the suffix tree. If a path starts at the *root*, but ends *inside* an edge, at some implicit vertex, we also say that the corresponding *substring* is *contained* in the suffix tree.

Note that the suffix tree must, by its definition, contain *all the suffixes* of the underlying text. Given the example above, that suffix tree must contain these substrings:

1. `ABABBA` 2. `BABBA` 3. `ABBA` 4. `BBA` 5. `BA` 6. `A`

But it also contains every *prefix* of these strings. The reason is that the path starting at the *root* and corresponding to any of the suffixes can be terminated sooner, at a vertex

or an implicit vertex inside an edge, which corresponds to the prefix of this suffix. And this effectively means that the suffix tree contains *every* substring of the underlying text.

The vertices which are not leaves and are different from the root are required to have *at least two* children. This is a *compactness requirement*. Consider a suffix tree, which would contain a sequence of vertices like this:

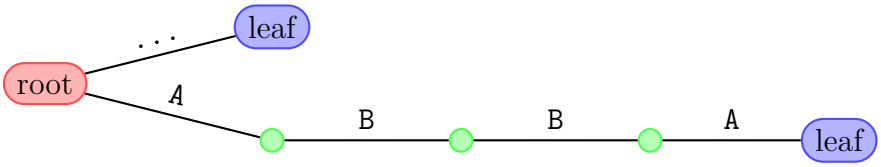


Figure 1.2: An incorrect sequence of vertices in the suffix tree containing the text `ABBA`

Such a sequence of vertices would be a waste of space, because the presence of the explicit vertices which have only one child brings no additional information. That's why it is more wise not to allow a non-root vertex to have only one child. Applying this rule, the same part of the suffix tree should look like this:

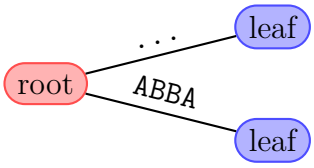


Figure 1.3: An edge in the suffix tree containing the text `ABBA`

As we can see, the path between the root and the bottom leaf has been contracted to a single edge.

On the other hand, notice that the restrictions on the number of children a vertex can have cannot apply to the *root*. When the underlying text is a sequence of a single character repeated arbitrarily many times, the root will have only one child. And if a suffix tree is *empty*, the *root* is even allowed not to have any children at all. Both of these situations are illustrated in the following figure.

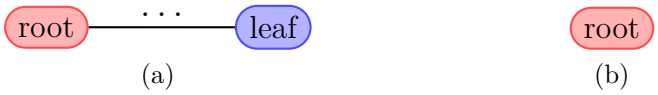


Figure 1.4: An illustration of why the *root* is allowed to have less than two children.

1.2 Convenience enhancements

It would be nice if all the suffixes of the underlying text corresponded to leaves. For some texts, the corresponding suffix trees do have this property. But our example in Figure 1.1 shows that it cannot be guaranteed to hold in general. Fortunately, there is an elegant way to modify any input text so that the suffix tree built on top of it will have this nice property.

All we have to do is pick a character, which does not occur anywhere in the text. Let us call it the *terminating* character and identify it as `$`. Then we just append this

terminating character to the text. A suffix tree built on top of such a slightly modified text ultimately gains this desired property.

If we use this technique to modify the text used in the example suffix tree above, we obtain `ABABBA$`. The suffix tree built on top of this text is shown in Figure 1.5.

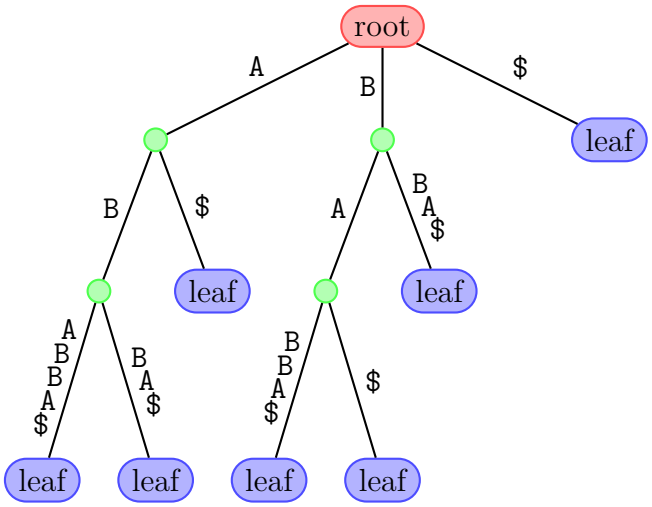


Figure 1.5: A simple suffix tree on the text `ABABBA$`

Using this simple text adjustment, we can ensure that each suffix corresponds to a leaf. Now we present a vertex numbering scheme, which takes advantage of this fact.

It is a common practice to identify each leaf with a suffix corresponding to it. We do it by giving each leaf a label containing the index of the position in the text of the first character of its corresponding suffix. All the other vertices in the suffix tree are numbered as well. The root is always given the number 1. The following, higher natural numbers are assigned to all the branching vertices as they are created.

This would, however, introduce ambiguity. The numbers assigned to the root and the branching vertices would overlap with the numbers assigned to the leaves. That is undesirable, so we need to avoid it.

The easiest way to differentiate between the numbers of leaves and the numbers of the other vertices is to change one of these number groups. Kurtz [Kur99] proposes a simple solution — make the numbers of leaves negative. And that's exactly what we do.

The suffix tree over the text `ABABBA$` with all these changes applied is shown in the Figure 1.6.

Note the numbers of the branching vertices. They seem to be assigned at random. But there is a strict rule according to which their numbers are chosen. They exactly denote the order of their creation during the suffix tree construction. The root supports this scheme as well, as it is always created at first.

Now we summarize the main properties of the suffix tree:

Note 1.2 (suffix tree properties) *The* suffix tree *built on top of the text ending with the* terminating character *has the following properties:*

1. *All the suffixes end at the leaves.*
2. *All the branching vertices have at least two children.*
3. *Each* substring *of the underlying text is contained in the suffix tree.*

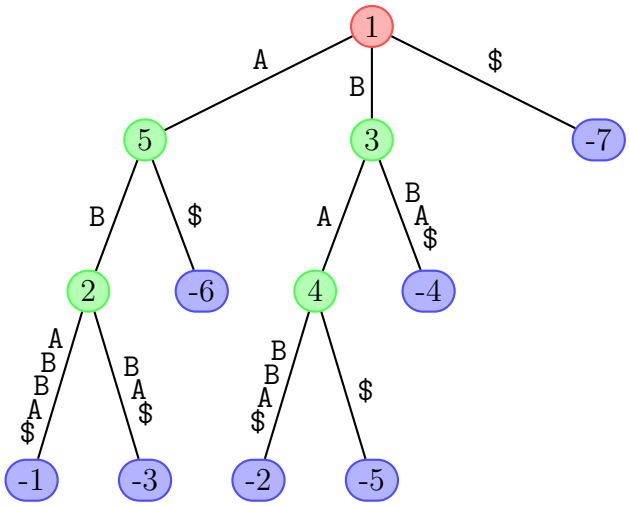
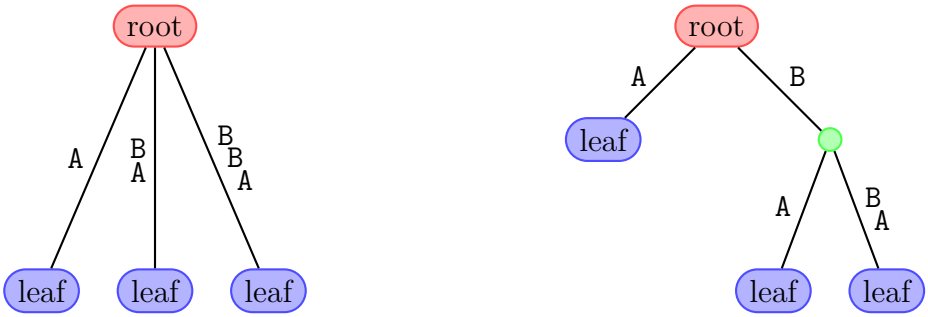


Figure 1.6: A simple suffix tree on the text `ABABBA$`

There is another interesting property of the suffix tree. Each of the labels of the edges leading to the children of any parent starts with a *different* character. This is important, because it enables faster edge selection, considering that only the first characters of the edges have to be checked.

The following figure demonstrates how would a part of the suffix tree look like, if it did not satisfy such a property.



(a) incorrect (b) correct

Figure 1.7: An illustration of the incorrect part of the suffix tree where the two edges start with the *same* character and the same part of the suffix tree, corrected, with the two edges *partially* joined.

1.3 The sliding window

Substantial part of this thesis is devoted to the analysis of the algorithms for the construction of the suffix tree over the *sliding window*. We now describe what it is and what are the benefits of its usage during the suffix tree construction.

Note 1.3 ***Sliding window*** *is a continuous range of characters in the text. Moreover, this range (not the underlying characters) can be arbitrarily moved and resized to cover any desired part of the text.*

Usually, the requirements on the sliding window also include its *maximum length*.

The sliding window is used to outline the part of the text, on top of which the suffix tree is currently built. This allows the suffix tree to be smaller and therefore consume less memory and possibly allow faster operations.

It would not make much sense to have a suffix tree built on top of a certain part of the text without the possibility to conveniently *move* the sliding window and accordingly *change* the suffix tree so that it will always contain all the suffixes of the *current* sliding window. That is exactly what the construction algorithms for the suffix tree over a sliding window must be able to achieve:

- create the suffix tree on top of the current sliding window
- provide the means for the suffix tree *maintenance* when the sliding window moves

In practice, these requirements are usually somewhat relaxed. The initial position of the sliding window is at the beginning of the text. The suffix tree on top of it can initially be created from scratch using any of the suffix tree construction algorithms.

The sliding window usually moves by one character at a time towards the end of the text. During this movement, its size remains the same. The construction algorithm must be able to quickly adjust the current suffix tree so that it becomes a valid suffix tree over the new sliding window, which has been moved by one character towards the end. And this requirement is a little bit more challenging. By far not every algorithm for the suffix tree construction can easily handle it.

Fortunately, there are approaches — most notably the extensions to the algorithm presented by Ukkonen [Ukk95] — which allow a suffix tree built over a sliding window to easily *slide* forward by a single character. That's why the name of a *sliding* window.

A very good question is: "Why do we use sliding window?". Why don't we just construct the suffix tree on top of the entire text? There can be several reasons for this.

At first, the available amount of memory might be insufficient. Some large texts, like the representation of the nucleotide bases in the human genome, contain billions of characters. A suffix tree on top of the text of such length would consume tens of gibibytes[1] of memory. But far more common reasons to use the suffix tree over a sliding window today come from the field of *data compression*.

As Senft [Sen05a] shows, a suffix tree can be used directly for the data compression. It can also be used as an auxiliary data structure allowing fast pattern matching in many common algorithms for the data compression. These algorithms usually allow compression and decompression of large files using only a small amount of memory in return for possibly worse compression ratio. That is achieved by using a smaller *window* (the currently buffered part of the input file), which of course takes less space. Consequently, the data structures like the suffix tree built over it take less space too.

[1] binary multiple *gibi* stands for 2^{30} or 1 073 741 824

1.4 Construction algorithms

Now we briefly mention a few of the most common algorithms for the suffix tree construction. We begin with the construction of the suffix tree on top of the whole text (i.e. not over a sliding window).

The first *linear* time algorithm for the suffix tree construction was introduced by Weiner [Wei73]. Assuming a constant alphabet size, its time complexity is linear with respect to the length of the underlying text. Then, McCreight [McC76] presented an improved version. Its main idea is to iteratively *insert* the suffixes into the empty suffix tree, starting from the *longest*, until the suffix tree is complete and contains all the suffixes.

Some time later, Ukkonen [Ukk95] suggested another algorithm, which uses slightly different approach. Starting from the empty suffix tree, it iteratively creates intermediate suffix trees for the all the *prefixes* of the text, starting from the *shortest*. In each iteration, it *prolongs* all the suffixes currently present in the suffix tree by one character and inserts a new suffix consisting of the new character only. By performing this, a suffix tree is changed so that it is a valid suffix tree over a new, longer, prefix of the text. When this prolonging of the suffixes is repeated sufficiently many times, the complete suffix tree is finally created.

This algorithm has one great advantage. During the construction, only the currently processed prefix of the text is required. The rest of the text can be loaded later, as the construction advances. This allows to use this algorithm for *on-line* suffix tree construction — a construction on top of a text, which is not entirely available at the moment or has unknown length. The on-line suffix tree construction algorithm is essential when constructing the suffix tree over a *sliding window*.

Another improvement came when Farach [Far97] presented an algorithm for the *truly* linear suffix tree construction. Its time complexity is linear with respect to the length of the underlying text also for the alphabet of the variable size. It is based on yet another technique, namely "divide and conquer". Unfortunately, its design and complexity make it unfavorable for the most tasks. That's probably why it is not as well-established as the previous algorithms.

A different approach to the suffix tree construction was presented by Tata, Hankins, and Patel [THP04]. It is based on an earlier algorithm called Write-Only Top-Down or WOTD, introduced by Giegerich, Kurtz, and Stoye [GKS99]. Tata, Hankins, and Patel originally developed it for the *disk-based* suffix tree construction — a construction which stores the suffix tree at least partially on disk instead of entirely in memory. But according to their experiments, it also performs well for the entirely *in-memory* construction. In this thesis, we confirm whether or not this is true.

Most recently, Senft and Dvořák [SD12] presented a couple of improvements to the traditional suffix tree construction algorithms. We also examine some of their results. In particular, we will analyze the *bottom-up* suffix link simulation and support the results which claim that using this technique can improve the time complexity a little.

We now introduce some of the approaches for the construction of the suffix tree over a sliding window. As mentioned previously, a suffix tree construction algorithm suitable for the construction over a sliding window must be able adjust a suffix tree for the new position of the sliding window *reasonably* quickly.

To the best of our knowledge, most of the algorithms, which are capable of this are nowadays based on the *on-line* algorithm presented by Ukkonen [Ukk95]. But the first-ever construction of the suffix tree over a sliding window was presented by Fiala and Greene

[FG89]. At that time, only the algorithms by Weiner [Wei73] and McCreight [McC76] had been known. Despite that, the approach introduced by Fiala and Greene is usable even in the combination with the Ukkonen's algorithm.

This approach presents a way to address two important issues, which arise when maintaining the suffix tree over a sliding window. First, when the sliding window decreases in size by one, the *second* character of the sliding window becomes its *first* character and it is necessary to delete the *longest* suffix from the suffix tree, while *keeping* all the other suffixes in it. Fiala and Greene [FG89] presented an algorithm which can do exactly that.

Second, some implementation details considering the movement of the suffix tree are addressed. The suffix tree contains some references to the underlying text which need to remain valid even when the sliding window moves. Note that we can *not* reference the text outside of the sliding window, as it would be a violation of its purpose. The details are described in subsection 3.3.1.

Another improvements to the construction of the suffix tree over a sliding window have been suggested by Larsson [Lar99]. He slightly modified the previous approach by Fiala and Greene and tried to make some clarifications regarding the correctness of these methods.

However, it was Senft [Sen05b], who finally reviewed both of these approaches and provided a correct proof to all of their parts. Fortunately, the algorithm itself has always been correct, but both Fiala and Greene as well as Larsson have underestimated the importance of supplying a *correct* proof.

In the following chapters, we analyze some of the mentioned suffix tree construction methods and provide the conclusions on which method is the most suitable for some of the most common use cases.

Chapter 2

Terminology

In this chapter, we employ a more formal language and precisely define most of the terms used later for the description of the suffix tree construction algorithms. We start with the very simple, text-related definitions, continue with suffix-tree-related definitions and end with the sliding-window-related definitions.

2.1 Basics

Our first definition explains the terms *alphabet* and *character*.

Definition 2.1 *Any* finite, nonempty *set* $\Sigma = \{c_1, c_2, \ldots, c_n\}$ *of* n *elements is called the* ***alphabet*** *of size* n*. Its members are called the* ***characters****.*

Sometimes, a character might also be referred to as *letter*. The most common alphabets considered in this thesis are:

- $\Sigma_1 = \{0, 1\}$ — binary alphabet — $|\Sigma_1| = 2$
- $\Sigma_2 = \{a, c, g, t\}$ — "dna" alphabet — $|\Sigma_2| = 4$
- $\Sigma_3 = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ — all the decimal digits — $|\Sigma_3| = 10$
- $\Sigma_4 = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$ — all the lower-case letters of English alphabet — $|\Sigma_4| = 26$

Definition 2.2 (basic properties of string) *A* ***string*** S *is a sequence of characters* $(c_1, c_2, \ldots, c_n)$*. The number of characters in this sequence is called the* ***length*** *of the string and is denoted by* $|S|$*. The string of length* 0 *is called the* ***empty*** *string and is denoted by* λ*.*

Some authors prefer to use the symbol ϵ for the empty string. Provided that $1 \leq i \leq |S|$, we can *refer* to the i^{th} character of the string S by $S[i]$.

It is usually well-understood what it means when the two strings *match*. But for clarity, we define it here as well.

Definition 2.3 (string equality) *Two strings* $A = (a_1, a_2, \ldots, a_n)$ *and* $B = (b_1, b_2, \ldots, b_m)$ ***match*** *(denoted by* $A = B$*) if their length is the same* $(n = m)$ *and* $(\forall i \in \{1, 2, \ldots, n\})(a_i = b_i)$.

Instead of saying that two strings match, we can also say that they are *equal*. When appropriate, we will follow the convention of identifying strings by *upper-case* letters and characters by *lower-case* letters.

Now we need to define some basic operations on strings.

Definition 2.4 (operations on strings) *Given a string* $S = (c_1, c_2, \ldots, c_n)$ *of length* n*, we call the string* $(c_n, c_{n-1}, \ldots, c_1)$ *its* ***reverse*** *and denote it by* S^R*. Given the strings* $A = (a_1, a_2, \ldots, a_n)$ *and* $B = (b_1, b_2, \ldots, b_m)$ *of lengths* n *and* m*, respectively, we call the string* $(a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_m)$ *of length* $n + m$ *their* ***concatenation*** *and denote it by* AB.

The terms like substring, prefix and suffix are intuitive enough, but we define them as well, just for the integrity of the definitions.

Definition 2.5 (string parts) *Suppose we have a string* $S = (c_1, c_2, \ldots, c_n)$ *of length* n.

- *A* ***substring*** S' *of string* S *is any string of the form* $(c_i, \ldots, c_j)$ *such that* $1 \leq i, j \leq n$ *and* $i \leq j$.
- *A* ***prefix*** S_p *of string* S *is any string* $(c_1, \ldots, c_i)$ *such that* $1 \leq i \leq n$.
- *A* ***suffix*** S_s *of string* S *is any string* $(c_i, \ldots, c_n)$ *such that* $1 \leq i \leq n$.

Additionally, the empty string λ *is considered to be a substring, a prefix and a suffix of* every *string.*

Simply speaking, a substring of string S is any *continuous* subsequence of the string S, its prefix is a continuous subsequence of S starting at its first character and its suffix is a continuous subsequence of S ending at its last character. In the description of the algorithms for the suffix tree construction, the following terms are often used:

Definition 2.6 (proper string parts) *Suppose we have a string* S *of length* n.

- *A* ***proper substring*** *of string* S *is any substring of* S different *from* S.
- *A* ***proper prefix*** *of string* S *is any prefix of* S different *from* S.
- *A* ***proper suffix*** *of string* S *is any suffix of* S different *from* S.

Alternatively, we can say that string parts like substring, prefix or suffix are *proper*, if their length is strictly smaller than the length of the original string.

Typically, a string can have many identical substrings. Using the following term, we can easily differentiate between them.

Definition 2.7 (string occurrence) *Suppose we have a string $T = (t_1, \ldots, t_n)$ of length n and a string $S = (s_1, \ldots, s_m)$ of length m. Consider a substring $T_s = (t_i, t_{i+1}, \ldots, t_{i+m-2}, t_{i+m-1})$ of string T. If $T_s = S$, we call the substring T_s an **occurrence** of string S inside the string T starting at its i^{th} character.*

It is a common practice to identify each string occurrence with its starting position. This convention is extensively used in the description of suffix tree implementation techniques (see Chapter 4).

For the definitions related to the suffix tree, a basic terminology from the graph theory is necessary. In order to avoid possible misunderstandings, we start to define it from the very beginning.

Definition 2.8 *A **graph** is an ordered pair $G = (V, E)$ such that V is an arbitrary set whose members are called the **vertices** and $E \subseteq \{\{u, v\} : u \neq v \text{ and } u, v \in V\}$ is a set of **edges**. An edge $e = \{u, v\}$ is said to* link *the vertices u and v. The sets of vertices and edges are sometimes denoted by $V(G)$ and $E(G)$ to make it clear that they are part of the graph G.*

This type of graph is also called *undirected*, because since its edges are just sets of two vertices, they do not have any direction. But there is another type of graph, which contains *directed* edges. They are defined as follows.

Definition 2.9 *A **directed edge** $e = (u, v)$ is an* ordered *pair of two vertices, u and v. It is important that the* directed *edge e links the vertices u and v in one direction only (e.g. form u to v, but not vice versa).*

Using this term, we can formally define a graph containing *directed* edges.

Definition 2.10 *A **directed graph** is an ordered pair $G = (V, E)$ such that V is an arbitrary set whose members are called the* vertices *and $E \subseteq \{(u, v) : u \neq v \text{ and } u, v \in V\}$ is a set of* directed *edges.*

If it is clear whether the graph in question is undirected or directed, we refer to it simply as *graph*. The following frequently used term is used to denote how many vertices are linked to a certain vertex in an undirected graph.

Definition 2.11 *A **degree** $deg_G(v)$ of a vertex v in an* undirected *graph G is the number of* edges *which are linked to it. That is, $deg_G(v) = |\{e \in E(G) : (\exists u \in V(G))(e = \{u, v\})\}|$.*

In a directed graph, this term is defined like this:

Definition 2.12 *Consider a* directed *graph G.*

- *An* ***input degree*** $deg_G^{in}(v)$ *of a vertex v in graph G is the number of* edges *which end at v. That is, $deg_G^{in}(v) = |\{e \in E(G) : (\exists u \in V(G))(e = (u, v))\}|$.*
- *An* ***output degree*** $deg_G^{out}(v)$ *of a vertex v in graph G is the number of* edges *which start at v. That is, $deg_G^{out}(v) = |\{e \in E(G) : (\exists u \in V(G))(e = (v, u))\}|$.*
- *A* ***degree*** $deg_G(v)$ *of a vertex v in graph G is the number of edges which start or end at v. That is, $deg_G(v) = deg_G^{in}(v) + deg_G^{out}(v)$.*

The following terms are used in both undirected and directed graphs.

Definition 2.13 (vertices of low degree) *A vertex of degree 0 is called an* ***isolated*** *vertex. A vertex of degree 1 is called a* ***leaf****.*

It is very important to express *when* two vertices in an undirected graph are *connected*. That's where the following definition comes in:

Definition 2.14 *In an undirected graph $G = (V, E)$, a* ***path*** $P_{v_1,v_n} = (v_1, \ldots, v_n)$ *is a sequence of n vertices, which starts at the vertex v_1, ends at the vertex v_n and the following holds:*

- *$v_i \in V(G)$ for each i such that $1 \leq i \leq n$,*
- *all the vertices are different, with the exception that the first one (v_1) might be the same as the last one (v_n),*
- *for each i such that $1 \leq i < n$ there exists an edge $e_i \in E(G)$ linking the vertices v_i and v_{i+1}, that is $e_i = \{v_i, v_{i+1}\}$.*

A path which starts and ends at the same *vertex is called a* ***cycle****.*

Using *path*, we can define the following:

Definition 2.15 *Consider an undirected graph $G = (V, E)$. If there exists a path $P_{u,v}$ between the vertices u and v in graph G, we say that these vertices are* ***connected****. Similarly, graph G is said to be* ***connected****, if there exists a path $P_{u,v}$ between any two vertices $u, v \in V(G)$.*

Now we have explained all the terms necessary for the definition of a *tree*.

Definition 2.16 *A* connected *graph is said to be a* ***tree*** *if there exists* exactly *one path between any two of its* different *vertices.*

In a *tree*, it is possible to define a parent–child relationships between the vertices. To be able to do that, we have to select one vertex, which will be the *ancestor* of all the remaining vertices. In the same manner, all the remaining vertices will be its *descendants*. This selected vertex is usually called the *root*.

Once the root is selected, we can extend the definition of ancestor to an arbitrary pair of vertices like this: A vertex u is called the ancestor of a vertex v if u is contained in a *path* $P_{r,v}$ between the root r and the vertex v.

The term ancestor is used to describe a *reflexive* relationship. It means that every vertex is considered to be an ancestor of itself. Now we can define the following:

Definition 2.17 *A **rooted tree** is any tree, which contains a vertex identified as the* root. *Its selection implicitly defines the parent–child relationships in the tree.*

Presented graphically, a rooted tree with such a relationships can look like this:

(a) *natural* style (b) *computer scientist's* style

Figure 2.1: An example of a *rooted* tree with its parent–child relationships

The *root* (the only red vertex) is the parent of both green vertices. Each green vertex has two children — the *leaves* (blue vertices) which are linked to it. The leaves' grandparent is the *root*. In general, parent–child relationships also define the ancestor–descendant relationships in the rooted tree.

In the picture, we have followed a coloring style, which we have used previously in the chapter 1. It shows the root as red, the leaves as blue and all the other vertices as green. We will use this style further in this thesis, when appropriate.

Note that Figure 2.1 presents two common ways of displaying a tree — a *natural* way and a *computer scientist's* way. For some, probably historical reasons, computer scientists tend to draw trees that grow *downwards*.

When talking about rooted trees, we should justify the common usage of the term *leaf*. Despite that according to Definition 2.13, every vertex of degree 1 is called a *leaf*, it is a usual practice *not* to apply this definition to the root. This means that even if a rooted tree contains the root of degree 1, it is *not* considered to be a leaf. Such a convention makes it easier to differentiate between the root and the leaves.

The terms defined in the following definition are used to enforce certain properties of the rooted tree, which will be used later.

Definition 2.18 (compactness requirement) *Given a rooted tree and its root, we can define the following:*

- *All the vertices which are not the* leaves *and are different from the root are called the **internal** vertices.*
- *We say that a vertex satisfies a **compactness requirement** if it has* at least *two children.*
- *We say that a rooted tree satisfies a **compactness requirement** if all its* internal *vertices satisfy the* compactness requirement.

As a consequence, rooted trees which satisfy the *compactness requirement* can *not* have arbitrarily many internal vertices. Their number is always limited by the number of leaves, as shown in the following theorem.

Theorem 2.1 (number of internal vertices) *Suppose we are given a rooted tree T which satisfies the* compactness *requirement. Also, suppose that the root satisfies this requirement as well. Let l be the number of leaves in T, excluding the root. Then the number of all the internal vertices in T and the root together is at most $l-1$.*

Proof Let n be the number of all the internal vertices in T and the root together (all the non-leaf vertices). We know that all the internal vertices in T, as well as the root, satisfy the *compactness* requirement. This means that each of these vertices must have at least two children.

Each of these children is either a leaf or it has at least one descendant which is a leaf. Hence, the presence of a vertex which satisfies the *compactness* requirement in a rooted tree increases the minimum number of leaves a rooted tree can have by at least one. Naturally, if we insert the first such vertex into the empty rooted tree containing only the root, the minimum number of its leaves increases by two. But the following insertions of such vertices increase the minimum number of leaves by one.

The presence of $n-1$ internal vertices in the rooted tree T increases the minimum number of leaves it can have by $n-1$. Since the root *also* satisfies the compactness requirement, the number of leaves is further increased by 2. Consequently, the minimum number of leaves in T is $n-1+2=n+1$. It can be expressed as an inequality $l \geq n+1$ which means that $n \leq l-1$. Therefore, the maximum number of all the internal vertices in T and the root together is $l-1$. □

Next definition introduces the edge *labels* in the tree.

Definition 2.19 *A **labeled tree** is a rooted tree whose edges are labeled by* nonempty *strings.*

If used carefully, edge labeling can be very useful. By introducing some additional constraints on the labeled tree, we can *enforce* the appropriate usage of the edge labels. Such a constraint is presented in the following definition.

Definition 2.20 (branching requirement) *Suppose we are given a labeled tree $T=(V,E)$. Let $v \in V(T)$ be an arbitrary vertex in T.*

- *We say that the vertex v satisfies a* branching *requirement, if all the* labels *of the edges leading to its children start with a* different *character.*
- *Similarly, tree T is said to satisfy a **branching requirement**, if* all *its vertices satisfy the* branching *requirement.*

Now we describe a tree, which meets the requirements we have just defined.

Definition 2.21 *A **string tree** is a* labeled tree *with the following properties:*

- *It satisfies the* compactness *requirement.*
- *It satisfies the* branching *requirement.*

In order for a *string tree* to be useful, it is necessary to allow some operations on it. Probably the most important task that can be done with a string tree is the check for an *occurrence* of a given string.

Definition 2.22 (string occurrence in a string tree) *Suppose we are given a string tree T and its root r. We say that a string S is **contained** in a string tree T if the following holds:*

- *There exists a path $P_{r,v}$ starting at the* root *r and ending at some vertex v,*
- *the* concatenation *of the edge labels on the path $P_{r,v}$ is a string S_v,*
- *string S is a* prefix *of string S_v.*

Note that if a string tree contains the string S, it also contains, by definition, all the *prefixes* of the string S. Formally, we can also say that the *empty string* is *contained* in every string tree.

2.2 Suffix tree

In this section, we finally define the long-awaited *suffix tree*.

Definition 2.23 *A **suffix tree** over the text T (or on top of the text T) is a* string tree *which contains* all *the suffixes of the text T. It is* not *allowed to contain any other string.*

A simple consequence following directly from the definition is that a suffix tree over the text T also contains all the *prefixes* of all the *suffixes* of the text T. And this effectively means that it contains all the *substrings* of the text T.

If $T = \lambda$ is the *empty* string, a suffix tree over T contains only one vertex — the *root*. Such a suffix tree is sometimes referred to as the *empty* suffix tree.

We can now define some suffix-tree-related terms, which are used in the description of the algorithms for the suffix tree construction.

Definition 2.24 (vertex–string correspondence) *Suppose we have a* string tree *T with the* root *r and a vertex $v \in V(T)$. Let S_v be the string formed by* concatenation *of the edge labels on the path $P_{r,v}$ from the* root *r to the vertex v. We say that the vertex v **corresponds** to string S_v.*

A special case is the root*, which* always *corresponds to the* empty string.

Note that if a vertex v *corresponds* to a string S in a string tree T, then the string S is *contained* in the string tree T. But it does not work the other way around. If a string

S is *contained* in a string tree T, then it does *not* necessarily mean that there is a vertex $v \in V(T)$, which *corresponds* to the string S. It might exist, but it definitely doesn't have to. There is an example in Figure 2.2 showing why.

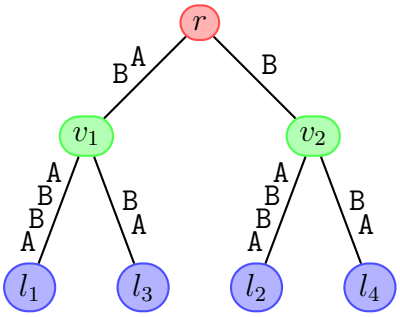


Figure 2.2: A simple string tree T. Note that the string `ABB` is *contained* in T, but there is *no* vertex which *corresponds* to it.

As we have mentioned in Chapter 1, the edge labels need to be read in the direction of the tree's growth. Knowing that, we can confirm that the string tree T from the Figure 2.2 contains the string `ABB`, because it is a prefix of the string `ABBA` corresponding to the leaf l_2. But if you check every string corresponding to any vertex in T, you will find out that *none* of them matches the string `ABB`.

This is perfectly correct. To be able to identify *all* the strings contained in the string tree with some "position" in it, we have to introduce the following term:

Definition 2.25 (implicit vertex) *Suppose we are given a* labeled *tree $T = (V, E)$ and its edge $e = \{u, v\} \in E(T)$ labeled by a string S of length $n > 1$. Suppose u is a parent of v. An* ***implicit vertex*** *$i = (e, k)$ inside the edge e at the position k such that $1 \leq k \leq n - 1$ is an ordered pair of the edge e and the position k. It denotes a position in the label of the edge e right between its k^{th} and $k + 1^{st}$ character in the direction from u to v.*

A vertex in a labeled tree which is not implicit is usually called an *explicit* vertex. In general, when using the term vertex with no attribute, we usually mean any vertex, either explicit or implicit.

In order to better illustrate the relation between an implicit vertex and a position inside an edge, suppose we have divided the edge e from the definition using exactly $n-1$ vertices $v_1 \dots v_n$ such that the path from u to v is $P_{u,v} = (u, v_1, \dots, v_n, v)$. The label of the edge e is divided into the individual characters which are then used for labeling the newly created edges, such that the concatenation of the edge labels on the path $P_{u,v}$ is equal to the string S, i.e. the original label of the edge e. The implicit vertex $i = (e, k)$ then represents the same position inside the edge e as the supposed vertex v_k.

Each edge $e \in E(T)$ labeled by a string S of length $n > 1$ *contains* $n - 1$ *implicit* vertices. They are positioned "inside" the edge e, between the characters of its *label*. Implicit vertices are only *imaginary* — they are not part of the labeled tree. It is just an abstraction, which enables us to divide any edge with a label of length greater than one into parts corresponding to the label characters. Moreover, if $e = \{u, v\}$ is an edge containing the implicit vertices and u is a *parent* of v, then it is common to consider the vertex u to be the parent of *all* the implicit vertices inside the edge e. This is probably

easier to understand from the picture:

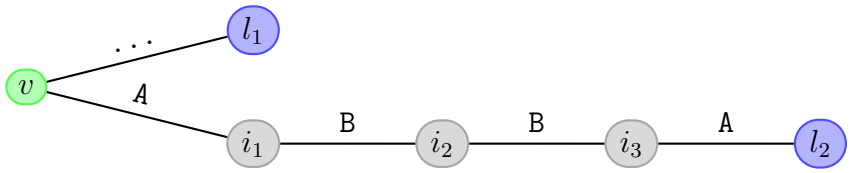


Figure 2.3: A part of a *labeled* tree illustrating its *implicit* vertices (in gray). The label of the edge between the vertices v and l_2 is `ABBA`.

The vertex v is considered to be the parent of all the displayed implicit vertices.

In a string tree, *every* implicit vertex corresponds to some string. This correspondence is defined in a slightly different way than the correspondence of *explicit* vertices to strings. But it is very similar and pretty straightforward:

Definition 2.26 (implicit vertex–string correspondence) *Suppose we have a* string tree T *and its implicit vertex* $i = (e, k)$ *positioned inside an edge* $e = (u, v) \in E(T)$ *labeled by a string* S*. Also, suppose that* u *is a* parent *of* v*. Let* S_u *be a string corresponding to the vertex* u*. Let* P_k *be a prefix of length* k *of the string* S*. We say that the implicit vertex* i ***corresponds*** *to string* $S_i = S_u P_k$ *formed by* concatenation *of the strings* S_u *and* P_k*.*

The implicit vertices are somewhat special. The *compactness* requirement for the string tree does not apply to them. Yet, they are useful for the correspondence with strings *contained* in a string tree. They allow *every* string contained in a string tree to have its corresponding vertex, either explicit or implicit.

This also applies to a *suffix* tree. Provided that it contains all the suffixes of a certain text T, it therefore must also contain a vertex, either explicit or implicit, which corresponds to each suffix of the text T.

The longest suffix of the text T — the T itself — *must* correspond to a *leaf* in the suffix tree. In fact, *all* the leaves in the suffix tree correspond to a *suffix* of the text T:

Theorem 2.2 (leaf–suffix correspondence) *Given a suffix tree* ST *over the text* T*, each of its* leaves *corresponds to a* suffix *of the text* T*.*

Proof Consider an arbitrary leaf l of the suffix tree ST which corresponds to a string S_l. We know that ST satisfies a *branching* requirement. We also know that l does not have *any* children. This implies that ST *cannot* contain any string S longer than S_l such that S_l is a prefix of S.

In other words, there can be no string S contained in the suffix tree ST which *extends* the string S_l. Since ST must *not* contain any other strings than the *substrings* of the text T, we can conclude that the string S_l is indeed a *suffix* of the text T. □

So, all the leaves in a suffix tree over the text T correspond to some *suffix* of the text T. But *not every* suffix of the text T corresponds to a leaf in the respective suffix tree, as shown in the following picture:

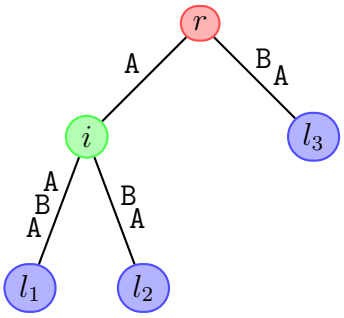


Figure 2.4: A simple suffix tree T over the text AABA. Note that the suffix A is *contained* in T and corresponds to the vertex i which is *not* a leaf.

There is another important property of the leaves in the suffix tree:

Theorem 2.3 (leaf property) *Suppose we are given a suffix tree ST over the text T and a suffix S_a corresponding to a leaf a in ST. Then one of the following holds:*

- *Either $S_a = T$ is the* longest *suffix present in the suffix tree,*
- *or $|S_a| < |T|$ and there exists a* suffix *S_b of the text T corresponding to a* leaf *$b \neq a$ such that $|S_b| = |S_a| + 1$ and S_a is a* suffix *of S_b.*

Proof If $S_a = T$, there is nothing to prove. So, let's suppose that $|S_a| < |T|$. Then S_a is *not* the longest suffix present in the suffix tree ST. This means that there exists at least one suffix of T, which is *longer* than the suffix S_a and is *contained* in ST.

Consider a suffix S_b, which is the *shortest* of these suffixes. The suffix tree ST must contain *all* the suffixes of the text T, which means that $|S_b| = |S_a| + 1$. Since S_a and S_b are both suffixes of T and $|S_a| < |S_b|$, the string S_a must be a *suffix* of S_b. Let the vertex, either explicit or implicit, which *corresponds* to the string S_b be called b. Knowing that $|S_b| \neq |S_a|$, we can be sure that $b \neq a$. The only question remaining is, whether b is indeed a *leaf*.

Since the suffix S_a corresponds to a *leaf* a, it means that there is *no* string S *contained* in the suffix tree ST such that S_a is a prefix of S. If b were not a leaf, it would be an internal vertex, an implicit vertex or the root. However, it could *not* be the root, because the length of S_b is *strictly* positive. If it were either an implicit vertex or an internal vertex, then ST would contain a *suffix* S_c, longer than S_b, such that S_b is a *prefix* of S_c. Now consider the string S_d, which is a *suffix* of S_c, whose length is $|S_c| - 1$. This string is also a *suffix* of the text T, which means that it is *contained* in the suffix tree ST. According to its definition, $|S_d| > |S_a|$ while S_a is a *prefix* of S_d. And this is a *contradiction* with the fact that the string S_a corresponds to a *leaf* a. This all implies that b has to be a *leaf*. □

A direct consequence of this theorem is that all the *leaves* contained in a suffix tree on top of any text T *correspond* to all the *longest* suffixes of the text T, until some length n. Suffixes shorter than n *correspond* to another vertices, either explicit or implicit, of the suffix tree, but they *cannot* correspond to a *leaf*.

Length n can be equal to the length of the text, which means that only the *longest* suffix — the T itself — corresponds to a *leaf* in the suffix tree and all the remaining

suffixes correspond to different kinds of vertices. On the other hand, n can also be 1, which means that *all* the suffixes correspond to *leaves* in the suffix tree. This is often desirable, because it simplifies the suffix tree *construction* algorithms.

When talking about the leaves of the suffix tree and the suffixes, which *correspond* to them the following naming convention might be helpful.

Definition 2.27 (leaf numbering) *Consider a suffix tree over the text T of length n. We use S_i to denote a suffix of the text T starting at its i^{th} character, implying that $|S_i| = n - i + 1$. Similarly, we use l_i to denote a vertex* corresponding to *the suffix S_i, provided that it is a* leaf.

Many algorithms for the suffix tree construction enhance the standard suffix tree with some additional structures. These enhancements are mainly used to optimize the construction speed. One of them — probably the most widely used — is defined like this:

Definition 2.28 (suffix links) *Suppose we are given a suffix tree $ST = (V, E)$ along with its root $r \in V(ST)$. Let $u \in V(ST)$ be an arbitrary vertex in ST* different *from the root and let S_u be a string corresponding to it. Consider a string S_v which is a* suffix *of S_u whose length is $|S_u| - 1$. If ST contains a vertex v corresponding to the string S_v, we can define a* ***suffix link*** *of the vertex u to be a* directed *edge which starts at the vertex u and ends at the vertex v.*

The vertex at which the suffix link starts is often called its *source* vertex. Likewise, the vertex at which the suffix link ends is often called its *target* vertex.

For the *root*, we do not define its suffix link, because the string corresponding to it is always the *empty string*. As there is no *shorter* string than the empty string, defining a suffix link in this case would make little sense. Similarly, a suffix link *cannot* be defined if its supposed target vertex does not exist. This *can* happen, because not all the strings contained in a suffix tree correspond to explicit vertices.

On the contrary, suffix links for the *leaves* are well defined. The following theorem shows that *almost* all of the suffix links starting at a *leaf* in a suffix tree also end at a *leaf*.

Theorem 2.4 (suffix links for leaves) *Let ST be a suffix tree over the text T of length n. Consider all the* leaves *$l_1, l_2, \ldots, l_k$ of ST such that the suffixes* corresponding *to them are $S_1, S_2, \ldots, S_k$, respectively. These leaves and suffixes are named according to the naming convention introduced in Definition 2.27. Then a suffix link starting at l_i ends at l_{i+1} for each $i < k$.*

Proof It is obvious, because the length of the suffix S_i is $n-i+1$ and *all* the suffixes in the *range* $S_1 \ldots S_k$ are present. □

According to this theorem, the *only* leaf whose suffix link does not *end* at a leaf as well is the one which corresponds to the shortest of the suffixes corresponding to the leaves. In general, a suffix link of such a leaf might not even exist, as shown in Figure 2.5. But nevertheless, it all makes the suffix links starting at leaves easy to figure out.

Considering the suffix links starting at the internal vertices, they *always* exist and each of them ends either at another internal vertex or at the root. The following theorem explains why.

Theorem 2.5 (suffix links for internal vertices) *Suppose we are given a suffix tree ST over the text T of length n. Let u be an arbitrary internal vertex of ST and let the string corresponding to it be called S_u. Consider a string S_v which is a suffix of S_u such that $|S_v| = |S_u| - 1$. Then the vertex v corresponding to the string S_v is* explicit *in ST and it is the* target *vertex of a suffix link starting at the vertex u.*

Proof According to the suffix link's definition, if the suffix link starting at the vertex u *exists*, its target vertex has to be the vertex v. The only relevant question is whether the vertex v is indeed *explicit*.

The branching requirement (Definition 2.20) applied to the the internal vertex u states that it has to have at least two children. This means that there are at least two suffixes of the text T which are strictly longer than the string S_u and whose longest common prefix is the string S_u. Let these suffixes be called S_1 and S_2. Consider a prefix P_1 of the suffix S_1 and a prefix P_2 of the suffix S_2 such that $|P_1| = |S_1| - 1$ and $|P_2| = |S_2| - 1$. The longest common prefix of the strings P_1 and P_2 is the string S_v. Hence, the vertex v has to be explicit as well. □

In practice, it is often necessary to consider also the suffix links which could start or end at the implicit vertices. We have already pointed out that the target vertex of a suffix link starting at the leaf which corresponds to the shortest suffix of the suffixes corresponding to the leaves does not necessarily have to be an explicit vertex. In this case, it is not be possible to define a standard suffix link of such a leaf, which might be inconvenient. For that reason, the following definition introduces a new type of suffix links, which are allowed to start or end at the implicit vertices.

Definition 2.29 (implicit suffix links) *Suppose we are given a suffix tree ST. Let $i_1 = (e_1, k_1)$ be an arbitrary implicit vertex in ST and let S_1 be a string corresponding to it. Consider a string S_u which is a* suffix *of S_1 whose length is $|S_1| - 1$ and let the vertex, either explicit or implicit, which corresponds to it be called u. An* ***implicit suffix link*** *of the vertex i_1 is an ordered pair (i_1, u) of its source vertex i_1 and its target vertex u.*

Similarly, let v be an arbitrary explicit vertex in ST and let S_v be a string corresponding to it. Consider a string S_2 which is a suffix *of S_v whose length is $|S_v| - 1$ and let the vertex, either explicit or implicit, which corresponds to it be called i_2. If the vertex i_2 is* implicit*, we can define an* ***implicit suffix link*** *of the vertex v to be an ordered pair (v, i_2) of its source vertex v and its target vertex i_2.*

In the pictures, the *suffix links* starting or ending at the *implicit* vertices are usually not drawn. The reason is that it would often mean to draw so many suffix links that the picture would become much less comprehensible.

But fortunately, suffix links for the implicit vertices inside an edge are also easy to figure out, provided that the suffix links of the vertices which are linked by this edge

are known. In order to explain how can such a suffix links be determined, the following definition is necessary.

Definition 2.30 *Consider a string tree $T = (V, E)$ with its root $r \in V(T)$ and let $e = \{u, v\} \in E(T)$ be any of its edges such that u is a parent of v. Suppose that e is labeled by a string S of length $n > 1$. Let $i = (e, k)$ be an arbitrary implicit vertex inside the edge e such that $1 \leq k \leq n - 1$. Consider any explicit vertex p on the path from the root r to the vertex u. A* ***generalized path*** *starting at the explicit vertex p and ending at the implicit vertex i is an ordered pair $P_{p,i} = (P_{p,u}, i)$ containing the path from the vertex p to the vertex u and the implicit vertex i. A* generalized path *which starts at the explicit vertex a and ends at the explicit vertex b in a string tree T is equal to the path $P_{a,b}$ defined in Definition 2.14.*

Simply speaking, a *generalized* path is allowed to end at an implicit vertex. The following theorem takes advantage of this enhanced definition and explains how the suffix links starting at the implicit vertices can easily be inferred.

Theorem 2.6 (implicit suffix links' inference) *Consider a suffix tree ST built on top of a text T. Suppose we are given an edge $e = \{a, b\}$ in which a is the* parent *of b. Let the edge e be labeled by a string S_e, whose length is* greater *than 1 and let the string corresponding to the vertex a be called S_a. Also, let c be the target vertex of a suffix link starting at the vertex a. If such a suffix link does not exist because a is the root, then let c be the root as well. Let the string* corresponding *to the vertex c be called S_c. Finally, let the target vertex, either explicit or implicit, of the possibly implicit suffix link starting at the vertex b be called d.*

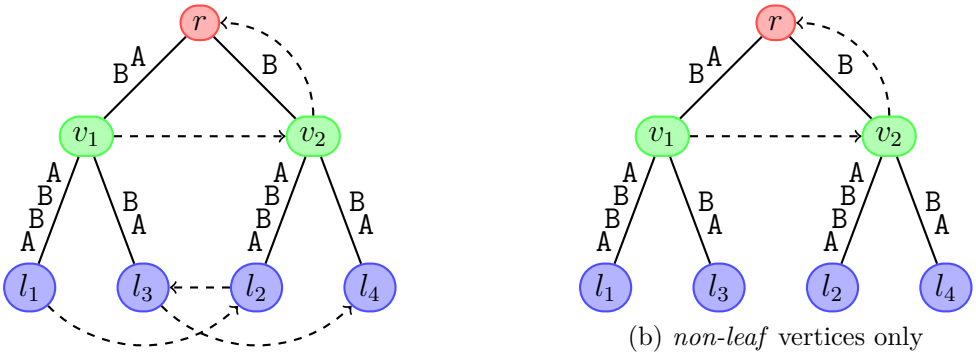
Consider any implicit vertex $i = (e, k)$ inside the edge e whose position is just after the k^{th} character of the string S_e while $1 \leq k \leq |S_e| - 1$. Let the prefix of length k of the string S_e be called P_k and let the string corresponding to the vertex i be called $S_i = S_a P_k$. Then the target vertex of the implicit suffix link starting at the implicit vertex i is defined like this:

It is the vertex v, either explicit or implicit, which corresponds to the string $S_v = S_c P_k$ and whose position is just after the k^{th} character on the generalized path from the vertex c to the vertex d.

Proof According to the *branching* requirement, a string tree *cannot* contain two *different* vertices, which correspond to the *same* substring. This implies that there is *exactly* one vertex v, either explicit or implicit, corresponding to the string S_v in ST.

As shown in Theorem 2.5, if the suffix link starting at the vertex a exists, it has to end at the internal vertex. This means that in any case, the vertex c must be explicit. Since S_c is a suffix of S_a and $|S_c| = |S_a| - 1$, it is clear that the string S_c *must* be a *prefix* of S_v. Thanks to the *branching* requirement, it means that the vertex c must be included in the *generalized path* from the root r to the vertex v, which might be either explicit or implicit. And this ultimately implies that $S_v = S_c P_k$. Since $|P_k| = k$ and the vertex c is explicit, we can shorten such a generalized path to start at the vertex c. Consequently, the vertex v must be positioned right after the k^{th} character of this shortened path. □

Considering this result, the *implicit* suffix links are usually not displayed in the pictures of the suffix trees. But, of course, the suffix links between the explicit vertices are displayed, when necessary. In the following figure, we present two simple drawing styles, which display the suffix links between the *explicit* vertices:



(a) all

(b) *non-leaf* vertices only

Figure 2.5: An example of a suffix tree with *suffix links* (dashed)

Note that the suffix link starting at the leaf l_4 ends at the *implicit* vertex just after the character `A` of the edge between the vertices v_2 and l_2. That's why it is not drawn.

Since it is obvious where the suffix link starting at *almost* any leaf ends, it is quite well established not to draw such suffix links at all. The most common practice is to draw the suffix links between the *non-leaf*, *explicit* vertices only.

Now we would like to mention one *enhancement*, which is often performed on the suffix tree in order to make it easier to use. It enforces that *all* the suffixes correspond to *leaves* in every such suffix tree. In addition, as there cannot be any *leaf* which does not correspond to a *suffix*, this modification introduces *one-to-one* leaf–suffix correspondence, which is very beneficial during the suffix tree construction.

Definition 2.31 *Given a text T, any character which does not occur anywhere in T is called its* ***terminating character*** *and is usually denoted by* `$`*. If we append the terminating character to the text T, we obtain a* ***terminated text*** *T\$.*

The following theorem explains why a suffix tree built on top of any such text has the previously mentioned properties.

Theorem 2.7 (suffix tree on top of T\$) *Suppose we have a suffix tree ST built on top of a* terminated *text T\$. Then* each *of its suffixes corresponds to a* leaf *in ST.*

Proof Since the last character of the text T\$ does *not* occur anywhere else in T\$, *none* of its suffixes can contain it at a position different than the *last*. This immediately implies that *none* of the suffixes can correspond to a *non-leaf* vertex. Consequently, all the suffixes *must* correspond to *leaves*. □

Here is an example of the suffix tree built over the text whose last character is the *terminating character*.

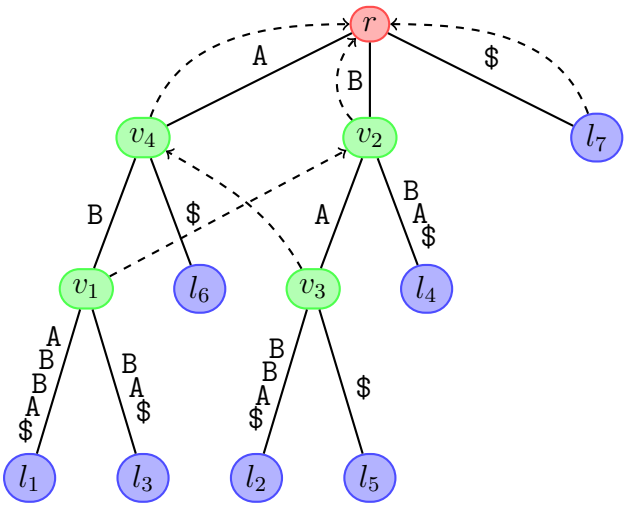


Figure 2.6: A simple suffix tree over the text `ABABBA$`. Note that all the suffixes end at leaves.

Introducing a terminating character brings one more advantage. The suffix link starting at the leaf $l_{|T\$|}$, corresponding to a *suffix* of length 1, always ends at the *root*. So, all the suffix links leading from the *leaves* are easily figured out. They can be handled implicitly — without the need for the actual suffix links.

By now, most of the basic terms related to the *suffix tree* have already been defined. However, the construction algorithms, which will be introduced in Chapter 3, will require some additional definitions. But because they are usually algorithm-specific, we will define them later, when necessary.

2.3 Sliding window

Here, we define the *sliding window*, which can be used by some of the suffix tree construction algorithms. Its *purpose* is to allow the suffix tree to be *constructed* only on top of a part of the text — called the *sliding window*. When the sliding window moves, the suffix tree is continuously adjusted instead of built from scratch, which considerably reduces the time complexity.

Definition 2.32 (sliding window) *Suppose we are given a text T of length n. Let $1 \leq b \leq f \leq n$ be the indices to the text T. A **sliding window** W of length m over the text T is an ordered pair $W = (b, f)$. Its length is $m = f - b + 1$.*

The first or the leftmost *character of the sliding window is called its **back**, while the last or the* rightmost *character of the sliding window is called its **front**.*

In other words, sliding window refers to a *substring* of the text T starting at its b^{th} character and ending at its f_{th} character. The rest will be defined later, when necessary.

Chapter 3

Construction algorithms

In this chapter, we present a *comprehensive* review of the suffix tree construction algorithms, which are analyzed in this thesis. In particular, we describe the conventional algorithms presented by McCreight [McC76] and Ukkonen [Ukk95], as well as the algorithm presented by Tata, Hankins, and Patel [THP04]. We characterize all the examined variants of these algorithms.

However, we will not dive too much into the implementation details. The main point of this chapter is to give a basic understanding of each algorithm, present its main idea and describe it using a higher level of abstraction.

Suffix tree representation

Generally, a suffix tree is represented by a structure containing the set of all its vertices and the set of all its edges. Moreover, it must also contain a reference to its root. We use this simple abstraction to represent a suffix tree in this chapter. However, no details on how the set of vertices or the set of edges can be represented are provided. Also, the means of effectively accessing all the children of a certain vertex are not addressed here. All of these implementation details are described in Chapter 4.

On the other hand, we ought to make some concepts clear right now. Most importantly, the representation of vertices and edges needs to be described in more detail.

Each vertex is represented by a structure containing the following two fields: its *depth* in the suffix tree and a *pointer* to the target vertex of a suffix link starting at this particular vertex. The depth of a vertex in the suffix tree is equal to the length of the string corresponding to it. Additionally, some algorithm variations require that the structure representing a single vertex also contains a pointer to its parent.

In practice, not all of these fields are present explicitly for every explicit vertex. For example, the target vertex of a suffix link starting at almost every leaf can be easily inferred when the appropriate leaf numbering is used, as mentioned in Theorem 2.4. In addition, a suffix link starting at the leaf which corresponds to the shortest of the suffixes corresponding to the leaves is never used by any of the suffix tree construction algorithms presented in this thesis. Similarly, the depth of a leaf can be inferred from the length of its corresponding suffix. This means that both of these fields can be omitted for all the leaves. As a result, the structure representing the leaves is often different from the structure representing the other vertices, which are usually called the *branching vertices*.

All the algorithms analyzed in this thesis use the same method for representing the edge labels. In particular, every edge label is represented by a pair of indices to the underlying

text over which the suffix tree is constructed. These indices represent the beginning and the end of a text substring, which matches the particular edge's label.

Such a representation of the edge labels is space-efficient and it allows the algorithms to achieve the desired space complexity. Despite the fact that it is an implementation detail, we decided to introduce it here because the presented algorithms *do* take advantage of the edge labels' design and therefore we have to include it in the algorithms' descriptions.

Common terms and functions

Both McCreight's and Ukkonen's algorithms for the suffix tree construction use the *active point*. It is just another name for a vertex, either explicit or implicit, which corresponds to the longest suffix currently present in the partially constructed suffix tree, which does not correspond to a *leaf*. During the suffix tree construction, a partially constructed suffix tree refers to a string tree, which contains only *some* of the substrings and suffixes of the current text while at the same time it does not contain any other string. The following definition explains the term active point more precisely.

Definition 3.1 *Suppose we are given a* string *tree ST containing exclusively the substrings of the text T. Let S_i be the* longest *suffix of T, which is already contained in the string tree ST and which does* not *correspond to a* leaf*. Let a be an internal vertex, either explicit or implicit, which corresponds to the suffix S_i. Then the vertex a is called the* ***active point*** *of the string tree ST.*

As a consequence, if *every* nonempty suffix currently contained in a string tree corresponds to a leaf, then its *active point* is the *root*. The reason is that the root always corresponds to the empty string and it is never considered to be a leaf.

The active point is used to speed-up the construction. The McCreight's and Ukkonen's algorithms use it to mark a location in the suffix tree from which the next iteration can safely start. It is therefore not required to always start from the *root*. This simple enhancement significantly reduces the time complexity.

Our implementation contains also modified versions of these algorithms which do not use the active point. It is therefore possible to see this difference in practice.

Some steps carried out by McCreight's and Ukkonen's algorithms are identical. Most importantly, the operations of scanning an edge, branching, following a suffix link, splitting an edge or creating a new leaf are common for both of these algorithms and can be performed using the same functions. Now we describe all of them.

The first of these functions is used to perform so-called *scanning* of an edge. It is an operation, which simply matches all the characters of the specified edge to the desired substring of the text. Afterwards, the number of matching characters and the overall scanning result are returned.

A pseudocode of every function outlined in this thesis contains a header which describes its parameters and its return value. There are two kinds of function's parameters. If a parameter's value is read *prior* to being overwritten, it is considered to be an *input parameter*. Similarly, if a parameter's value *can* change upon returning from the function, it is considered to be an *output parameter*. It is possible for a parameter to be an input and an output parameter at the same time. The function's header contains the list of all its input and output parameters.

A function performing the scanning operation can be described like this.

```
function Scan (ST, T, n, k, p, c, d)

Input:   ST — string tree
         T — entire text
         n — length of the text T
         k — index of the first character of the text substring to be matched
         p — parent vertex of an edge to be scanned
         c — child vertex of an edge to be scanned
Output:  d — number of matching characters
Returns: scanning result

i ← index of the first character of the p → c edge's label;
ii ← i; /* remembering the initial value of the variable i */
m ← index of the last character of the p → c edge's label;
while ((T[i] == T[k]) and (k ≤ n) and (i ≤ m)) do
        i ← i + 1; k ← k + 1; /* incrementing both indices */
end
d ← i - ii; /* number of checked and matching characters */
if (i > m) do
        return complete match; /* entire edge's label matches */
else if (k > n) do /* and i ≤ m */
        return partial match (long edge); /* text substring is shorter than the edge's label */
else /* T[i] ≠ T[k] and k ≤ n and i ≤ m */
        return partial match (mismatch); /* edge's label contains a mismatching character */
end
```

The following function is used to perform the *branching* operation, which consists of checking all the edges starting at the specified vertex for a *possible* match. A "possibly matching" edge is an edge whose label starts with the first character of the text substring to be matched. Due to the branching requirement (Definition 2.20), there can be at most one "possibly matching" edge. The branching operation simply checks the first characters of the labels of all the edges starting at the provided vertex and looks for the matching one. If it is found, the target vertex of the corresponding edge is returned. Otherwise, *null* is returned.

```
function Branch (ST, T, k, p, c)

Input:   ST — partially constructed suffix tree
         T — entire text
         k — index of the first character of the text substring to be matched
         p — parent vertex of the edges to be scanned
Output:  c — the selected child of the provided vertex p
Returns: branching result

for (each child c of the vertex p) do
        if (the first character of the edge p → c is equal to T[k]) do
                return success; /* the "possibly matching" edge has been found */
        end
end
c ← null; /* resetting the c variable */
return failure; /* there is no "possibly matching" edge */
```

This function can be implemented in many ways depending on the selected implementation technique. The version presented here corresponds to the implementation technique which uses the linked lists for providing the access to the children of any particular vertex. A notable example of such an implementation technique is SLLI (see section 4.1 for details). It is also possible to use some other implementation technique, which provides a *direct* access to all the children of any particular vertex. A hashing-based implementation technique of this kind called SHTI is introduced in section 4.2. If such an implementation technique is used, the pseudocode of this function, as well as its time complexity change.

Another common function is used to *follow* a suffix link. Usually, it is important to use it instead of using a suffix link of the provided vertex directly, because it might not exist. The reason is that the provided vertex might be the *root*, whose suffix link cannot be defined. In this case, this function returns the *root* itself (denoted by `ST.root` in pseudocode), because the algorithms which are using it require this behavior.

```
function SuffixLink (ST, v)

Input:   ST — partially constructed suffix tree
         v — any explicit vertex in ST
Returns: target vertex of a suffix link starting at the provided vertex v

if (v == ST.root) then
        return ST.root; /* the root does not have a suffix link */
end
return v.suffix_link; /* success */
```

Next function is used to split the provided edge at the specified position by creating an explicit vertex inside it. At first, it simply creates a new explicit internal vertex and deletes the original edge. Then it links the newly created vertex to the vertices of the previously deleted edge and sets up the edge labels appropriately. If necessary, this function also creates a new suffix link starting at the provided *source* vertex and ending at the newly created *target* vertex. Finally, the parent vertex of the provided edge is adjusted to this newly created vertex and the function returns successfully. The pseudocode of this function is now presented. Note that the term `s.suffix_link` refers to the target vertex of a suffix link starting at the vertex `s`.

```
function SplitEdge (ST, T, p, c, d, s)

Input:   ST — partially constructed suffix tree
         T — entire text
         p — parent vertex of the edge to be split
         c — child vertex of edge to be split
         d — position at which the provided edge will be split
         s — source vertex of the suffix link to be created, if necessary
Output:  ST — partially constructed suffix tree containing the new explicit internal vertex
         p — newly created vertex
         s — cleared as soon as the suffix link is set
Returns: splitting result

v ← new vertex; /* creating a new explicit vertex in ST */
delete the p → c edge in ST;
create the p → v and v → c edges in ST and set up their edge labels according to d;
if (s ≠ null) then
        s.suffix_link ← v; /* setting the suffix link starting at s */
        s ← null; /* clearing the s variable */
end
p ← v; /* adjusting the variable p to the newly created vertex */
return success;
```

Note that this function can only return one value — *success*. Therefore, in some programming languages it would be called a *procedure*. But the pseudocode presented here omits some tests, which would have to be performed in the real implementation. For example, we would have to check whether it is possible to allocate memory for the new vertex v. In case of allocation failure, we would have to return failure as well. This is why we use the term function for all the functions presented in this thesis, regardless of the number of their possible return values.

The last of the common functions is used to create a new leaf vertex at the specified position in the partially constructed suffix tree. It simply creates a new leaf and a new edge, which is then set to link the provided parent vertex and the newly created leaf vertex. Afterwards, the label of this edge is set up appropriately.

```
function CreateLeaf (ST, T, i, p)

Input:   ST — partially constructed suffix tree
         T — entire text
         i — starting position of the substring corresponding to the leaf to be created
         p — future parent of the new leaf to be created
Output:  ST — partially constructed suffix tree containing the newly created leaf
Returns: creation result

v ← new leaf; /* creating a new leaf in ST */
create the p → v edge in ST and set up its edge label according to i;
return success;
```

Having presented these common functions, we can now describe the McCreight's and Ukkonen's algorithms.

3.1 McCreight's algorithm

The first algorithm we are going to describe is the one presented by McCreight [McC76]. As we have mentioned previously in Chapter 1, its main idea is to iteratively *insert* the suffixes into the empty suffix tree, starting from the *longest*, until the entire suffix tree is complete and contains all the suffixes.

The construction starts from the *empty* suffix tree, which contains only one vertex — the *root*. A partially constructed suffix tree obtained after the insertion of each but the last suffix is not guaranteed to have all the properties of a complete suffix tree.

In order to speed up the construction, this algorithm uses the *active point*. However, it is not used *directly*. Instead, another term closely related to the active point is used. Now we present its definition.

Definition 3.2 *Suppose we are given a* string *tree ST containing exclusively the substrings of the text T and let a be the* active point *of ST. Consider an explicit vertex $i \in V(ST)$ defined like this:*

- *If a is explicit then i is equal to a.*
- *Otherwise, i is a* parent *of the implicit vertex a.*

The explicit vertex i is called the ***insertion point*** *of the string tree ST.*

An explicit vertex marked by the insertion point is the closest ancestor of the active point. When compared to the the active point, it has one considerable advantage — it is always explicit. This is the main reason why is it used in the McCreight's algorithm, whose pseudocode is now presented.

Algorithm 3.1: ***McCreight's suffix tree construction***

```
Data:   T — input text
        n — length of the text T
Result: ST — suffix tree on top of the text T


ST ← ({r}, ∅); /* empty suffix tree containing only the root r */
p ← r; /* the first insertion point is the root */
s ← null; /* source vertex of the suffix link to be created */
for i ← 1 to n do
        /* i is an index of the first character of the currently inserted suffix */
        InsertSuffix(ST, T, n, i, p, s);
end
return ST;
```

The variable `s` is used to contain the most recently created vertex, if it still needs to have its *suffix link* set. The function `InsertSuffix`, which performs the operation of inserting a suffix into the suffix tree, also creates a suffix link for the vertex `s` if $s \neq null$.

The reason why a suffix link might not be created in a single call to this function is that the target vertex of this suffix link may not exist yet. In this case, it will be created

in the next call to this function. The variable `s` is used to pass the source vertex of the missing suffix link between the consecutive calls to the function `InsertSuffix`.

This function matches the appropriate part of the currently inserted suffix with the label of the currently examined edge. This operation has previously been introduced as *scanning*. According to its result, the currently examined edge might be split. Then a new leaf corresponding to the newly inserted suffix is created. In more detail, it can be described using the following pseudocode:

```
function InsertSuffix (ST, T, n, i, p, s)

Input:   ST — partially constructed suffix tree
         T — entire text
         n — length of the text T
         i — starting position of the suffix to be inserted
         p — current insertion point itself or its ancestor
         s — source vertex of the suffix link to be created
Output:  ST — partially constructed suffix tree containing the newly inserted suffix of T
         p — next insertion point itself or its ancestor
         s — source vertex of the new suffix link to be created
Returns: insertion result

k ← i + p.depth; /* setting the index of the first text character to be examined */
/* while there is a "possibly matching" edge */
while (Branch(ST, T, k, p, c) == success) do
        /* c is the selected child of p */
        result ← Scan(ST, T, n, k, p, c, d); /* matching the edge p→c */
        if ((result == complete match) and (all the suffix characters match)) then
                p ← SuffixLink(ST, p);
                return success;
        else if (result == complete match) then
                p ← c; /* descending down along the p→c edge */
                k ← k + d; /* here, d is the length of the p→c edge */
        else if (result == partial match (any)) then
                /* split position is determined by the number of matching characters d */
                SplitEdge(ST, T, p, c, d, s);
                CreateLeaf(ST, T, i, p);
                simulate the suffix link; /* see subsection 3.1.2 */
                return success;
        end
end
CreateLeaf(ST, T, i, p);
p ← SuffixLink(ST, p);
return success;
```

The lines displayed in red are required only if there is no guarantee that the text is terminated with a *terminating* character. As we pointed out in Chapter 1, it is a common practice to have the text terminated with a *terminating* character at first and just then construct a suffix tree on top of it. Assuming that this has been done enables us to simplify the function `InsertSuffix` like this:

```
function InsertSuffix (ST, T, n, i, p, s)
a simplified version, which supposes that the text T is terminated with a terminating character

Input:   ST — partially constructed suffix tree
         T — entire text
         n — length of the text T
         i — starting position of the suffix to be inserted
         p — current insertion point itself or its ancestor
         s — source vertex of the suffix link to be created
Output:  ST — partially constructed suffix tree containing the newly inserted suffix of T
         p — next insertion point itself or its ancestor
         s — source vertex of the new suffix link to be created
Returns: insertion result

k ← i + p.depth; /* setting the index of the first text character to be examined */
/* while there is a "possibly matching" edge */
while (Branch(ST, T, k, p, c) == success) do
        /* c is the selected child of p */
        result ← Scan(ST, T, n, k, p, c, d); /* matching the edge p→c */
        if (result == complete match) then
                p ← c; /* descending down along the p→c edge */
                k ← k + d; /* here, d is the length of the p→c edge */
        else if (result == partial match (any)) then
                /* split position is determined by the number of matching characters d */
                SplitEdge(ST, T, p, c, d, s);
                CreateLeaf(ST, T, i, p);
                simulate the suffix link; /* see subsection 3.1.2 */
                return success;
        end
end
CreateLeaf(ST, T, i, p);
p ← SuffixLink(ST, p);
return success;
```

We shall now briefly explain what this function does. At first, it tries to find a "possibly matching" edge leading from the vertex `p`, which is either the current insertion point or one of its ancestors. Its label starts with the next unprocessed character of the currently inserted suffix. Because of the branching requirement (Definition 2.20), there can be at most one such edge. The operation of selecting the appropriate edge is usually called simply *branching*.

If there is no "possibly matching" edge, this function just creates a new edge leading from the vertex `p`, which in this case is the current insertion point itself, to a new leaf corresponding to the currently inserted suffix. Then it adjusts the variable `p` to the next insertion point itself or to its ancestor by following a suffix link from the current insertion point `p` and returns successfully.

On the other hand, if there is a "possibly matching" edge leading from the variable `p`, this function tries to match the label of that edge. If the *entire* edge label matches, the variable `p` is changed to the target vertex of this matching edge. Then the branching operation is repeated.

If the edge label matches only *partially*, then this function does the following: It splits this edge at the position just *after* the last matching character by creating an explicit vertex there. This vertex is the current insertion point. Then it creates a new edge leading from this insertion point to a new leaf corresponding to the currently inserted suffix.

Finally it attempts to set the suffix link of the newly created vertex and at the same time it adjusts the variable `p` to the next insertion point itself or to its ancestor. This last operation is usually referred to as the suffix link *simulation* and it will be explained in subsection 3.1.2.

It is important to note that after the edge is split, there is at least one more *unmatched* character of the currently inserted suffix left. The reason is that the text is supposed to be terminated with a *terminating* character, which is guaranteed to occur only at the very end of the text.

However, if this were not the case, we would have to check at first if all the characters of the currently inserted suffix had already been matched. If they had, we would be almost done and no new vertex would need to be created. We would only have to adjust the variable `p` to the next insertion point itself or to its ancestor by following a suffix link from the current insertion point `p`. Otherwise, there would be at least one unmatched character of the currently inserted suffix and we would continue to check if the edge label matches completely or partially the usual way.

3.1.1 Insertion point adjustment

In general, the next insertion point is the parent of the vertex, either explicit or implicit and possibly not yet present in the current partially constructed suffix tree, which corresponds to the next, shorter suffix. It is our intention to set the variable `p` either directly to the next insertion point or to some of its ancestors while keeping the time complexity reasonably low.

If no edge has been split, then all the explicit internal vertices currently present in the partially constructed suffix tree must already have their suffix links set. This allows us to use the suffix link starting at the current insertion point and set the variable `p` to the target vertex of this suffix link. It is either the new insertion point itself or some of its ancestors. Both of these cases are acceptable, because at the beginning of each call to the function `InsertSuffix`, the variable `p` is adjusted by descending down until it reaches the current insertion point.

When the edge has been split and a new explicit internal vertex has been created, it becomes the current insertion point. If necessary, the target vertex of a suffix link starting at the vertex created in the previous call to the function `InsertSuffix` is set to this newly created vertex.

Also, the suffix link *starting* at this newly created vertex needs to be set. However, its target vertex might not be explicit, yet. It has to be already present in the partially constructed suffix tree, but it can be implicit. The following theorem explains why.

Theorem 3.1 (Suffix link's target presence) *Suppose we are given a partially constructed suffix tree ST obtained at some point of the McCreight's algorithm. Let p be an explicit internal vertex created during the insertion of the last suffix. Also, let q be an implicit or explicit vertex, which is the target vertex of a suffix link starting at the vertex p. Then q is already present in ST.*

Proof Let the string corresponding to the vertex p be called P, the string corresponding to the vertex q be called Q and the last inserted suffix be called L.

For a contradiction, suppose that q is not present in ST. Then no suffix already contained in ST can have Q as its prefix.

The vertex corresponding to the last inserted suffix is a leaf, because a new explicit internal vertex has been created during the insertion of the last suffix. Moreover, it is one of the two children of the vertex p. Let the other child of this vertex be called c and let the string corresponding to it be called C.

If c is a leaf, then C is a suffix which is *longer* than L. If c is not a leaf, then C is a prefix of at least two suffixes, which are *longer* than L.

Either way, there exists a *suffix*, which is longer than L and whose prefix is C. Let it be called S_p, since the string P is also its prefix.

Consider a suffix S_q, which is the longest *proper* suffix (Definition 2.6) of the string S_p. Since S_q is at least as long as the suffix L, it means that it has to be already contained in ST. Considering that Q is a suffix of P and $|Q| = |P| - 1$, it means that the string Q is a prefix of S_q. And this is a contradiction. □

In case the suffix link's target vertex is not explicitly present in the current partially constructed suffix tree, it will be made explicit in the next call to the function `InsertSuffix`. That is why we have to remember the current insertion point as a source vertex of this suffix link.

3.1.2 Suffix link simulation

The adjustment of the insertion point for the next, shorter suffix is a little bit more complex when the edge has been split, because at the same time we also attempt to fill in the missing suffix link. This combined operation is usually called the suffix link *simulation*.

There are two main ways to simulate a suffix link. The first and the original one is called the *top-down* suffix link simulation.

It makes use of the fact that the suffix links for all the other explicit internal vertices are already set. So, we can use the suffix link of the parent of the current insertion point, move to its target vertex and then descend down to the appropriate branch and depth.

However, as we have mentioned previously, the explicit internal vertex at the desired branch and depth might not exist yet. In this case, it will be created in the next call to the function `InsertSuffix`, so we just remember the source vertex of the missing suffix link and set it later. If the vertex at the appropriate branch and depth exists, the desired suffix link is created.

The variable `p` is adjusted either to the target vertex of the simulated suffix link, if it is explicitly present, or to its parent. The following is a pseudocode of the function performing the *top-down* suffix link simulation.

```
function TopDown (ST, T, i, pp, p, s)

Input:   ST — partially constructed suffix tree
         T — entire text
         i — starting position of the substring corresponding to the vertex p
         pp — parent of the vertex p
         p — source vertex, for which we would like to simulate its suffix link
Output:  ST — partially constructed suffix tree containing the newly created suffix link
         p — target vertex of the simulated suffix link or its parent
         s — source vertex of the suffix link to be created later, if necessary
Returns: suffix link simulation's result

d ← p.depth - 1; /* the depth of the target vertex of the simulated suffix link */
s ← p; /* remembering the source vertex of the simulated suffix link */
p ← SuffixLink(ST, pp); /* using the suffix link starting at the parent of p */
k ← i + p.depth; /* index of the first text character to be examined */
/* while there is a "possibly matching" edge */
while (Branch(ST, T, k, p, c) == success) do
        /* c is the selected child of p */
        if (c.depth < d) then
                k ← k + length of the p→c edge;
                p ← c; /* descending down along the p→c edge */
        else if (c.depth == d) then /* target vertex has been found */
                s.suffix_link ← c; /* completing the suffix link */
                p ← c; /* setting up the output variable */
                s ← null; /* clearing the source vertex */
                return success;
        else if (c.depth > d) then
                return partial success; /* the target vertex is not explicit yet */
        end
end
return failure; /* none of the current branches contains the target vertex */
```

At first, this function follows a suffix link starting at the parent of the vertex, for which we would like to simulate its suffix link. In case of McCreight's algorithm, it is the current insertion point. Then it descends down to the appropriate branch and depth in the partially constructed suffix tree until it reaches the position of the supposed target vertex of the simulated suffix link. If this vertex is explicitly present, a new suffix link is created. Otherwise, the source vertex of this suffix link is remembered and used in the next call to the function `InsertSuffix`.

Note that in case of McCreight's algorithm, it is not necessary to *scan* the edges while descending, because the target vertex of the suffix link is guaranteed to be present in the partially constructed suffix tree. It might not be explicit, but as shown in theorem 3.1, it has to be available.

Similarly, when this function is used by the Ukkonen's algorithm, the edge scanning is not necessary as well. The reasons are slightly different and are explained in theorem 3.2. As a consequence, this function cannot return failure when used by both McCreight's or Ukkonen's algorithms.

The *top-down* approach to the suffix link simulation has one considerable disadvantage. While descending down to the appropriate branch and depth, there might be too many branching operations necessary. In fact, their number might be as high as the total number of characters in the label of an edge leading to the source vertex of the suffix link to be simulated, decreased by one.

To our knowledge, when using any reasonably space-efficient suffix tree implementation technique, the branching operations are always among the slowest. Therefore it is very important to keep their number as low as possible.

Senft and Dvořák [SD12] propose an improvement to the traditional suffix link simulation technique, which aims to decrease the total number of branching operations necessary for the suffix link simulation. It is called the *bottom-up* suffix link simulation.

Its main idea is the same — to make use of the existing suffix links — but it is performed "the other way around", literally. Given a vertex for which we would like to simulate its suffix link, we at first select one of its children. Then we follow a suffix link starting at this child and finally we climb up to the appropriate depth.

Note that there is no need for any branching operations. The climbing up in the (partially constructed) suffix tree requires only that for each vertex, except for the root, we have access to its parent. Since these vertices always have exactly one parent, no edge selection is necessary.

On the other hand, we have to store the parent for each vertex, except for the root. Many algorithms for the suffix tree construction, including McCreight's, do not require that information. So, using this suffix link simulation technique almost certainly increases the memory requirements of the suffix tree.

Despite not being optimal in theory, Senft and Dvořák [SD12] argue that this approach is beneficial in practice. Their benchmarks show that there is a considerable reduction in the number of branching operations necessary for the suffix tree construction on top of many commonly used types of text. However, there exists at least one family of texts, which fully exploits the theoretical disadvantages of this approach. More details are given in Chapter 5.

In this thesis we support the results of Senft and Dvořák [SD12] and state that in some cases, the use of *bottom-up* suffix link simulation technique is favorable and should be recommended. Its pseudocode is similar to that of the *top-down* suffix link simulation:

```
function BottomUp (ST, T, i, p, c, s)

Input:   ST — partially constructed suffix tree
         T — entire text
         i — starting position of the substring corresponding to the vertex p
         p — source vertex, for which we would like to simulate its suffix link
         c — child of the vertex p
Output:  ST — partially constructed suffix tree possibly containing the newly created suffix link
         p — target vertex of the simulated suffix link or its parent
         s — source vertex of the suffix link to be created later, if necessary
Returns: suffix link simulation's result

d ← p.depth - 1; /* the depth of the target vertex of the simulated suffix link */
s ← p; /* remembering the source vertex of the simulated suffix link */
p ← SuffixLink(ST, c); /* using the suffix link starting at the child of p */
/* while we are too deep */
while (p.depth > d) do
        p ← p.parent; /* ascending up to the parent of p */
end
/* from now on, the depth of p is at most d */
if (p.depth == d) then /* target vertex has been found */
        s.suffix_link ← p; /* completing the suffix link */
        s ← null; /* clearing the source vertex */
        return success;
end
return partial success; /* the target vertex is not explicit yet */
```

As we can see, this function at first uses the suffix link of the child of the vertex `p`, for which we would like to simulate the suffix link. Then it uses the parent pointers to ascend up to the desired depth. If the reached depth is equal to the depth of the desired target vertex, it sets the suffix link, clears the variable `s` and returns successfully. Otherwise, the target vertex is not explicit yet, but this function has at least found its position, namely its future parent and child. This is a partial success and it is returned.

Either of these two functions for the suffix link simulation can be used by both McCreight's or Ukkonen's algorithms. When used by McCreight's algorithm, they need to be called if the edge has been split and a new leaf vertex has been created.

3.2 Ukkonen's algorithm

This algorithm was introduced by Ukkonen [Ukk95] as an *on-line* alternative to the other suffix tree construction algorithms known at the time, namely the ones presented by Weiner [Wei73] and McCreight [McC76]. In Chapter 1, we have already mentioned that its main idea is to iteratively *prolong* the suffixes currently present in the partially constructed suffix tree.

The construction starts from the empty suffix tree. At the beginning of every prolonging step, all the suffixes currently present in the partially constructed suffix tree are prolonged by one character, namely the next text character. Then a new suffix consisting of this text character only is inserted, if necessary.

One such iteration effectively transforms the partially constructed suffix tree on top of the prefix of length n of the text into the partially constructed suffix tree on top of the prefix of length $n+1$ of the very same text. It is repeated until all the suffixes are present

and are long enough so that the suffix tree on top of the entire text is complete.

In contrast with the McCreight's algorithm, the intermediate data structure obtained after each prolonging step has every property of a complete suffix tree built on top of the currently processed prefix of the text. Moreover, no other text parts are accessed during its construction.

This enables the Ukkonen's algorithm to be used for constructing the suffix tree *on-line*. In this type of construction the text may not be entirely available at the beginning. Instead, it can be accessible only sequentially and exclusively in *forward* direction. Similar restrictions arise when constructing a suffix tree over a *sliding window*, where an *on-line* suffix tree construction algorithm is essential. More details are given in section 3.3.

The pseudocode of this algorithm looks like this:

Algorithm 3.2: *Ukkonen's suffix tree construction*

```
Data:   T — input text
        n — length of the text T
Result: ST — suffix tree on top of the text T


ST ← ({r}, ∅); /* empty suffix tree containing only the root r */
p ← r; /* the first insertion point is the root */
i ← 1; /* index of the first character of the first suffix to be prolonged */
for j ← 1 to n do
        /* j is an index of the character to be appended */
        ProlongSuffixes(ST, T, i, j, p);
end
return ST;
```

Note that the *insertion point* refers to the same term as used by the McCreight's algorithm and introduced in Definition 3.2. In particular, it is the closest of the active point's *ancestors* represented by explicit vertices. The reason why we use it instead of the *active point* is the same as the reason why we use it in the McCreight's algorithm — it is guaranteed to be explicit.

As we can see, most of the work is carried out by the function `ProlongSuffixes`, which performs the prolonging step itself. When it returns successfully, the intermediate suffix tree becomes a complete suffix tree on top of the current prefix of the text. This function checks which suffixes of the currently processed prefix of the text are already present in the current partially constructed suffix tree. The ones which are not present are inserted. Since all their prefixes must have already been present in the current partially constructed suffix tree, this operation can also be referred to as *prolonging*. The other suffixes do not require any action at all.

The prolonging starts from the longest suffix which needs to be prolonged and continues towards the shorter suffixes. If the currently prolonged suffix is already present in the partially constructed suffix tree, the prolonging step can be safely finished without checking the remaining suffixes for presence. This is correct, because all the shorter suffixes are already present in the suffix tree.

The reason is that the previous iteration created a *complete* suffix tree on top of the shorter prefix of the text, which must have contained *all* the suffixes of this shorter text

prefix. Since then, some suffixes might have been prolonged, but none have been removed. This implies that if a partially constructed suffix tree contains a particular suffix created in the previous prolonging steps, it must also contain *all* of its suffixes.

In order to achieve the desired time complexity, the next iteration will start the prolonging right from the suffix at which the current iteration stopped. This is possible, because all the previous, longer suffixes are prolonged automatically whenever the length of the currently processed text prefix increases. An implementation technique called *open edges* is used to allow that.

3.2.1 Open edges

During each prolonging step, by far not every suffix which needs to be prolonged is prolonged *explicitly*. There is a certain group of suffixes, which are prolonged *implicitly*, requiring no action at all. It is possible because of an improved method for labeling the edges leading to the leaves used by the Ukkonen's algorithm. Rather than using a fixed edge label, these edges are given a *dynamically expanding* edge label, which always ends at the last character of the currently processed prefix of the text. Such edges are called the *open edges*.

When used in the Ukkonen's algorithm, the suffixes corresponding to the leaves are prolonged automatically. If a suffix is inserted into the partially constructed suffix tree in which it has not been present previously, it *must* correspond to a leaf. This means that once a suffix is prolonged by explicitly inserting it into the partially constructed suffix tree, there is no need to perform any other action in order to further prolong it. Usage of the open edges therefore reduces the algorithm's time complexity. They enable the prolonging part to be started from the suffix, at which the prolonging part of the previous iteration stopped.

The usage of open edges is essential for Ukkonen's algorithm to achieve the desired time complexity. Using the alphabet of constant size, this algorithm can construct a suffix tree in the time linear with respect to the length of the input text. Without the open edges, it would not be possible. See e.g. the original Ukkonen's paper [Ukk95] for details.

The following is a pseudocode of the function `ProlongSuffixes`.

```
function ProlongSuffixes (ST, T, i, j, p)

Input:   ST — suffix tree on top of the first j − 1 characters of the text
         T — entire text
         i — starting position of the first suffix to be prolonged
         j — ending position of the suffixes after being prolonged
         p — current insertion point itself or its ancestor
Output:  ST — suffix tree on top of the first j characters of the text
         i — starting position of the first suffix to be prolonged in the next call to this function
         p — next insertion point itself or its ancestor
Returns: prolonging result

s ← null; /* source vertex of the suffix link to be created */
do /* do the prolonging */
        result = ProlongSuffix(ST, T, i, j, p, s);
/* while there might be a suffix, which needs to be prolonged */
while (result == partial success)
return success;
```

Similarly to the McCreight's algorithm, the variable `s` contains the most recently created vertex, if it still needs to have its *suffix link* set. The function `ProlongSuffix`, which performs the operation of prolonging (or inserting) a single suffix, also creates a suffix link for the vertex `s` if `s` $\neq$ *null*. The reason why a suffix link might not be created in a single call to this function is the same as in the McCreight's algorithm — the target vertex of this suffix link may not exist yet.

As we have already mentioned, a new suffix consisting of the next text character only is inserted into the partially constructed suffix tree during each prolonging step. Fortunately, no extra operation is necessary to perform this, because the insertion of a new suffix is very similar to the prolonging of a suffix and both operations can be handled by the same function, namely `ProlongSuffix`. We just need to call it sufficiently many times and with the appropriate parameters. Considering that, the suffix in question will be inserted during the last call to this function, if necessary.

The operation of prolonging (or inserting) a single suffix, as performed by the function `ProlongSuffix`, is very similar to the operation of inserting a new suffix used in the McCreight's algorithm. We start at the current insertion point or at its ancestor, depending on whether the previous call to this function has already reached the current insertion point or not. From here, we descend down to the appropriate branch and depth, trying to reach a vertex corresponding to the currently prolonged suffix. If such a vertex, either explicit or implicit, is found, we are done. It means that the suffix to be prolonged is already contained in the current partially constructed suffix tree. Moreover, in this case we can also finish the entire prolonging step, because all the shorter suffixes are also contained in this suffix tree.

In case the vertex corresponding to the currently prolonged suffix is not present, we have to insert it. We need to reach its parent, which eventually becomes the new insertion point. However, it might be implicit, so in this case we have to make it explicit at first. Then we can insert a new leaf corresponding to the currently prolonged suffix. Thanks to the *open edges*, it will be further prolonged automatically, so we can advance to the next, shorter suffix.

If the leaf corresponding to the currently prolonged suffix is already present, it means that this suffix has already been prolonged automatically and we are done. It also means that all the shorter suffixes have already been prolonged automatically or are already contained in the current partially constructed suffix tree. In this case we can safely finish the entire prolonging step.

The pseudocode of this function is very similar to the pseudocode of the function `InsertSuffix` used by the McCreight's algorithm, because both of them perform almost the same actions. It looks like this:

```
function ProlongSuffix (ST, T, i, j, p, s)

Input:   ST — partially constructed suffix tree on top of the first j − 1 characters of the text
         T — entire text
         i — starting position of the suffix to be prolonged
         j — ending position of the suffix after being prolonged
         p — current insertion point itself or its ancestor
         s — source vertex of the suffix link to be created
Output:  ST — partially constructed suffix tree possibly containing the prolonged or new suffix
         i — starting position of the suffix to be prolonged in the next call to this function
         p — next insertion point itself or its ancestor
         s — source vertex of the new suffix link to be created
Returns: prolonging result

k ← i + p.depth; /* setting the index of the first text character to be examined */
/* while there is a "possibly matching" edge */
while (Branch(ST, T, k, p, c) == success) do
        /* c is the selected child of p */
        result ← Scan(ST, T, j, k, p, c, d); /* matching the edge p→c */
        if (result == complete match) then
                if (c is a leaf) then
                        p ← SuffixLink(ST, p);
                        i ← i + 1; /* advancing to the next suffix */
                        return success; /* suffix is prolonged implicitly */
                else if (c.depth == j - i + 1) then /* exact depth reached */
                        p ← c; /* descending down along the p→c edge */
                        return success; /* suffix is already explicitly present */
                end
                k ← k + d; /* here, d is the length of the p→c edge */
                p ← c; /* descending down along the p→c edge */
        else if (result == partial match (long edge)) then
                return success; /* suffix is already implicitly present */
        else if (result == partial match (mismatch)) then
                /* split position is determined by the number of matching characters d */
                SplitEdge(ST, T, p, c, d, s);
                CreateLeaf(ST, T, i, p);
                simulate the suffix link; /* see subsection 3.1.2 */
                i ← i + 1; /* advancing to the next, shorter suffix */
                return partial success; /* split the edge and created a new leaf */
        end
end
CreateLeaf(ST, T, i, p);
p ← SuffixLink(ST, p);
i ← i + 1; /* advancing to the next, shorter suffix */
return partial success; /* created a new leaf */
```

When the currently examined edge matches only *partially*, a new explicit internal vertex is created. Naturally, it needs to have its suffix link set. At the same time, a suffix link starting at the vertex created in the previous call to this function might need to be set as well.

In the McCreight's algorithm, both of these actions are handled by the suffix link *simulation*. Fortunately, we can use it also in the Ukkonen's algorithm, because the idea remains the same.

We can adapt both the top-down or bottom-up suffix link simulation techniques for the Ukkonen's algorithm. Actually, no adjustments are necessary and we can even use the

same functions, because they are algorithm-independent.

These suffix link simulation techniques require that the target vertex of the suffix link simulation is already present in the current partially constructed suffix tree. It might be only implicit, but it has to be present. We shall now explain why.

Theorem 3.2 (Suffix link's target presence) *Suppose we are given a partially constructed suffix tree ST obtained at some point of the Ukkonen's algorithm. Let p be an explicit internal vertex in ST which has been created during the prolonging (or insertion) of the last suffix. Then the partially constructed suffix tree ST must have* already *contained a vertex q, either explicit or implicit, which is the target vertex of a suffix link starting at the vertex p.*

Proof We already know that partially constructed suffix tree obtained after each prolonging step of the Ukkonen's algorithm is in fact a *complete* suffix tree on top of the current prefix of the text. This means that the target vertex of a suffix link starting at any explicit or implicit vertex in such a suffix tree is also present in this suffix tree.

Let ST_c be a *complete* suffix tree created after the most recently finished prolonging step of the Ukkonen's algorithm prior to the point of obtaining ST. The suffix tree ST_c has since been expanded to become a partially constructed suffix tree ST. During this expansion, none of the vertices of the original suffix tree ST_c have been removed. However, some leaves might have been added and some of the vertices which were implicit in ST_c might have become explicit in ST.

Consequently, ST contains the target vertices of all the suffix links starting at all the vertices, either explicit or implicit, which were originally present in the suffix tree ST_c. The question is whether the vertex p, for which we would like to simulate its suffix link, was also present in the suffix tree ST_c.

If p were not present in ST_c, it would have to be an implicit vertex *inside* an edge created during the expansion of the suffix tree ST_c to the partially constructed suffix tree ST. But during the Ukkonen's suffix tree construction, each suffix is always prolonged by *exactly* one character. This means that the maximum, as well as the only allowed length of a new edge's label is 1. As a result, there can *not* be any implicit vertex inside a newly created edge, which implies that the vertex p was already present in the suffix tree ST_c. □

3.3 Sliding window

In this section, we explain how to adapt the Ukkonen's algorithm to the suffix tree construction over a *sliding window*. The McCreight's algorithm cannot be used effectively for this type of construction, because it is not an *on-line* algorithm. It means that it cannot provide access to full-featured intermediate suffix tree until the construction on top of the entire text is complete.

The formal definition of the sliding window is presented in Definition 2.32. As we have pointed out in Chapter 1, an algorithm suitable for suffix tree construction over a sliding window must be able not only to create a suffix tree on top of the static text, but it must also provide the means for the suffix tree maintenance when the sliding window moves. Now we explain how the Ukkonen's algorithm can be adjusted in order to meet this requirements.

Even without any modifications, this algorithm can create the suffix tree on top of the current sliding window from scratch. However, it is more important how to quickly update such a suffix tree when the sliding window moves.

This movement can be split into two parts. At first, the front of the sliding window advances while increasing its length by one. After that, the suffix tree needs to be updated for the first time, so that it contains all the suffixes of this longer sliding window. Such an update is the very nature of the prolonging step performed by the Ukkonen's algorithm, so it is very easy to realize it.

Then, the back of the sliding window advances while decreasing its length by one. At the same time, the suffix tree is updated again, so that it contains only the suffixes of the new, shorter sliding window. The original Ukkonen's algorithm does not include such an update, but it can be extended to contain it.

In order to describe such an extension, we at first have to exactly determine, what needs to be done. Consider the text T of length n and the sliding window $W = (b, f)$ of length $m \leq n$ over this text. Let the suffix tree built on top of this sliding window be called ST. If the size of the sliding window increases (f is incremented), the suffix tree is updated by performing one prolonging step of the Ukkonen's algorithm. On the other hand, if the size of the sliding window decreases (b is incremented), we need to update the suffix tree in a different way.

If we compare the suffix tree on top of the original sliding window and the suffix tree on top of the shortened sliding window, we can see that they are *almost* identical. The only difference is that the longest suffix of the original sliding window is *not* present in the suffix tree on top of the shortened sliding window while all the remaining suffixes *are* present. This means that if we would like to adjust the suffix tree on top of the original sliding window so that it would become the suffix tree on top of the shortened sliding window, we have to *remove* the above-mentioned suffix from the suffix tree. Naturally, this removal has to *preserve* all the other suffixes present in the suffix tree.

The first algorithm able to perform this has been presented by Fiala and Greene [FG89]. Its idea is very simple. At first, the leaf corresponding to the longest suffix is located and either it is deleted or the label of the edge leading to it is shortened. Then, in case the leaf has been deleted, its parent is checked and it is deleted as well if it has only one child remaining. In this case, the edge originally leading to the deleted vertex from its parent and the edge originally leading from the deleted vertex to its only remaining child are joined into the new edge leading from the former parent of the deleted vertex to its only remaining child. The label of this edge consists of the concatenation of the edge

labels of the original two edges. Now we present a pseudocode of this algorithm.

Algorithm 3.3: ***Longest suffix deletion by Fiala and Greene***

```
Data:   T — entire input text
        n — length of the text T
        W — sliding window over the text T
        ST — suffix tree on top of the sliding window W
Result: W — sliding window with its back advanced by one character
        ST — suffix tree on top of the new sliding window W


l ← leaf corresponding to the longest suffix in ST;
p ← explicit or implicit vertex corresponding to the longest repeated suffix in ST;
pp ← parent of the vertex p in ST;
if (p is implicit) then /* p is located on an edge leading to the leaf l */
        shorten the label of the pp → l edge in ST according to p;
else /* p is explicit and it is the parent of the leaf l */
        delete the p → l edge in ST;
        delete the leaf l from ST;
        if (p has only one child remaining) then
                c ← the only remaining child of the vertex p in ST;
                delete the pp → p and p → c edges in ST;
                delete the vertex p from ST;
                create the pp → c edge in ST;
                set up its edge label appropriately;
        end
end
return success;
```

This algorithm uses the longest *repeated* suffix. It is the longest suffix which is also a *proper* prefix (Definition 2.6) of the longest suffix currently present in the suffix tree. The vertex corresponding to the longest repeated suffix determines a position in the suffix tree where the longest suffix deletion occurs.

If the longest repeated suffix corresponds to an explicit vertex p, it means that the longest suffix currently present in the suffix tree corresponds to a leaf l, which is one of the children of the vertex p.

If we delete this leaf and its incoming edge from the suffix tree, no other suffix except for the longest suffix currently present in the suffix tree is deleted. The reason is that no implicit vertex inside the $p \to l$ edge corresponds to a suffix. If there were a suffix which corresponded to an implicit vertex inside this edge, it would mean that this suffix would be the longest repeated suffix instead. And by our assumption, it is not true. This means that deleting the leaf l and its incoming edge in this case is correct.

After this deletion, the vertex p might be left with only one child remaining. Since the compactness requirement (Definition 2.18) does not allow that, the two edges leading to and from the vertex p have to be joined together. The label of the resulting edge is formed by the concatenation of the edge labels of the original two edges.

On the other hand, when the longest repeated suffix corresponds to an *implicit* vertex

p, the longest suffix currently present in the suffix tree corresponds to a leaf l, which is the target vertex of the edge containing the implicit vertex p. Let the parent of the leaf l be called pp. Then, instead of deleting the leaf l and its incoming edge, we just change the label of this edge. More precisely, we have to *shorten* it.

Just like in the previous case, there cannot be any suffix, which corresponds to an implicit vertex located deeper than the vertex p inside the edge between the vertex pp and the leaf l. However, there can be suffixes corresponding to implicit vertices, which are located inside the same edge, but are less deep than p. The reason is the same as in the previous case: The suffix in question would have to be the longest repeated suffix instead of the suffix corresponding to the implicit vertex p.

This algorithm can surely be used for the deletion of the longest suffix from the suffix tree while keeping all the other suffixes intact. But there might be issues regarding the effective ways to obtain the necessary vertices, namely the ones corresponding to the longest suffix and the longest *repeated* suffix.

As shown in Chapter 4, all the suffix tree implementation techniques suitable for the sliding window used in this thesis provide easy access to the leaf corresponding to the longest suffix currently present in the suffix tree. Usually, there is a pointer referring directly to this vertex. So, it is simple to effectively obtain it.

In order to access the vertex corresponding to the longest repeated suffix during the Ukkonen's suffix tree construction over a sliding window, we can use the *active point* or alternatively, the *insertion point*. In fact, during the course of the Ukkonen's algorithm, the active point exactly determines the location of the vertex corresponding to the longest repeated suffix.

If the active point is an implicit vertex inside an edge leading to the leaf corresponding to the longest suffix currently present in the suffix tree, then it is also the vertex corresponding to the longest repeated suffix. Otherwise, the vertex corresponding to the longest repeated suffix is the parent of the leaf corresponding to the longest suffix currently present in the suffix tree.

This proposition has been proved in subsection 2.2.3 of the PhD thesis "Structures of String Matching and Data Compression" by N. Jesper Larsson [Lar99], so we will omit its proof here. Larsson reviewed the original algorithm by Fiala and Greene [FG89], explained some of its less clear parts and presented a slightly modified version of their algorithm for the suffix tree construction over a sliding window. The differences between the two algorithms are rather small, but since the Larsson's version is more recent and it is explained using the modern terminology, it is probably more commonly used.

The vertex corresponding to the longest repeated suffix could therefore be located using the insertion point. In case it is an implicit vertex located inside an edge leading from the current insertion point to the leaf corresponding to the longest suffix currently present in the suffix tree, its location is determined by the currently inserted suffix. To be exact, the longest repeated suffix would also be the currently inserted suffix. On the other hand, in case the vertex corresponding to the longest repeated suffix is explicit, it is equal to the insertion point itself.

In theory, this is all we need in order to create and maintain the suffix tree over a sliding window. In practice, however, there are still some details missing. Most importantly, we have to take some implementation details into account. In this case, the edge labels require our attention.

3.3.1 Edge label maintenance

In most implementations of the suffix tree construction over the static text, the edge labels are represented by the strings only *indirectly*. Instead of the string itself, only a pair of pointers is used for each edge's label. They determine the beginning and the end of the text substring, which corresponds to the label of the particular edge.

This method is used because of its low memory requirements compared to storing the entire strings representing the edge labels directly. When the suffix tree is constructed over the static text, the usage of pointers to the text is quick and effective. But when the sliding window is used instead of the static text, these pointers might cause problems. In particular, they can become invalid.

The reason is that in practice, the sliding window cannot be implemented simply as an ordered pair of pointers outlining the sliding window in the entire text. It would require that the entire text is available and it is possible to randomly access it, which would taint one of the main sliding window advantages — its space requirements. Considering that, the sliding window is usually implemented as a copy of the corresponding part of the text, which is updated whenever the sliding window moves. As a result, the parts of the text outside the sliding window are not accessible. And therefore the pointers pointing to these parts of the text are invalid.

During the suffix tree construction, the edges are created so that their labels are contained in the current sliding window. But as the sliding window moves, the labels are also moving out of the sliding window and they are becoming invalid. So, when an edge is created, it is incorrect to assume that its label will always remain valid. It will not. But fortunately, it is possible to introduce some additional measures, which will keep the edge labels valid during the entire construction.

Such a mechanism is usually called the edge label *maintenance*. To our knowledge, there exist three approaches to achieve this goal. They all update the edge labels so that they point to the currently available part of the text. The reason why the current sliding window always contains the part of the text equivalent to the arbitrary edge label is that the current suffix tree is constructed on top of the current sliding window. And this means that all the labels used by all the edges in this suffix tree have to be substrings of the current sliding window.

The first algorithm for the edge label maintenance has been presented by Fiala and Greene [FG89]. It updates the edge labels selectively, when necessary. In order to determine which edge labels need to be updated, the authors introduce the so-called *percolating* update. It requires that each explicit internal vertex contains additional information, which indicates how urgent it is for the label of the edge leading into its parent to be updated. Two values of this information are necessary, so it is sufficient to store it as a single bit. In compliance with the original terminology used by Fiala and Greene [FG89], we call this information the *update bit*. Other authors also use the name credit bit or credit counter, despite the fact that it can have only two values.

Initially, all the update bits are set to 0. Every time a new leaf is created, the update bit of its parent is flipped. At the same time, the label of an edge leading into the new leaf's parent is updated. The new label is selected to be the substring of the suffix corresponding to the newly created leaf. It is therefore guaranteed to be present in the current sliding window.

Whenever the update bit of a vertex is flipped and its incoming edge is updated, we have to check whether it is necessary to propagate this change upwards in the suffix tree. If the vertex had its update bit flipped from 1 to 0, then the update is propagated. It

means that the update bit of its parent is also flipped. Similarly, the edge leading into this parent is updated as well. This way, we continue until no more flipping is necessary (the update bit is flipped from 0 to 1) or until we reach the child of the root (from which the update cannot be further propagated). The propagation of the bit flips and the updates upwards in the suffix tree is called the *percolation* of the update.

If a leaf is deleted or its edge label is shortened, the update bit of its parent is flipped and the label of its parent's incoming edge is updated. Also, if an explicit internal vertex is deleted, the update bit of its former parent is flipped and the label of its incoming edge is updated. Then, the update is percolated.

It is important to know *how* exactly are these updates expected to change the encountered edge labels. In the previous case when the new leaf has been created, it is sufficient to update the edge labels so that they are substrings of the suffix corresponding to the newly created leaf. But when a leaf is deleted, the updated edge labels can not be changed to the substrings of the suffix corresponding to the deleted leaf. Instead, they have to be updated to the substrings of the suffix corresponding to one of the remaining children of the deleted leaf's parent.

Moreover, since the edge labels higher in the tree might have already been updated with more recent edge labels, we always have to check whether the updated edge label is more or less recent than the current edge label. The edge label is more recent if it starts at the later text character. The update is percolated upwards so that the following edges will be updated with the substrings of the more recent suffix.

As stated by Fiala and Greene [FG89], this type of update is correct and has constant amortized time complexity. This means that its usage can worsen the overall time complexity of the suffix tree construction over a sliding window only by a constant factor. However, in Chapter 1 we have mentioned that the original correctness proof was found to be incorrect by Senft [Sen05b], who also provided the correct proof. Fortunately, the percolating update itself as well as its estimated time complexity have always been correct, so it can be used without any adjustments.

The percolating update has been reviewed by Larsson [Lar99]. He provided a slightly modified and extended version which better suited his needs. However, due to its similarity, we decided not to use it.

The last method for the edge label maintenance has been presented by Senft [Sen05b]. It is a simpler alternative to the percolating update which does not require the extra information (the update bit) to be stored in every explicit internal vertex. On the other hand, it does require that the sliding window is effectively twice as large as its usually accessible, primary part. The supplementary part of the sliding window is used to contain the parts of the text which have recently been removed from it. Since the size of the supplementary part is the same as the size of the primary part, such enhanced sliding window is capable of storing the same number of past characters as is the size of its entire primary part. The increased effective size of the sliding window allows the edge labels to remain valid longer.

Senft calls this edge label maintenance method the *batch update*. Its idea is to update all the edge labels at once every time the sliding window moves by the length of its primary part. After that, the sliding window can be safely moved by another length of its primary part while no edge label can become invalid. Then, another batch update is performed.

The recommendation of the particular edge label maintenance method is the subject of our analysis. Its result can be found in Chapter 5, while the corresponding benchmarks are present in Chapter 6.

3.4 PWOTD

The last suffix tree construction algorithm analyzed in this thesis has been introduced by Tata, Hankins, and Patel [THP04] and it is called the Partition and Write Only Top Down, or simply PWOTD. It is based on the WOTD-eager algorithm, the *eager* variation of the Write Only Top Down algorithm presented earlier by Giegerich, Kurtz, and Stoye [GKS99]. Different in many ways from both the McCreight's and the Ukkonen's algorithm, the PWOTD algorithm presents another way of constructing a suffix tree.

Originally, this algorithm is part of the Top Down Disk-based or TDD approach for constructing the suffix trees on disk, which has also been presented by Tata, Hankins, and Patel [THP04]. It is designed for efficient suffix tree construction in external memory and consists of the PWOTD algorithm itself and a sophisticated buffer management strategy. The authors state that despite the fact that the theoretical worst case time complexity of their approach is not optimal,[1] it performs very well in practice even for the construction in the main memory on the recent cache-enabled processors. They argue that it is because of much better *locality of reference* than the usual suffix tree construction algorithms by McCreight or Ukkonen. Better locality of reference means less cache misses and this in return translates into faster running times.

Since we are not going to examine the disk-based suffix tree construction in this thesis, we have decided to separate the PWOTD algorithm itself and use it for the traditional, in-memory suffix tree construction. The buffer management strategy of the TDD approach is not necessary, because we are constructing the suffix tree entirely in memory, where no buffers are required. Our decision to implement the PWOTD algorithm has been motivated by the conclusion presented by the authors of the TDD approach, who claim that their technique is faster than the Ukkonen's algorithm even when the entire suffix tree is created in the main memory. In Chapter 6 we show that according to the results of our benchmarks, this is generally *not* true.

The PWOTD algorithm consists of two parts. In the first of them, all the suffixes of the text are divided into a number of groups called *partitions* according to their prefix of the previously determined length. Each partition consists only of the suffixes, which share the *same* prefix of this length. The suffixes which are shorter than the selected prefix length are each placed into a separate partition. Since each suffix has a *different* length, there will be at least as many partitions containing only a single suffix as is the selected length of the prefix, decreased by one. This partitioning part is an extension to the original WOTD-eager algorithm by Giegerich, Kurtz, and Stoye [GKS99].

The prefix length is chosen in advance and it should be large enough so that the average number of suffixes in a single partition is acceptable. Of course, the exact number of suffixes in the individual partitions might vary, because it is dependent on the text structure.

After all the suffixes are divided into the partitions, the partitions themselves are ordered lexicographically with respect to the prefix shared by all the suffixes contained in them. Then, the partitions are partially evaluated and the upper part of the suffix tree is built. This means that for each partition, some prefixes of the suffixes it contains are inserted into the suffix tree. During this part of the partitioning, some of the partitions containing only a single suffix might be completely evaluated and therefore removed from the list of partitions. At the same time, the partitions which are not removed are organized into a *stack* which determines the order of their later evaluation.

[1] it is quadratic with respect to the length of the underlying text

In the second part of this algorithm, the partitions are completely evaluated according to the order determined in the partitioning part. For each partition, the corresponding part of the suffix tree containing all of the partition's suffixes is created. In more detail, the entire algorithm can be described using the following pseudocode.

Algorithm 3.4: ***Partition and Write Only Top Down***

```
Data:   T — input text
        n — length of the text T
        l — length of the prefix used for the partitioning
Result: ST — suffix tree on top of the text T


ST ← ({r}, ∅); /* empty suffix tree containing only the root r */
S ← []; /* empty array of starting positions of all the suffixes of T */
P ← []; /* empty array of the suffix partitions */
ES ← []; /* empty array used as the stack partitions to be evaluated */
/* divide all the suffixes into the partitions */
PartitionSuffixes(T, n, l, S, P, m); /* m is the final number of partitions */
/* create the upper part of the suffix tree */;
PreprocessPartitions(ST, T, n, l, S, P, m, ES);
/* m is the number of partitions arranged for evaluation in the stack ES */
for i ← m downto 1 do /* for each partition */
        /* i is an index of the currently processed partition */
        ProcessPartition(ST, T, n, l, S, ES[i]);
end
return ST;
```

The order in which the partitions are evaluated is not specified by the original PWOTD algorithm as described in [THP04]. Therefore there is some room for enhancements and optimizations left. As the authors suggest, it is possible to evaluate all the partitions even without any preprocessing.

In that case, however, the suffix tree traversal becomes more complicated. The reason is that without the preprocessing, it is not possible to natively traverse the suffix tree from the root to an arbitrary branch. It is caused by the different nature of the upper part of the suffix tree, which is shared by all the partitions, while the lower parts of the suffix tree usually belong to a single partition.

This all means that if the upper part of the suffix tree is not created in advance and shared by all the partitions, it is very likely that most of its parts are duplicated for every partition. And this is undesirable, because it disallows simple traversal across the entire suffix tree. In this case, a special traversal technique needs to be used for the upper part of the suffix tree while the "usual" traversal technique is used for the lower parts of the suffix tree. Since it would make most of the operations with the suffix tree more complicated, we have decided to avoid these problems and implement also the preprocessing of the partitions. How exactly is it done is described in the following subsection.

3.4.1 Partitioning

As we have mentioned previously, the first part of the PWOTD algorithm divides all the suffixes of the text into several partitions. The suffixes which are placed in the same partition share a common prefix. It is at least as long as the length of the prefix used during the partitioning.

In order to divide the suffixes into the partitions, they are at first ordered lexicographically according to their prefix of the previously specified length. The suffixes which are shorter than this prefix are also ordered. If more suffixes share the same prefix, the longer ones are placed at the beginning while the shorter ones at the end. This is different from the traditional lexicographic order, which is defined so that when more strings share the same prefix, shorter ones are placed at the beginning. We have decided to slightly modify this definition, because there are some implementation details which make it more convenient. But of course, we could also choose to order the suffixes sharing the same prefix in the traditional direction.

After the suffixes are lexicographically ordered, we make one final transition over them during which the partitions are created. We just look for the partition boundaries, which determine where one partition ends and the following partition starts. In order to find them, we compare the prefixes of every two neighboring suffixes. If their common prefix is at least as long as the specified prefix length, they belong to the same partition. Otherwise, a partition boundary is encountered and a new partition is created.

The partition itself is therefore just a pair of indices into the array of suffixes. The array of suffixes contains only the starting positions of the individual suffixes. This representation, which is based on the representation proposed by the authors of the PWOTD algorithm in [THP04], is therefore very space-efficient. The pseudocode of the function performing the partitioning itself looks like this:

```
function PartitionSuffixes (T, n, l, S, P, m)

Input:   T — entire text
         n — length of the text T
         l — desired length of the prefix
Output:  S — appropriately ordered array of starting positions of all the suffixes of T
         P — partitions containing the suffixes of the text T
         m — final number of suffix partitions created
Returns: partitioning result

S ← []; /* initializing the array of suffixes represented by their starting positions */
for i ← 1 to n do
        S[i] ← i; /* arranging the suffixes from the longest to the shortest */
end
if (l == 0) then /* no partitioning is required */
        m ← 1; /* only one partition is created */
        P[1] ← new partition containing all the suffixes in S;
        return success;
end
for i ← l downto 1 do /* sorting the suffixes using radix sort */
        /* ordering all the sufficiently long suffixes in S according to their i^th character */
        OrderSuffixes(T, n, i, S);
end
m ← 0; /* setting the current number of partitions to zero */
b ← 1; /* the beginning of the current partition */
for i ← 2 to n do /* looking for the partition boundaries */
        /* if the longest common prefix of the suffixes S[i-1] and S[i] is shorter than l */
        if (MatchPrefixes(T, n, l, S[i-1], S[i]) == failure) then
                /* partition boundary encountered */
                m ← m + 1; /* incrementing the current number of partitions */
                P[m] ← new partition starting at the b^th and ending at the i^th suffix in S;
                b ← i; /* remembering the beginning of the next partition */
        end
end
m ← m + 1; /* incrementing the total number of partitions */
P[m] ← the last partition starting at the b^th and ending at the (n + 1)^st suffix in S;
return success;
```

As we can see, the suffixes are ordered using the *radix sort*. This sorting algorithm has been chosen with regard to the expected short length of the prefix and large number of suffixes. Its subroutine, which orders the suffixes on the specified prefix character, uses the *counting sort*. Under these circumstances, the radix sort is usually faster than any of the comparison-based sorting algorithms.

One its iteration simply orders all the sufficiently long suffixes according to the specified prefix character. This ordering is performed from the least significant character to the most significant character. At the end, all the suffixes are lexicographically ordered according to their prefix of the previously specified length. The ordering of the suffixes at the specified prefix character is handled by the function `OrderSuffixes`, which can be described using the following pseudocode.

```
function OrderSuffixes (T, n, k, S)

Input:   T — entire text
         n — length of the text T
         k — index of the suffix character to be considered while ordering
         S — array of starting positions of the suffixes of T to be ordered
Output:  S — array of starting positions of the suffixes of T ordered on their k^th character
Returns: ordering result

O ← []; /* empty array containing the numbers of individual character occurrences */
TS ← []; /* temporary array of starting positions of suffixes used during the ordering */
for c ← lowest order character to highest order character do
        O[c] ← 0; /* resetting the number of the current character's occurrences to zero */
end
for i ← k to n do
        O[T[i]] ← O[T[i]] + 1; /* increasing number of the character's occurrences */
end

/* transforming the array of character occurrences into the array of starting positions of the
segments of suffixes having the same character at the desired position */
sum ← 1; /* starting position of the next segment */
oldsum ← 1; /* the first segment starts at the first suffix */
for c ← lowest order character to highest order character do
        sum ← sum + O[c]; /* including the number of occurrences of the character c */
        O[c] ← oldsum; /* setting the starting position of the current segment */
        oldsum ← sum; /* remembering the starting position of the next segment */
end
k ← k - 1; /* transforming index into an offset */
for i ← 1 to n - k do
        TS[O[T[S[i] + k]]] ← S[i]; /* placing the suffix S[i] at the determined position */
        O[T[S[i] + k]] ← O[T[S[i] + k]] + 1; /* incrementing the first unused position */
end
for i ← 1 to n do
        S[i] ← TS[i]; /* copying the ordered suffixes back to the original array */
end
return success;
```

As stated before, this function orders the suffixes according to the specified prefix character using the *counting sort*. At first, it just scans all the suffixes at the specified position and counts the number of occurrences of the individual characters. They are then used for placing the suffixes at the correct position into the temporary array of suffixes. Finally, the ordered suffixes are copied back to the original array of suffixes.

The temporary array of suffixes can be shared among all the calls to this function. It is therefore possible to have only one instance of this data structure, which is repeatedly reused. The reason is that the information stored in there needs to be preserved in between the calls. Moreover, at the beginning of this function the contents of this data structure are overwritten.

Another function used during the partitioning of the suffixes is `MatchPrefixes`. Its purpose is to determine whether the provided suffixes share the same prefix of the minimal required length. It is used in the last part of the function `PartitionSuffixes` to find the boundaries between the partitions. Its pseudocode looks like this:

```
function MatchPrefixes (T, n, l, S1, S2)

Input:   T — entire text
         n — length of the text T
         l — desired length of the prefix to be matched
         S1 — starting position of the first suffix of T to be matched
         S2 — starting position of the second suffix of T to be matched
Returns: matching result

if (S1 + l - 1 > n) then /* first suffix is too short */
        return failure;
else if (S2 + l - 1 > n) then /* second suffix is too short */
        return failure;
end
for i ← l - 1 downto 0 do
        if (T[S1 + i] ≠ T[S2 + i]) then /* mismatching character encountered */
                return failure;
        end
end
return success; /* all of the first l characters of the provided suffixes of T match */
```

After the partitions are successfully created, we perform their preprocessing. As stated before, the goal of this operation is to create the upper part of the suffix tree, so that the native suffix tree traversal from the root to an arbitrary branch is possible. At the same time, the suffix partitions are organized into the *evaluation stack*, which determines the order of their later processing, or evaluation. Some of the partitions, which consist only of a single suffix, might be completely evaluated during the preprocessing. In this case they are *not* included in the evaluation stack.

The preprocessing is performed by the function called `PreprocessPartitions`. At the beginning, it scans the entire range of partitions and determines which ones have to be evaluated immediately and which ones can be scheduled for later evaluation. During this process, some smaller ranges of partitions might be scheduled for repetitive preprocessing by pushing them into the stack of partition ranges, which still need to be partially evaluated. This function's goal is to empty this stack by partially evaluating all of the smaller partition ranges contained in there. Its pseudocode is very simple and basically consists only of two another function calls.

```
function PreprocessPartitions (ST, T, n, l, S, P, m, S)

Input:   ST — empty suffix tree on top of the text T
         T — entire text
         n — length of the text T
         l — length of the prefix shared by all the suffixes in each partition
         S — appropriately ordered array of starting positions of all the suffixes of T
         P — partitions of suffixes ordered lexicographically
         m — number of suffix partitions
Output:  ST — partially constructed suffix tree on top of the text T with its upper part created
         ES — stack of partitions to be completely evaluated later
         m — final number of suffixes in the stack of partitions S
Returns: preprocessing result

/* all the partitions in P are scheduled for partial evaluation */
ProcessPartitionRange(ST, T, n, l, S, P, 1, m + 1, ES, PRS);
/* partially evaluating the partition ranges in the partition range stack PRS */
EmptyPartitionRangeStack(ST, T, n, l, S, P, m, ES, PRS);
m ← number of suffixes in the stack S; /* setting the output parameter */
return success;
```

The most important action performed by this function is the separation of the partitions, which have to be evaluated completely during the preprocessing from the remaining partitions, which are evaluated only partially. Those partitions are scheduled for the later, complete evaluation in the carefully determined order. This entire operation is in fact performed by another function, namely `ProcessPartitionRange`.

This function at first checks, whether the number of partitions in the provided partition range is exactly one. If it is, this partition is further examined. In case it consists of a single suffix only, this partition is completely evaluated. It means that the leaf vertex corresponding to this suffix is output to the suffix tree. On the other hand, if this partition consists of more than one suffix, it is simply scheduled for the later, complete evaluation.

If the provided partition range consists of at least two partitions, we need to examine all of them. In case we find out that all of the leaves corresponding to the suffixes in all of the partitions in some smaller partition range share a single ancestor vertex, we output it to the suffix tree. Then, we push this range of partitions into the stack of the partition ranges, which still need to be partially evaluated.

In the situation when the smaller range of partitions contains only a single partition, we have to proceed similarly as before. It means that we have to check, whether the partition in question contains only a single suffix. In this case, the leaf vertex corresponding to this suffix is output to the suffix tree. On the other hand, if there is more than one suffix present in this partition, the explicit internal vertex, which is the common ancestor of these suffixes, is output to the suffix tree instead. Then we schedule this partition for the later, complete evaluation.

After the smaller partition ranges are pushed into the stack, they have to be further processed. This is managed by another function described later.

The function `ProcessPartitionRange` can be called multiple times to process the smaller partition ranges, until the entire partition range stack is empty and no more partition ranges need to be processed. The following is a pseudocode of this function.

```
function ProcessPartitionRange (ST, T, n, l, S, P, b, e, ES, PRS)

Input:   ST — partially constructed suffix tree on top of the text T
         T — entire text
         n — length of the text T
         l — length of the prefix shared by all the suffixes in each partition
         S — appropriately ordered array of starting positions of all the suffixes of T
         P — partitions of suffixes ordered lexicographically
         b — beginning of the partition range to be processed
         e — end of the partition range to be processed
         ES — stack of partitions to be completely evaluated later
         PRS — stack of partition ranges
Output:  ST — suffix tree possibly containing some new portions of its upper part
         ES — possibly expanded stack of partitions to be completely evaluated later
         PRS — possibly expanded stack of partition ranges to be further processed
Returns: processing result

if (e - b == 1) then /* if the provided partition range consists only of a single partition */
        if (partition P[b] contains only one suffix) then
                output the corresponding leaf vertex to ST;
        else /* partition P[b] contains more than one suffix */
                push the partition P[b] to the stack ES;
        end
else /* provided partition range consists of more than one partition */
        oldc ← last character of the lcp of all the suffixes in the partition P[b];
        /* sequentially examining each partition */
        for i ← b + 1 to e do
                c ← last character of the lcp of all the suffixes in the partition P[i];
                if (oldc ≠ c) then
                        if (b + 1 == i) then /* partition sub-range of size 1 */
                                if (partition P[b] contains only one suffix) then
                                        output the corresponding leaf vertex to ST;
                                else /* partition P[b] contains more than one suffix */
                                        output the explicit internal vertex to ST;
                                        push the partition P[b] to the stack ES;
                                end
                        else /* partition sub-range of size more than one */
                                output the corresponding explicit internal vertex to ST;
                                push the partition range P[b] ...P[i - 1] to the stack
                                PRS;
                        end
                        b ← i; /* setting the beginning of the next partition sub-range */
                        oldc ← c; /* remembering the current value of the c variable */
                end
        end
end
return success;
```

The term "lcp" used in the pseudocode of this function refers to the longest common prefix. It is used to determine the depth of an explicit internal vertex, which needs to be output into the suffix tree. There are two possible situations, when this is necessary.

First, when the current partition range consists of more than one partition, the new explicit internal vertex needs to be created. It is output to the suffix tree as the common ancestor of all the suffixes in all the partitions in this partition range. Second, when the partition sub-range consists of a single partition only, which contains more than one suffix, the new explicit internal vertex is created as well. In this case, it is output into the suffix

tree as the common ancestor of all the suffixes in this partition. In both of these situations the depth of the explicit internal vertex created is determined using the longest common prefix of all the respective suffixes.

The following function is used to partially evaluate the stack of partition ranges. After this stack is initially populated by the function `ProcessPartitionRange`, we use this function to process it and therefore to empty it. During this process, it is possible that some partition ranges are also added to this stack. But we continue to process its entries until it becomes empty. This function can then finish successfully. Its pseudocode looks like this:

```
function EmptyPartitionRangeStack (ST, T, n, l, S, P, m, ES, PRS)

Input:   ST — partially constructed suffix tree on top of the text T
         T — entire text
         n — length of the text T
         l — length of the prefix shared by all the suffixes in each partition
         S — appropriately ordered array of starting positions of all the suffixes of T
         P — partitions of suffixes ordered lexicographically
         m — number of suffix partitions
         ES — stack of partitions to be completely evaluated later
         PRS — partition range stack to be processed (and therefore emptied)
Output:  ST — suffix tree possibly containing some new portions of its upper part
         ES — possibly expanded stack of partitions to be completely evaluated later
Returns: emptying result

while (PRS is not empty) do
        b ← beginning of the partition range at the top of the stack PRS;
        e ← end of the partition range at the top of the stack PRS;
        PRS.pop(); /* removing this partition range from the top of the stack PRS */
        ProcessPartitionRange(ST, T, n, l, S, P, b, e, ES, PRS); /* and processing it
        */
end
return success;
```

In short, this function simply processes the entries of the stack of the partition ranges until it is empty. At first, it pops the partition range from the top of the stack. Then, it processes it using the function `ProcessPartitionRange`. During this processing, a partial evaluation of this partition range is done. Of course, this might result in pushing another, smaller partition ranges into the partition range stack. These steps are iteratively repeated until the partition range stack is empty.

This concludes the description of the partitioning part of the PWOTD suffix tree construction algorithm. At its end, the upper part of the suffix tree is constructed and the stack of partitions, which need to be completely evaluated is prepared. Their evaluation is the subject of the second part of the PWOTD algorithm.

3.4.2 Evaluating the partitions

The partitions are processed and therefore evaluated in the order specified by their position in the stack created during the partitioning part of the PWOTD algorithm. Each partition is evaluated separately using the function `ProcessPartition`.

The partitioning part ensures that all the partitions consisting only of a single suffix are already completely evaluated. This means that in this function, we are not dealing with single-suffix partitions and we can focus on the partitions, which contain more than one suffix.

At the beginning, we have to find the longest common prefix of all the suffixes present in the specified partition. Then, all these suffixes are ordered lexicographically according to the character just after their longest common prefix. When this is done, the suffix tree construction itself can begin.

The suffixes are scanned in the lexicographically ascending order. When a range of at least two suffixes having the same character after their longest common prefix is encountered, we output the explicit internal vertex, which is the ancestor of the leaves corresponding to these suffixes. Moreover, we also have to schedule the range of suffixes in question for the later evaluation, because they haven't been completely evaluated yet. We do it by pushing it to the stack of the yet unevaluated partition ranges.

On the other hand, if there is a suffix, which has a different character after the longest common prefix, than both of its neighboring suffixes, we output a leaf vertex corresponding to this suffix. Then we can safely proceed and scan the next suffix.

When all the suffixes are scanned and the corresponding explicit internal vertices and leaf vertices are created, we can proceed to process the yet unevaluated suffix ranges. But since the partition itself is just a range of suffixes, it can be performed using the same function. So, at the end, we just make the stack of unevaluated suffix ranges empty. The pseudocode of this function looks like this:

```
function ProcessPartition (ST, T, n, l, S, P)

Input:   ST — partially constructed suffix tree on top of the text T
         T — entire text
         n — length of the text T
         l — length of the prefix used during the partitioning
         S — appropriately ordered array of starting positions of all the suffixes of T
         P — partition containing the suffixes of T to be processed
Output:  ST — suffix tree also containing all the suffixes present in the specified partition
         S — more appropriately ordered array of starting positions of all the suffixes of T
Returns: processing result

SRS ← []; /* empty stack of the unevaluated ranges of suffixes */
update l to the length of the "lcp" shared by all the suffixes in the specified partition;
sort all the suffixes in the partition P according to their l^th character;
k ← number of suffixes in the provided partition P;
OutputNodes(ST, T, n, l, S, 1, k, SRS); /* output the appropriate vertices to */ ST;
EmptyStack(ST, T, n, l, S, SRS); /* evaluating all the remaining ranges of suffixes */
return success;
```

The vertices are output to the suffix tree using the function `OutputNodes`. To make the terminology simpler, we call the explicit internal vertices the *branching* vertices.

The function `OutputNodes` needs to output both the branching vertices and the leaves.

As we have mentioned previously, the branching vertex is encountered whenever there is a continuous range of suffixes, which share the same character at the position just after their longest common prefix. In this case, not only the branching vertex is output to the suffix tree, but also the partition range in question is scheduled to be evaluated again. Before the branching vertex can be output, we have to determine the length of the longest common prefix of all the suffixes in the current range. It is then used in the suffix tree to determine the length of the edge label incoming to this branching vertex.

Since the length of the longest common prefix of all its suffixes has indeed increased from the original value holding for the entire range of suffixes, it is certain that the suffix range is processed in a different way than it has been previously. This ensures that the evaluation never enters an infinite loop.

Alternatively, when there is a single suffix, which has different character than any other suffix in the partition at the position just after their longest common prefix. In this case, the corresponding leaf is output into the suffix tree, but no more suffix ranges are scheduled for the later evaluation. This time, the suffix is evaluated completely.

The pseudocode of this function looks like this:

```
function OutputNodes (ST, T, n, l, S, b, e, SRS)

Input:   ST — partially constructed suffix tree on top of the text T
         T — entire text
         n — length of the text T
         l — length of the prefix used during the partitioning
         S — appropriately ordered array of starting positions of all the suffixes of T
         b — beginning of the range of suffixes in S to be processed
         e — end of the range of suffixes in S to be processed
         SRS — stack of the partition ranges
Output:  ST — suffix tree also containing all the suffixes present in the specified partition
         S — more appropriately ordered array of starting positions of all the suffixes of T
         SRS — updated stack of the partition ranges
Returns: outputting result

oldc ← first character of suffix S[b] after the lcp of all the suffixes in the provided range;
/* scan the provided range of suffixes */
for i ← b + 1 to e do
        c ← first character of suffix S[i] after the lcp of all the suffixes in the provided range;
        if (oldc ≠ c) then
                if (b + 1 == i) then /* suffix range of size 1 */
                        output the corresponding leaf vertex to ST;
                else /* suffix range of size more than one */
                        output the explicit internal vertex to ST;
                        push the suffix range S[b] ...S[i - 1] to the stack SRS;
                end
        end
        b ← i; /* setting the beginning of the next suffix sub-range */
        oldc ← c; /* remembering the current value of the c variable */
end
return success;
```

This function therefore partially processes the entire range of suffixes. In some situations, the provided range can also be processed completely. The parts which are not yet completely processed are pushed into the stack of yet unevaluated suffix ranges, where they wait for their later processing. And this is handled by the following function called

`EmptyStack`.

It simply processes the stack by partially evaluating all of its entries using the function `OutputNodes` and stops when this stack is empty.

```
function EmptyStack (ST, T, n, l, S, SRS)

Input:   ST — partially constructed suffix tree on top of the text T
         T — entire text
         n — length of the text T
         l — length of the prefix used during the partitioning
         S — appropriately ordered array of starting positions of all the suffixes of T
         SRS — stack of the ranges of suffixes to be processed, or evaluated
Output:  ST — suffix tree also containing all the suffixes present in all the partitions of the stack S
Returns: emptying result

while (stack S is not empty) do
        b ← beginning of the suffix range at the top of the stack S;
        e ← end of the suffix range at the top of the stack S;
        l ← "lcp" of all the suffixes in this range;
        sort this range of suffixes according to their l^th character
        OutputNodes(ST, T, n, l, S, b, e, SRS);
end
return success;
```

Chapter 4

Implementation details

When it is necessary to implement these algorithms in practice, it is not enough to just present the idea of an algorithm. The implementation details are needed and must be provided as well. This chapter tries to address this need.

There are two types of suffix tree construction algorithms presented in this thesis. The first or the traditional one is represented by the McCreight's [McC76] and the Ukkonen's [Ukk95] algorithms. The second one is represented by the PWOTD algorithm [THP04]. Each of these algorithm types is designed for a different implementation technique.

In addition, the variation of the Ukkonen's algorithm for the suffix tree construction over the *sliding window* requires additional implementation enhancements. This means that is is necessary to use a *different* implementation technique for each of the previously mentioned types and variations of the suffix tree construction algorithms. They are all characterized in this chapter.

Before we proceed to the particular implementation techniques, we present some simple enhancements, which are used in all the algorithms in order to either simplify them or to improve their performance.

The first of these enhancements consists of terminating the input text with a *terminating character* (Definition 2.31). It is used in all the algorithms presented in this thesis, which create the suffix tree over the *entire* text. Naturally, this enhancement is not used when constructing the suffix tree over the *sliding window*. The reason is that it would make very little sense, as the initial parts of the text would still remain unchanged and only the substrings containing the last, terminating character would be affected.

The termination of the text with a terminating character ensures that every suffix of the text on top of which the suffix tree is built corresponds to a leaf. Moreover, as the Theorem 2.2 shows, *every* leaf in the suffix tree corresponds to a suffix. In conclusion, this simple enhancement introduces a one-to-one correspondence between the suffixes of the text and the leaves in the suffix tree.

Next enhancement specifies the ordering of the children of any parent vertex. They are ordered lexicographically according to the labels of the edges between them and their parent. Thanks to the *branching* requirement (Definition 2.20), it is always sufficient to consider only the first character of each edge label when ordering. All the suffix tree implementation techniques described in this thesis, which allow the children to be ordered, use this enhancement.

The main reason why the edge ordering is used is to make the edge selection a little faster. In some implementation techniques, all the children of a parent are organized in a list. Without this enhancement, it would be necessary to scan the entire list in order

to select the appropriate edge. But if the children are ordered, it is possible to stop the scanning sooner, as soon as its alleged position is passed.

4.1 Simple linked list

The first of the implementation techniques we are going to describe is called the Simple Linked List Implementation technique, or SLLI. It can be directly used with the McCreight's or Ukkonen's algorithms. With some minor modifications, it can also be used when constructing the suffix tree over a sliding window, as described in the section 4.3.

This implementation technique has been proposed by Kurtz [Kur99] and as the name suggests, it utilizes the linked lists. In particular, they are used in the suffix tree representation to access the children of any explicit internal vertex. This means that the average-case time complexity of accessing the particular child of a parent is linear with respect to the number of children. Therefore, this implementation technique is not recommended when the expected average number of children is high (see Chapter 5 for details).

The suffix tree is represented by two tables. The first of them, called `tleaf` is used to represent the *leaves* in the suffix tree. The second table is called `tbranch` and is used to represent the *branching* vertices, or in other words the explicit internal vertices including the *root*, in the suffix tree. It is beneficial to represent these tables using arrays, which is also what we have decided to do in our implementation.

The table `tleaf` consists of *leaf records* while the table `tbranch` consists of *branching records*. A *leaf record* stores all the information necessary to describe a leaf. Similarly, a *branching record* stores all the information necessary to describe an explicit internal vertex or the root. Each explicit vertex in the suffix tree can therefore be represented by either of these two records.

In order to uniquely identify the particular explicit vertex, this implementation technique uses a clever numbering method. The branching vertices are numbered by *positive* integers starting from 1 and continuing to higher integers. The number associated with a particular branching vertex determines its offset in the table `tbranch`. In our implementation, the *root* is always associated with the number 1. This corresponds to the assigning strategy for the numbers of branching vertices, which states that every branching vertex is given the lowest positive integer available at the time of its creation. Since the root is always created as the first branching vertex, its number conforms to this rule perfectly.

Similarly, the leaves are numbered by *negative* integers starting from (-1) and continuing to lower integers. The number associated with a particular leaf is a *negation* of its offset in the table `tleaf`. At the same time, the offset of *every* leaf in the table `tleaf` denotes the index of the first character of the text suffix corresponding to the respective leaf. This numbering is correct, because of one-to-one correspondence between the leaves in the suffix tree and the suffixes of the text on top of which the suffix tree is constructed. Such a correspondence is ensured by the previously mentioned implementation enhancement, which always terminates the text with a terminating character. Moreover, the range of integers taken up by the leaf numbers is always continuous.

In this numbering method, 0 is never used as a number of any vertex. Despite that, this value is useful to indicate an invalid number of vertex. It also implies that the record at the offset of 0 in both tables is unused.

Now we describe a branching record in more detail.

Detail 4.1 (SLLI branching record) *A **branching record** used in the implementation technique SLLI consists of the following entries:*

1. first child 2. next brother 3. suffix link 4. depth 5. head position

The first three entries hold the vertex numbers of the following vertices: the first child of this branching vertex, its next brother and the target vertex of its suffix link, respectively. In case any of these entries is not relevant for the particular branching vertex, it is filled with 0 — the only integer, which cannot be a valid number of any vertex.

The next entry holds the depth of a branching vertex in the suffix tree. It is measured by the number of characters on the path from the *root* to this branching vertex. Therefore, the depth of the root is always zero.

Finally, the branching record contains the *head position* entry. It is just an index to the text of the first character of the substring which corresponds to this branching vertex in the suffix tree. Thanks to the *compactness* requirement (Definition 2.18), all the explicit internal vertices (but not the root) must have at least two children. It implies that there are at least two substrings corresponding to every branching vertex, except for the root. Therefore it is very important to introduce an exact rule according to which this substring is selected.

The root is a special case, because as stated in the Definition 2.24, it corresponds to the *empty* string. Consequently, its head position entry is unused and it is set to zero.

The other branching vertices, however, need to have the appropriate occurrence (Definition 2.7) of its corresponding text substring selected. We could simply select its *leftmost* occurrence. But as Kurtz [Kur99] shows, there is a less obvious, but more convenient way.

He points out that the reason of existence of the branching vertex is the leftmost "branching occurrence" of its corresponding string in the text. It can be defined like this:

Definition 4.1 *An occurrence O of the string S in the text T is called the **branching occurrence**, if the following holds:*

- *There is another occurrence O_{prev} of the same string S in the text T, which starts* before *the occurrence O.*
- *The characters immediately following the occurrences O and O_{prev} are* different.

The branching occurrence of the corresponding substring is available whenever a new branching vertex is created. Therefore it is equally easy to use it as it is to use e.g. the leftmost occurrence. But since Kurtz [Kur99] has chosen it, so have we.

Now we can summarize the previous paragraphs into a compact definition of the head position as used in the SLLI implementation technique.

Definition 4.2 *Suppose we have a suffix tree ST on top of the text T and its explicit internal vertex v corresponding to the substring S_v. A **head position** of the vertex v is an index h_v to the text T of the first character of the* leftmost *branching occurrence O_v of the substring S_v in the text T. That is $T[h_v] = O_v[1]$.*

The head position along with the depth of an explicit internal vertex are used together with the depth of its parent to determine the label of an edge leading from its parent to this vertex.

Now we describe a leaf record in more detail:

Detail 4.2 (SLLI leaf record) *A **leaf record** used in the implementation technique SLLI contains only a single entry referring to its* next brother.

As with the branching record, the *next brother* entry is set to 0 if the corresponding leaf does not have any next brother and therefore this entry is not relevant for it. In order to determine the label of the leaf's incoming edge, we need to know the index of the first character of its corresponding suffix, its depth and the depth of its parent.

It is easy to obtain the depth of its parent, because it is a branching vertex and its depth is simply a part of its description. It is also easy to obtain the index of the first character of the suffix corresponding to the specified leaf. Thanks to the selected numbering method, this index is the same as the offset of the corresponding leaf record in the table `tleaf`.

Finally, to obtain the depth of a leaf, it is necessary to know the current length of the text on top of which the suffix tree is constructed. This information is maintained by all of the implementation techniques used in this thesis and it is easily available. The depth of a leaf is therefore obtained by subtracting the index of the first character of the suffix corresponding to this leaf, decreased by one, from the total length of the text. In conclusion, it is sufficient for a leaf record to contain only a single entry — the next brother of the represented leaf.

Each vertex number used in this implementation technique can be represented by a single *signed* integer. Suppose that 32-bit integers are used. In the most common case, the negative numbers are represented using the *two's complement*, which means that the 32-bit signed integers have the range from -2^{31} to $2^{31}-1$. This implies that each of them can store $2^{31}-1$ different branching vertex numbers or 2^{31} different leaf numbers. The maximum number of branching vertices including the root is almost always by one less than the number of leaves (see Theorem 2.1). The only exception is when the root does *not* satisfy the *compactness* requirement. In this case, the entire suffix tree consists only of the root and exactly one leaf.

The reason is that a suffix tree must contain *all* the substrings of a given text. For a contradiction, suppose that there exists a suffix tree which contains more than one leaf and at the same time its root does not satisfy the compactness requirement. This means that the suffixes corresponding to these leaves must share a common, nonempty prefix. If we remove this prefix from these suffixes, we obtain at least two text substrings which must also be present in this suffix tree. Thanks to the *branching* requirement, these substrings start with a *different* character. And this means that the root must have at least two children and therefore it must satisfy the compactness requirement as well.

In conclusion, it is not possible for a suffix tree containing at most 2^{31} leaves to contain more than $2^{31}-1$ branching vertices. Therefore it is possible to utilize all the vertex numbers available in this representation.

Considering the number of available leaf numbers, this translates into the maximum allowed text length of $2^{31} = 2\,147\,483\,648$ characters. For the purposes of in-memory suffix tree construction in this thesis, it is more than sufficient. The reason is that it is

impossible to create a suffix tree on top of the text of this length using any of the presented implementation techniques even on a computer with 16 GiB of memory. Therefore, it is sufficient to use 32-bit integers.

The *depth* and the *head position* entries of the branching record can each be represented by a single *unsigned* integer. We would like to maintain the limitation on the maximum length of the text imposed by the representation of the vertex numbers. For that reason, we have to ensure that the depth and the head position of all the branching vertices in any suffix tree on top of the text of such length can be stored in our representation.

The maximum depth of a branching vertex in the suffix tree on top of the text of length 2^{31} is $2^{31} - 1$. The largest head position of the branching vertex in such a suffix tree is the same as the text length, namely 2^{31}. This means that 32-bit unsigned integer is sufficient to store all the required values of these entries.

Consequently, a branching record can be represented by *five* 32-bit integers, or 20 bytes, while a leaf record can be represented by a *single* 32-bit integer, or 4 bytes. The worst-case space complexity of this implementation technique can therefore be summarized like this:

Detail 4.3 (SLLI worst-case space complexity) *A* leaf record *occupies exactly* 4 *bytes and a* branching record *occupies exactly* 20 *bytes.*

For a text of length $n > 1$, there can be at most n leaves and therefore at most $n - 1$ branching vertices, including the root. This means that the worst-case space complexity of this implementation technique is $n \cdot 4 + (n - 1) \cdot 20 = 24 \cdot n - 20$ bytes, which is almost 24 *bytes per each text character.*

The main advantage of this implementation technique is its relatively low space complexity, compared to the almost all of the other implementation techniques presented in this thesis. It is also very easy to implement. However, as we have mentioned before, it has one considerable disadvantage as well. Due to the nature of the linked lists, the operation of selecting the appropriate child of the specific parent has the worst-case time complexity *linear* with respect to the number of its children. This can considerably worsen the time complexity of the entire suffix tree construction, most notably when the expected average number of children is high.

If the *bottom-up* suffix link simulation is used, there are additional requirements for this implementation technique. In particular, this kind of suffix link simulation requires that it is possible to directly access a parent of each vertex. This is easy to implement by adding another entry called *parent* to both the leaf record and the branching record. This entry contains the number of parent of the vertex represented by the particular record.

A parent of each vertex is a *branching* vertex, which means that its number is always positive. The only exception is the root, which does not have a parent and whose parent entry is unused and set to zero. As a result, the parent number can be stored in an *unsigned* integer which is large enough to contain all the possible numbers of branching vertices. Therefore, it is possible to use a 32-bit unsigned integer. But so far, we have used only signed integers for the numbers of vertices. For this reason and because it is possible, we rather use 32-bit *signed* integers also for the representation of the numbers of parents. In addition, using the same data type for all the vertex numbers means that we definitely avoid possible unnecessary typecasting.

The parent entries, however, increase the space requirements of the entire suffix tree representation. In particular, the sizes of leaf record and the branching record increase by 4

bytes each. Consequently, the worst-case space complexity of this modified implementation technique increases as well and it can be summarized like this:

Detail 4.4 (SLLI with parent pointers worst-case space complexity) *A* leaf record *occupies exactly* 8 *bytes and a* branching record *occupies exactly* 24 *bytes.*

For a text of length $n > 1$*, there can be at most* n *leaves and therefore at most* $n - 1$ *branching vertices, including the root. This means that the worst-case space complexity of this implementation technique is* $n \cdot 8 + (n - 1) \cdot 24 = 32 \cdot n - 24$ *bytes, which is almost* 32 *bytes per each text character.*

4.2 Simple hash table

Next implementation technique has also been presented by Kurtz [Kur99] and it is called the Simple Hash Table Implementation technique, or SHTI. Similarly to SLLI, it can be directly used with either the McCreight's or the Ukkonen's algorithm. When slightly modified, it can also be used when constructing a suffix tree over a sliding window (see the section 4.3 for more details).

As the name suggests, this implementation technique utilizes a hash table. In particular, it is used to access the children of any explicit internal vertex. The hash table operations need to be fast enough, so that the overall asymptotic time complexity of the suffix tree construction is not affected. This is mostly dependent on the technique used to resolve the *hash collisions* in the hash table. A hash collision is a situation in which the hash functions determine the same hash table location for two *different* keys.

Kurtz proposes to use the *double hashing*, which is a kind of an *open addressing* collision resolution technique. He argues that it is more space efficient than e.g. the chaining techniques, which require additional pointers. Probably the greatest advantage of the double hashing is its ability to utilize the *entire* hash table. That's also why we have decided to implement it.

Moreover, we have also implemented another kind of an open addressing collision resolution technique called the *cuckoo hashing* (see Pagh and Rodler [PR01] for more information). Similarly to the double hashing, it offers very high space utilization, while at the same time it ensures *constant* lookup time in the worst case. However, in this thesis we will not explain the details of any of these collision resolution techniques.

The SHTI implementation technique represents the suffix tree using two tables. The first of them, called `tbranch`, stores the branching vertices in a way similar to the table `tbranch` used in the SLLI implementation technique. The second table is the hash table itself and it is called `tedge`, because its purpose is to store all the edges in a suffix tree.

In contrast with the SLLI implementation technique, the leaves are represented only *implicitly*. It means that there is no table `tleaf` which would store some information about the leaves. The reason is that it is not necessary, because all the information about the leaves can be retrieved from the table `tbranch` or the table `tedge`.

Table `tbranch` consists of *branching records*, which are similar to the branching records used in the SLLI implementation technique. However, since the edges are represented by the table `tedge`, the branching records do not have to contain any information describing the parent–child relationships between the vertices. This means that the branching record

entries containing the numbers of the first child or the next brother used in the implementation technique SLLI are not necessary. Therefore, a branching record used here is smaller and contains the following entries:

Detail 4.5 (SHTI branching record) *A* ***branching record*** *used in the implementation technique SHTI consists of the following entries:*

1. depth 2. head position 3. suffix link

The entries contained in this branching record form a subset of the entries contained in the branching record used in the implementation technique SLLI. Their meaning remains the same. In particular, the first entry holds the depth of a vertex represented by this branching record. The second entry holds the head position of this vertex. And finally, the third entry holds the number of vertex which is the target vertex of a suffix link starting at the vertex represented by this branching record. The method for numbering the vertices used in this implementation technique is the same as the method used in the implementation technique SLLI.

Each of the entries in this branching record can be represented in the same way as the corresponding entries contained in the branching record used in the implementation technique SLLI. Namely, the depth and the head position can each be represented by a single 32-bit *unsigned* integer and the suffix link can be represented by a single 32-bit *signed* integer. The usage of signed integers for the representation of the numbers of branching vertices implies that their number must not exceed $2^{31} - 1$.

The table `tedge` consists of *edge records* which represent the edges in a suffix tree. Each edge record represents a single edge. For every edge in a suffix tree, there is exactly one edge record present in the table `tedge`. It contains the following entries:

Detail 4.6 (SHTI edge record) *An* ***edge record*** *used in the implementation technique SHTI consists of two entries referring to the* source vertex *and the* target vertex *of the represented edge.*

Both entries in the edge record are represented by the numbers of the corresponding vertices. In order to access a particular edge record in the table `tedge`, its location must be known. Since the table `tedge` is a *hash* table, the location of a particular entry can only be determined using hashing.

As we have already mentioned, we use a hash table with an open addressing collision resolution technique. It means that in case of a hash collision, alternative locations of the desired entry in the hash table are determined by additional *probing*. We have implemented two kinds of probing strategies — the *double* hashing and the *cuckoo* hashing. Each of them uses a different kind of hash functions and features its own algorithms for entry lookup, insertion and deletion. More details on how exactly these collision resolution techniques are implemented can be found in the source code but as stated earlier, their details are not discussed in this thesis.

The location (or address) of a particular entry in the hash table is determined by a hash function and the corresponding *key*. The hash function is determined by the selected probing strategy and the key can be any unique identification of the hash table entry.

In our case, the hash table consists of the edge records which represent the edges in

a suffix tree. Kurtz [Kur99] proposed that a key can be formed by combining a head position of the edge's source vertex and the first character of its label. Since the source vertex of an edge is always a branching vertex, such a key is well defined. In "Reducing the space requirement of suffix trees" [Kur99], Observation 5.1, Kurtz shows that a head position is unique for every branching vertex. Considering also the *branching* requirement (Definition 2.20), it means that such a key can be used to uniquely identify every edge record in the table `tedge`.

However, since the head position of the edge's source vertex is accessible only *indirectly*, we have decided to slightly modify the Kurtz's approach. We have decided to use the number of the edge's source vertex *directly* instead of its head position. In our implementation, the key used by the hash functions is formed by combining the number of the edge's source vertex and the first character of its label. Again, thanks to the *branching* requirement, such a key uniquely identifies the appropriate edge record.

It is important to explain how exactly a number of branching vertex and a character can be combined into a key which can then be used by a hashing function. The number of edge's source vertex is stored in a single 32-bit *signed* integer. But since it always refers to a *branching* vertex, it must be positive. Therefore, only 31 bits are necessary for its representation.

The number of bits necessary to represent the first character of the edge's label is dependent on the internal character encoding. By default, our implementation uses ASCII as an internal character encoding and each character is represented by C data type `char`. It is an integral data type which might be signed or unsigned, depending on platform, compiler and its settings. But it is not important whether this data type is signed or unsigned. The most important is its size, because it limits the maximum supported size of the alphabet. On most platforms, the size of `char` is 8 bits, which translates into the maximum supported alphabet size of $2^8 = 256$.

However, it is also possible to enable the support for *wide characters*. It can be done at compile time by uncommenting the definition of macro `SUFFIX_TREE_TEXT_WIDE_CHAR` at line 33 in the header file `common/h/suffix_tree_common.h`. This setting changes the internal data type used to represent the individual characters from `char` to `wchar_t`. At the same time, the internal character encoding is changed as well to reflect the possibly increased maximum alphabet size. It might be changed to UCS-4, UCS-2 or it can remain unchanged at ASCII, depending on the size of `wchar_t`, which is platform-dependent.

On unix systems, its size is typically 32 bits, which translates into the theoretical maximum alphabet size of 2^{32} characters. The total number of all the traditional characters in the most recent version of Unicode is 110 116 (according to *The Unicode Standard, Version 6.1.0*, [Con12]). Therefore, if the size of `wchar_t` is 32 bits, its range is more than sufficient to contain *any* Unicode character. In this case, the internal character encoding is changed to UCS-4 which can encode all the Unicode code points. If the size of `wchar_t` is only 16 bits, then its range is $2^{16} = 65\,536$ and the internal character encoding is changed to UCS-2 which supports the entire Basic Multilingual Plane of Unicode only.

The ability to use Unicode is one of the extra features provided by our implementation of all the suffix tree implementation techniques presented in this thesis. We have added the Unicode support in order to be able to handle the texts which use the alphabets of size more than 256 characters. Most of the alternative suffix tree implementation techniques do not provide any means for handling such texts.

To summarize the previous paragraphs, we can conclude that the first character of any edge's label can be represented by at most 32 bits. Together with 31 bits necessary to

represent the edge's source vertex, we obtain 63 bits. Therefore, both of these values can be represented by a single 64-bit integer. In particular, we use the C data type `unsigned long long`, which is an *unsigned* integer of size *at least* 64 bits.

The key corresponding to the particular edge record is created simply by concatenating the bit representation of the first character of the corresponding edge's label and the source vertex contained in this edge record. Hence, the hash functions must be able to provide a mapping from 64-bit address space to a much smaller, hash-table-sized address space. Their implementation is not discussed in this thesis.

Now we estimate the maximum number of edges a suffix tree can have, so that we can impose some limits on the maximum size of the hash table `tnode`. In section 4.1, we have shown that a suffix tree containing $l > 1$ leaves can have at most $l - 1$ branching vertices. This means that a suffix tree on top of the text of length $l > 1$ can contain at most $2l - 1$ explicit vertices of any kind.

The number of edges in a suffix tree is by one less than the total number of its vertices. This is a general property from the graph theory which holds for *every* tree. Therefore, a suffix tree on top of the text of length $l > 1$ can have at most $2l - 2$ edges.

However, the actual size of the hash table `tnode` is determined not only by the theoretical maximum number entries which are expected to be inserted into it. Usually, not every position in the hash table is used. The ratio of the number of occupied hash table entries to the number of all its entries is called the hash table *load factor*. As more and more entries are inserted into the hash table, its load factor increases.

Generally, the speed of hash table operations decreases when the hash table load factor increases. This is the reason why it is advisable to have a hash table whose size is *larger* than the actual expected number of its occupied entries. By increasing the hash table size, we effectively increase the speed of its operations at the cost of higher memory requirements.

Another important fact to consider is that the collision resolution techniques are not perfect. There can be a hash collision, which they are unable to resolve. In this case, the entire hash table must be rehashed into a new, larger hash table, using new hash functions. This operation is very time consuming and therefore it can slow down the entire suffix tree construction if it is performed too often. Moreover, since the rehashing accesses two hash tables of similar size at the same time, the hash-table-related peak memory requirements are almost twice as high as without the rehashing.

Now we analyze the theoretical worst-case space complexity of this implementation technique. As shown in section 4.1, the numbers of vertices can be represented using a single 32-bit signed integer which occupies 4 bytes. Similarly, the same section also shows that the depth and the head position of a branching vertex can each be represented by a single 32-bit unsigned integer, occupying 4 bytes as well. This means that the entire branching record can be represented using 12 bytes and the entire edge record can be represented using 8 bytes.

Suppose that the hash table load factor is 1, which means that no extra space is used to represent the empty hash table entries. Despite the fact that it is very unlikely for such a situation to happen, it is nevertheless possible to experience it. This simplified assumption enables us to estimate the following:

Detail 4.7 (SHTI worst-case space complexity) *A* branching record *occupies exactly* 12 *bytes and an* edge record *occupies exactly* 8 *bytes.*

For a text of length n, there can be at most n leaves and therefore at most $n-1$ branching vertices, including the root. The number of edges in such a suffix tree is $n+(n-1)-1=2n-2$. This means that the worst-case space complexity of this implementation technique is $(n-1)\cdot 12+(2n-2)\cdot 8=28\cdot n-28$ bytes, which is almost 28 *bytes per each text character.*

The space requirements of this implementation technique are therefore higher than the space requirements of the implementation technique SLLI. This disadvantage is balanced by increased speed of accessing the desired child of any parent vertex. The hash table `tedge` along with either of the collision resolution techniques we have implemented allows us to find any particular edge in *constant* amortized time. Therefore, even when the expected average number of children a single vertex has is high, the time complexity of this implementation technique remains almost the same. This is its main advantage. As a result, we can effectively use this implementation technique to construct suffix trees on top of the texts which use large alphabets.

The *bottom-up* suffix link simulation can also be used in combination with this implementation technique, provided that some additional requirements are met. We have already mentioned that bottom-up suffix link simulation needs to directly access a parent of each vertex. This can be implemented similarly to the way it is implemented in the implementation technique SLLI. In particular, a parent entry is added to the branching record. It represents the number of parent of the corresponding branching vertex. Also, a new table `tleaf` of leaf records is created. Its only purpose is to store the references to the parents of the leaves. This implies that the leaf record contains only one entry — the number of the corresponding leaf's parent.

Each of the parent entries can be represented by a single 32-bit signed integer which occupies 4 bytes. This means that the size of the branching record increases to 16 bytes. Also, a new table `tleaf` consisting of leaf records is added to the suffix tree representation. Its leaf record occupies 4 bytes. These changes increase the space requirements of this modified implementation technique in the following way:

Detail 4.8 (SHTI with parent pointers worst-case space complexity) *A* branching record *occupies exactly* 16 *bytes, a* leaf record *occupies exactly* 4 *bytes and an* edge record *occupies exactly* 8 *bytes.*

For a text of length n, there can be at most n leaves and therefore at most $n-1$ branching vertices, including the root. The number of edges in such a suffix tree is $n+(n-1)-1=2n-2$. This means that the worst-case space complexity of this implementation technique is $(n-1)\cdot 16+n\cdot 4+(2n-2)\cdot 8=36\cdot n-32$ bytes, which is almost 36 *bytes per each text character.*

4.3 Sliding window

The implementation of the suffix tree construction over a sliding window requires a suffix tree representation which provides a reference to the parent of each vertex. The reason is that in order to perform the deletion of the the longest suffix (Algorithm 3.3), these references are necessary. As we have already shown, both the SLLI and the SHTI implementation techniques can be modified so that they provide the desired parent references. Therefore, each of them can be used as a base of the suffix tree implementation technique suitable for a sliding window.

However, they need to be further adjusted in order to meet all the requirements which arise when constructing a suffix tree over a sliding window. At first, this type of suffix tree construction requires that any explicit vertex except for the root can be *deleted* from a suffix tree. This means that we have to introduce an improved vertex numbering method which allows the numbers of deleted leaves or branching vertices to be reused.

Originally, leaf number is a *negation* of an index to the text of the first character of the leaf's corresponding suffix. During the suffix tree construction over a sliding window, only a certain part of the entire text is available at a time. Despite that, it is possible to use an index to the entire text which is *inside* the current sliding window. However, this would require that the size of the table `tleaf` were the same as the size of the *entire* text. And this is not acceptable.

For that reason, we modify the leaf numbering method so that the leaf number is a negation of an index to the text of the first character of the leaf's corresponding suffix, *modulo* the maximum allowed size of a sliding window. This ensures that the size of the table `tleaf` is the same as the maximum allowed number of leaves in a suffix tree.

This modification allows the number of the most recently deleted leaf (which corresponds to the longest suffix currently present in the suffix tree) to be immediately reused. It can be assigned to a leaf corresponding to the *shortest* suffix that can possibly be present in the suffix tree over the current sliding window.

On the other hand, this modified leaf numbering method also has one disadvantage. It is not as easy to obtain the text index of the first character of the suffix corresponding to a particular leaf as it was when using the original leaf numbering method. The reason is that a leaf number itself is not sufficient for obtaining a text index of the first character of the corresponding suffix. To be able to determine it, it is also necessary to know the current position of the sliding window within the entire text.

The numbers of branching vertices in the table `tbranch` are determined by the order of their creation. Provided that we would like to reuse the numbers of deleted branching vertices, we have to change their allocation strategy. Therefore, when inserting a new branching vertex into the table `tbranch`, we have to look for empty positions in the *entire* table, not only in the previously unused part.

Next issue arises when deleting an edge from the suffix tree. If the implementation technique SLLI is used, it can be achieved easily. The reason is that the edges are not represented explicitly, so it is sufficient to remove the appropriate vertices and all the edges starting or ending at them will be implicitly removed as well.

If the implementation technique SHTI is used, the edges are represented explicitly and we must be able to perform their deletion. Whenever an edge is deleted, it is necessary to remove its corresponding edge record from the hash table `tedge`. Originally, this hash table does not have to support deletions. But if it is used in the suffix tree construction over a sliding window, its ability to effectively handle the delete operations is essential.

Both of the collision resolution techniques described in section 4.2 can be adapted to support deletions. It is important to note that the *cuckoo* hashing is able to handle the deletions exceptionally well. The worst-case time complexity of its delete operation is *constant*. Therefore, it is advisable to prefer this collision resolution technique to the double hashing when implementing the suffix tree construction over a sliding window.

However, there is another disadvantage of using SHTI implementation technique when constructing a suffix tree over a sliding window. After an edge is deleted, we must check whether its source vertex has at least two children remaining. There is only one way to perform this operation when using SHTI implementation technique.

We have to check whether the hash table contains an entry for *each* edge that can possibly start at the source vertex in question. This means checking for all the edges starting with all the characters which can possibly be represented by the current implementation settings. Since this is extremely time consuming, it makes the SHTI implementation technique practically unusable for the suffix tree construction over a sliding window. Despite that, we have implemented it as well.

Another implementation issue arises when the edge label maintenance method by Fiala and Greene (see subsection 3.3.1) is used. It requires an additional 1 bit of information for every branching vertex to store the *update bit*. Therefore, we have to provide a way to represent this bit inside a branching record. Fortunately, it is quite easy. Since the parent entry is represented by a signed integer and the parent number is *always* positive, the sign of this entry is unused. Therefore we can use it to represent the update bit. If the update bit is set to 1, we make the sign of this entry negative. Otherwise, we keep it positive.

The last issue with the implementation of the suffix tree construction over a sliding window is the representation of the sliding window itself. We have to introduce an effective method for replacing old characters in the sliding window with the new, more recent ones. These operation is essential for the movement of a sliding window.

We have decided to use a simple buffering method in which the sliding window is represented by a *circular* buffer divided into *blocks*. A block is the smallest part of the sliding window which can be replaced at a time. To replace a block in the sliding window means to replace all of its characters.

Only several sliding window blocks are accessible at a time. They contain the *active part* of the sliding window, which is in fact the entire text on top of which the current suffix tree is built. The remaining blocks are used for buffering. Every time the active part of a sliding window moves outside of a particular block, it is marked as old and scheduled for replacement. There is a special thread dedicated to replacing the old blocks by filling them with new characters. When a block is replaced, it is marked as ready so that the active part of the sliding window can be moved over it.

In conclusion, the suffix tree construction over a sliding window can be implemented using the slightly modified version of either the implementation technique SLLI with parent pointers or the implementation technique SHTI with parent pointers. The resulting space complexity of these modified implementation techniques is essentially the same as the space complexity of the original implementation techniques, because no new data structures have been added.

However, their time complexity is affected. The main reason is that the longest repeated suffix must be deleted from the suffix tree every time the sliding window is moved. Additional reasons include the new method for assigning the vertex numbers, which is more time-consuming, the necessity to perform the edge label maintenance and the sliding window buffering. In case of the modified SHTI implementation technique, its inability

to effectively lookup the remaining children of a branching vertex makes it significantly slower and much less suitable for this type of suffix tree construction. Moreover, the average speed of hash table operations might also be negatively affected when the hash table entries are allowed to be deleted.

All these facts increase the average-case time complexity of the suffix tree construction over a sliding window. But the main reason why it is used is not the speed of its construction compared to the speed of the traditional suffix tree construction, but its ability to construct and maintain a suffix tree over a sliding window. Such a suffix tree is smaller than a suffix tree built on top of the entire text, which means that the increased time complexity of its construction is exhibited only on the smaller scale.

4.4 Simple linear array

This section describes an implementation technique used to implement the Partition and Write Only Top Down, or simply PWOTD suffix tree construction algorithm (see section 3.4 for details). Similarly to the algorithm itself, this implementation technique has been introduced by Tata, Hankins, and Patel [THP04]. The authors did not give it any particular name, so we have decided use the name Simple Linear Array Implementation technique, or SLAI. It refers to the fact that this implementation technique stores the entire suffix tree in one table represented by a single array. Besides, this name fits the pattern shared by the names of all the other implementation techniques used in this thesis.

A table which is used to store the entire suffix tree in this implementation technique is called `tnode`, because it contains all the information about every vertex in a suffix tree. It is represented by a linear array of simple values called *node records*.

In this implementation technique, the root is represented only *implicitly*. Each leaf is represented by a *single* node record and each branching vertex is represented by *two* consecutive node records. Therefore, it is necessary to differentiate between the node records representing the leaves and the node records representing the branching vertices. For that reason, each node record contains a flag indicating whether it represents a leaf or a branching vertex. This allows the node records of any kind to be almost arbitrarily positioned within the entire table `tnode`, provided that any two node records representing the same branching vertex are always placed next to each other.

The fact that different numbers of node records are used to represent different kinds of vertices means that it is not possible to randomly access a location in the table `tnode` which represents the desired vertex. Nevertheless, it *is* possible to iteratively access all the children of a branching vertex, which is represented by the edge records whose location is known. It makes this implementation technique similar to the implementation techniques which use linked lists.

Now we describe a *node record* in more detail:

Detail 4.9 (SLAI node record) *A **node record** used in the implementation technique SLAI consists of the following entries:*

1. data 2. rightmost child flag 3. leaf flag

The first entry, data, is used for various purposes depending on whether the node record represents a leaf or a branching vertex. In case it represents a branching vertex,

the purpose of its data entry is further dependent on whether this node record is the first or the second of the two consecutive node records representing a branching vertex.

The second entry, rightmost child flag, indicates whether or not the vertex represented by this edge record is the *rightmost* child of its parent. If this flag is set, it means that the parent of the represented vertex does not have any more children ordered *after* this vertex. The children are ordered lexicographically according to the labels of the edges between them and their parent. It is the same ordering as used in the implementation technique SLLI (see section 4.1).

The last entry, leaf flag, is used to indicate whether a node record represents a leaf or a branching vertex. If this flag is set, it represents a leaf.

In our implementation, each node record is represented by a single 32-bit *unsigned* integer, occupying 4 bytes. Its most significant bit is used to represent the leaf flag and its second most significant bit is used to represent the rightmost child flag. This leaves the remaining 30 bits for the representation of the data entry.

If a node record represents a leaf (i.e. its leaf flag is set), its data entry contains an index to the text of the first character of the label of the corresponding leaf's incoming edge. This means that the maximum length of the text supported by this implementation technique is limited by the size of the node record's data entry to $2^{30} = 1\,073\,741\,824$.

As we have already mentioned, a branching vertex is represented by two consecutive node records. The data entry of the first of them contains an index to the text of the beginning of the label of an edge which ends at this branching vertex. This is similar to the way the data entry is used in node records corresponding to leaves.

The data entry of the second node record representing a branching vertex contains a position in the table `tnode` of the first child of this branching vertex. Since the size of this entry is limited to 2^{30}, it is also the maximum supported size of the table `tnode`.

The order of node records in the table `tnode` helps to determine the parent–child relationships between the vertices. In particular, all the node records representing all the children of a single branching vertex are placed sequentially in the table `tnode`. They are ordered lexicographically according to their incoming edge's label and the last one of them has its corresponding rightmost child flag set. This way, it is possible to determine where the node records representing the children of a particular branching vertex end. The overall placement of node records in the table `tnode` is determined by the order of evaluation of the particular parts of a suffix tree.

In addition to the table `tnode`, the suffix tree construction requires additional data structures — the temporary tables. In this thesis, we do not discuss the details on how these tables are used or how large they need to be. Tata, Hankins, and Patel [THP04] provide more exhaustive explanation of their usage and their size.

Considering the table `tnode` only, the space requirements of this suffix tree implementation technique can be summarized like this:

Detail 4.10 (SLAI worst-case space complexity) *A* node record *occupies exactly* 4 *bytes. In order to represent a leaf, one node record is necessary and in order to represent a branching vertex, two node records necessary.*

For a text of length n*, there can be at most* n *leaves and therefore at most* $n - 2$ *branching vertices, excluding the root. This means that the worst-case space complexity of this implementation technique is* $(n - 2) \cdot 2 \cdot 4 + n \cdot 4 = 12 \cdot n - 16$ *bytes, which is almost* 12 *bytes per each text character.*

Apparently, the space complexity of this implementation technique is considerably smaller than the space complexity of the other implementation techniques introduced in this thesis. But it is important to note that in order to construct a suffix tree using this implementation technique, additional temporary tables are necessary. However, the table `tnode` along with the text itself are sufficient to represent the entire suffix tree.

This implementation technique also has one notable disadvantage. Unlike the SLLI or SHTI implementation techniques, it does *not* support the updates to the suffix tree. As a result, once a suffix tree in this representation is fully constructed, it can not be further adjusted. To our knowledge, no method for modifying a suffix tree represented by this implementation technique has been published yet.

Chapter 5

Usage recommendations

Based on the theoretical description of the suffix tree construction algorithms presented in the previous chapters of this thesis, we now provide a simple analysis and recommendations on when it is advisable to use a particular algorithm and implementation technique and why.

At first, we focus on the theoretical worst-case analysis and we begin with the space complexity. The following table shows the theoretical worst-case space complexity of all the suffix tree implementation techniques described in this thesis.

Table 5.1: Worst case space complexity per each text character.

Parent pointers	Simple Linked List	Simple Hash Table	Simple Linear Array
no	24 bytes	28 bytes	12 bytes
yes	32 bytes	36 bytes	—

Considering only the worst-case space complexity, we can conclude that the most space-efficient of all the implementation techniques analyzed in this thesis is the Simple Linear Array Implementation technique presented by Tata, Hankins, and Patel [THP04]. However, as noted in section 4.4, it does not allow an already constructed suffix tree to be modified. If these modifications are necessary, another implementation technique must be used. The lowest space complexity of all the remaining implementation techniques is achieved by the Simple Linked List Implementation technique without the parent pointers, originally presented by Kurtz [Kur99].

In case parent pointers are necessary, for example because a suffix tree needs to be constructed and maintained over a sliding window, the lowest space complexity is also achieved by the SLLI implementation technique, this time with the parent pointers. In general, we can conclude that the implementation techniques which use the linked lists are more space efficient than the ones which use hashing.

The analysis of the theoretical worst-case time complexity is not very expressive, because under the real-life conditions it is *usually* not important. But nevertheless, we present it here in order to have this information available for reference. Table 5.2 shows the asymptotic worst-case time complexity of all the algorithms and their variations presented in this thesis. The top-down and bottom-up suffix link simulation used in the McCreight's and Ukkonen's algorithms are analyzed separately. Their analysis is not limited to any particular implementation technique.

Table 5.2: Asymptotic worst case time complexity on the text of length n.

McCreight's		Ukkonen's		PWOTD
top-down	bottom-up	top-down	bottom-up	
$\mathcal{O}(n \cdot b)$	$\mathcal{O}(n^2 \cdot b)$	$\mathcal{O}(n \cdot b)$	$\mathcal{O}(n^2 \cdot b)$	$\mathcal{O}(n^2)$

The worst-case time complexity of the bottom-up suffix link simulation has been estimated by Senft and Dvořák [SD12]. Similarly, Tata, Hankins, and Patel [THP04] determined the worst-case time complexity of the PWOTD algorithm. The worst-case time complexity of the suffix tree construction algorithms presented by Ukkonen [Ukk95] and McCreight [McC76] have been determined by their authors in the original papers.

The variable b used in the specification of the worst-case time complexity is a *branching factor*. It determines the worst-case time complexity of the selection of the appropriate child of a parent. When using the SLLI implementation technique, $b \in \mathcal{O}(|\Sigma|)$ where $|\Sigma|$ is the size of the text alphabet. However, when using the SHTI implementation technique, the value of branching factor is determined by the collision resolution technique in use. Usually, $b \in \mathcal{O}(|\Sigma|)$ in the worst case as well, but in the average case we have $b \in \mathcal{O}(1)$ for both of the implemented collision resolution techniques. Therefore, we can conclude that in general it is advisable to prefer the SHTI implementation technique if the expected alphabet size is large.

We will now focus on several typical user case scenarios. For each of them, we present the most appropriate algorithm and implementation technique based on their analysis in the previous chapters. Our main concern is the lowest time complexity. However, if the space complexity is also important, it is possible to adjust the selection of the appropriate algorithm and its implementation technique according to the Table 5.1.

Construction speed

If the speed of the suffix tree construction is the most important, then we can base our recommendations on the theoretical worst-case time complexity (see Table 5.2). Therefore, we would recommend either the McCreight's or Ukkonen's algorithm with top-down suffix link simulation.

However, as we have stated earlier, the worst-case time complexity alone is not sufficient for determining the fastest algorithm in practice. The most important factor which can change the expected time complexity on real input files is the *structure* of these files. We have already presented one heuristics which determines the implementation technique to use based on the expected alphabet size. It says that if the alphabet is small, it is better to use SLLI and otherwise, if the alphabet is large, it is better to use SHTI. This recommendation is also applicable here.

There is another heuristics according to which we can select the appropriate type of suffix link simulation. It is based on the result by Senft and Dvořák [SD12], who claim that the *bottom-up* suffix link simulation technique is not suitable for a certain family of input texts, which they call *adversary*. In particular, the authors proved that the time complexity of the Ukkonen's algorithm with bottom-up suffix link simulation on such texts is $\Omega(n^{1.5} \cdot b)$ where b is a *branching factor*. Now we present a definition of this string family, which was originally described by Senft and Dvořák [SD12].

Definition 5.1 *Suppose we have two integers k and m such that $k \geq m \geq 1$ and two characters a and b, $a \neq b$. A string $A_{k,m} = (a, b^k, a, b^1, a, b^2, a, b^3, \ldots, a, b^m)$ is called the **adversary** string with parameters k and m.*

In the definition above we have used the notation b^i which is simply a short form of describing a sequence of i characters where each of them is b.

Applying the results presented by Senft and Dvořák [SD12], we can conclude that if most of the input text is expected to be formed by adversary strings $A_{k,m}$, we should *not* use the *bottom-up* suffix link simulation.

We should also note that there might be situations in which it is advisable to use the PWOTD algorithm. In the original paper "Practical suffix tree construction" by Tata, Hankins, and Patel [THP04], the authors state that it is more beneficial to use the PWOTD algorithm when the alphabet size is large. They reason that the algorithm's ability to perform the partitioning part and divide all the suffixes into the partitions can decrease the number of necessary branching operations in the upper parts of the suffix tree. This is technically true. But on the other hand, it is important to note that partitioning can *not* eliminate the branching operations in the deeper parts of the suffix tree. Therefore, we would be a little more cautious when recommending this algorithm for large alphabets.

On the other hand, the PWOTD algorithm has one indisputable advantage. According to Tata, Hankins, and Patel [THP04], it provides much better *locality* of used memory references. This means that it *should* induce fewer processor's cache misses which should result in faster memory access times. However, it is disputable whether or not this advantage can outperform the other disadvantages of this algorithm on a particular input file. In Chapter 6 we show that according to the results of our experiments, the time of the suffix tree construction using the PWOTD algorithm varies mainly according to the average length of the longest common prefix of any two lexicographically adjacent suffixes. Also, it usually performs faster on files with larger alphabets and slower on files with smaller alphabets.

Sliding window

When constructing and maintaining a suffix tree over a sliding window, we can only use the modified version of the Ukkonen's algorithm. Also, the only available implementation techniques are SLLI with parent pointers and SHTI with parent pointers. Except for the standard two types of suffix link simulation, we can also choose between the two methods for edge label maintenance (see subsection 3.3.1), namely the original method by Fiala and Greene [FG89] or the batch update by Senft [Sen05b].

The batch update is simple to implement, but it requires an additional traversal of the entire suffix tree once in a while. When a traditional edge label maintenance method by Fiala and Greene is used, part of the suffix tree traversal necessary for performing the edge labels' updates is done by the suffix tree construction algorithm itself. Therefore, if well implemented, the overall amortized time complexity of this approach should be lower than the amortized time complexity of the batch update. In both cases it will be a constant. But the value of this constant is also important. Therefore, we recommend to use the edge label maintenance method by Fiala and Greene.

In order to determine which algorithm variation is the most suitable, we have to

know the text structure. As we have previously mentioned, there is at least one family of texts, for which the *bottom-up* suffix link simulation is significantly slower than the *top-down* suffix link simulation. For these texts, we recommend using the *top-down* suffix link simulation. For the other texts, *bottom-up* suffix link simulation can be used.

In section 4.3 we have shown that SHTI implementation technique with parent pointers is not suitable for implementation of the suffix tree construction over a sliding window, because it can *not* provide a quick access to the next remaining child of a vertex. Therefore, for the suffix tree construction over a sliding window we definitely recommend to use the SLLI implementation technique with parent pointers.

Read-only usage

When a suffix tree is not going to be further modified after its construction, it is possible to use all the suffix tree construction algorithms presented in this thesis. According to the asymptotic worst-case time complexity of a suffix tree construction, it is the most advisable to use the McCreight's or the Ukkonen's algorithm with top-down suffix link simulation.

However, the time complexity of a suffix tree construction may not necessarily correspond to the time complexity of the additional read-only operations. There are many kinds of read-only operations which can be performed on a suffix tree, but they use only a handful of actions available to be performed on a read-only suffix tree. One of the most time-consuming of these actions is the selection of the appropriate child of a parent, which is also called *branching*. The ability to perform fast branching operations is therefore essential when using the suffix tree read-only.

In general, the fastest branching operations are provided by the SHTI implementation technique because it uses a hash table. But for small alphabets, it might be equally fast to use the SLLI implementation technique as well.

Our implementation of the SHTI implementation technique provides two kinds of hash table collision resolution techniques — the double hashing and the cuckoo hashing. When the double hashing is used, the speed of its lookup operations is dependent on the hash table *load factor*. Therefore, they might be too slow when the relative hash table utilization is high. On the other hand, when the cuckoo hashing is used, the speed of its lookup operations is *always* constant, regardless of the hash table load factor. This property makes the cuckoo hashing a collision resolution technique of choice in this case.

The comparison of the top-down and bottom-up suffix link simulation techniques is not relevant in this case, because a suffix link simulation is only performed during the suffix tree construction.

As we have stated above, it is possible to use the PWOTD algorithm for read-only suffix tree usage. However, one of its most common operations is the determination of the length of the longest common prefix (often abbreviated simply as *lcp*) of several suffixes. In order to determine the depth of a certain vertex in a suffix tree, it is essential to determine the length of the longest common prefix of all its children's edge labels. The time complexity of this operation is linear with respect to the length of the longest common prefix. Therefore, if the average length of the longest common prefix of any two text suffixes is high, we might also expect that the time complexity of the lcp determination in the PWOTD algorithm will be high as well.

In addition, the speed of branching operation is also linear with respect to the number of children. This all means that in general, regarding the construction of an entire suffix

tree in the main memory and its subsequent read-only usage, it is not appropriate to use the PWOTD algorithm, because its expected time complexity on read-only operations is relatively high when compared to e.g. McCreight's algorithm and SHTI implementation technique.

Frequent modifications required

When it is necessary to perform subsequent modifications to an already constructed suffix tree, it is necessary to use an implementation technique which supports it. For that reason, we can not use the SLAI implementation technique along with the PWOTD algorithm.

There are many ways a suffix tree can be modified. In general, we can say that any suffix tree modification can be achieved by inserting and removing vertices and edges. Insertion of vertices and edges is very well supported by every implementation technique we have presented in this thesis, because it is required during the suffix tree construction. Therefore, if the desired suffix tree modifications only require that vertices and edges are *inserted*, then no changes to the original implementation techniques are necessary. In this case, we can recommend the same implementation techniques as in the case when our main concern was the speed of the suffix tree construction.

However, if the suffix tree modifications require that the vertices or edges are deleted from a suffix tree, we have to adjust our recommendations according to the speed of these operations. If the SLLI implementation technique is used, the speed of edge deletion is linear with respect to the number of children of a parent vertex at which this edge starts. It is therefore equally fast as the speed of edge insertion. Using this implementation, the vertices can be removed in a constant time.

Similarly, a vertex can be deleted in a constant time also when using the implementation technique SHTI. On the other hand, the time complexity of the edge deletion is dependent on the used collision resolution technique. If the double hashing is used, the time required for deletion of an edge is the same as the time required for its lookup. It is dependent on the hash table load factor and on the number of already deleted hash table entries. Therefore, the edge deletions can become slower over time.

Fortunately, this is not the case when using the cuckoo hashing collision resolution technique. The reason is that the worst-case time complexity of its delete operation is *constant* and does not change with the increasing hash table load factor or with the number of already deleted hash table entries. Therefore, this hashing technique possesses a great advantage in the time complexity of its delete operation compared to the double hashing, as well as to the implementation technique SLLI. Therefore, we can recommend it also for the construction of a suffix tree which is expected to be further modified.

In this user case, it is not necessary to choose between the two suffix link simulation types, because its selection will only affect the suffix tree construction itself. The desired modifications may or may not use the suffix link simulation. Therefore, a recommendation of the most appropriate type of suffix link simulation depends on the type of suffix tree modifications required. In general, we can say that unless the text is expected to be formed mostly by adversary strings (see Definition 5.1), it is safe to use both types of suffix link simulation.

Chapter 6

Benchmarks

As with any science, the theory must be supported by the practical experiments. In this chapter, we present the results of our benchmarks and conclude on whether they support the theoretical recommendations from Chapter 5 or not.

As we have mentioned previously, we have created our own implementations of all the suffix tree construction algorithms, their variations and suitable implementation techniques described in this thesis. We consider this implementation to be one of our main contributions to this thesis. Most of our benchmarks are devoted to the experimental evaluation of our implementations. But for comparison, we have also tested some alternative implementations, namely:

- implementation of the McCreight's algorithm using SLLI implementation technique by Kurtz [Kur99]
- implementations of different variations of the Ukkonen's algorithm using a custom linked lists implementation technique by Senft and Dvořák [SD12]
- implementation of the PWOTD algorithm using SLAI implementation technique by Tata, Hankins, and Patel [THP04] which was formerly available at [TP04]

Kurtz's implementation does not provide any options. It is implemented so that it uses McCreight's algorithm with *top-down* suffix link simulation and SLLI implementation technique. It does not support wide characters and it can only be used for the suffix tree construction entirely in memory. But nevertheless, we have decided to use it mainly for the comparison with our implementation of the SLLI implementation technique.

On the contrary, implementation by Senft and Dvořák provides many different variations of the suffix tree construction. In particular, it supports both *top-down* (also called *rescan*) and *bottom-up* (also called *climb*) suffix link simulation. The authors also introduced a hybrid type of the suffix link simulation, which they call *climbscan*. Its idea is to use the *bottom-up* suffix link simulation unless a certain number of ascending operations is encountered. In this case, the current suffix link simulation is aborted and *top-down* suffix link simulation is used instead. The details are presented in [SD12].

Moreover, the implementation by Senft and Dvořák also supports an enhanced linked list management called *move to front* or simply *mtf*. Its main idea is to adjust the child order of a vertex every time one of its children is accessed so that this child is positioned at the *beginning* of the linked list. This implementation also supports the construction of a suffix tree over a sliding window. Unfortunately, just like the implementation by Kurtz, it does not support wide characters.

The implementation by Tata, Hankins, and Patel supports the suffix tree construction using the PWOTD algorithm and the SLAI implementation technique only. It does not

provide any settings and just like the previous alternative implementations it does not support wide characters either.

Our implementation supports both the construction of a suffix tree in memory or over a sliding window. While constructing a suffix tree entirely in memory, it is possible to use the McCreight's algorithm, the Ukkonen's algorithm or the PWOTD algorithm. When McCreight's or Ukkonen's algorithms are used, it is possible to select one of the two possible implementation techniques — SLLI or SHTI. With each of these implementation techniques, it is possible to select the desired type of the suffix link simulation.

When constructing a suffix tree over a sliding window, only the Ukkonen's algorithm is available. It is possible to use either of the SLLI or SHTI implementation techniques. Also, it is possible to select the desired edge label maintenance method. We have implemented the original method by Fiala and Greene [FG89] as well as the batch update by Senft [Sen05b].

In addition, our implementation can be customized to use any of the two hash table collision resolution techniques described in section 4.2. And finally, it supports wide characters, which makes it unique among all the implementations we have presented. Moreover, it is possible to read input files not only on byte-by-byte basis but more importantly on character-by-character basis in various encodings, thanks to the usage of function `iconv`. This means that our implementation in fact supports the suffix tree construction on top of arbitrary Unicode text.

Benchmark setup

All the benchmarks have been conducted on a computer with Intel® Core™ i7 860 CPU running at 2.8 GHz with 16 GiB of dual channel DDR-3 memory running at 1600 MHz. The operating system used was Ubuntu 12.04, architecture x86-64 with Linux kernel 3.2.0-27 and the newest updates as of July 25, 2012.

We have tested our own implementation of all the algorithms and implementation techniques presented in this thesis. Moreover, we have also tested all the alternative implementations described earlier.

Our benchmarks consisted of suffix tree construction on top of the text coming from various kinds of input files. We have measured the total running time of the particular implementation and its *maximum resident set size*, which is the maximum amount of memory consumed *at a time*.

Since not all the implementation techniques can be directly compared, we present only a limited amount of our benchmark results in this chapter. Complete benchmark results, as well as all the scripts used for testing, along with the source code of most of the implementations can be found on the attached DVD, whose contents are described in Appendix A.

6.1 Input files

Now we briefly describe input files used in our benchmarks.

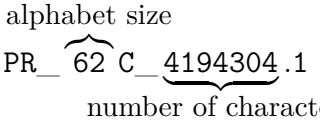
At first, we have generated input files containing pseudorandom characters of several different alphabets of various sizes. In particular we have used the alphabets of size 2, 4, 10, 26, 62, 100, 256, 1024, 4096 and 16 384 characters. These alphabets have been used for the generation of pseudorandom files of sizes 256 Ki, 1 Mi, 4 Mi, 16 Mi and 64 Mi

characters. Note that the file size is not in bytes, because thanks to the usage of large alphabets, some characters might be encoded as more than one byte. The encoding of all the pseudorandom input files is UTF-8.

In order to reduce the risk of generating a pseudorandom file of poor randomness, we have used an alternative pseudorandom number generator of higher quality called the Mersenne twister. Additionally, we have generated three files of each kind to minimize the risk that the benchmark results would be affected by an "unluckily" generated file. The results of our benchmarks on pseudorandom files are presented as an average over all three files of each kind.

In order to generate these pseudorandom files, we have used our own program for pseudorandom string generation called *rsgen* [Baš12b]. It uses Mersenne twister as implemented by the *randomc* library written by Fog [Fog10].

The names of generated pseudorandom files have the following form:



The prefix `PR` of these files' names stands for *pseudorandom* and each file's name is given a unique numbered suffix (in this case "`.1`") for its identification among the pseudorandom files of the same type and size.

All the generated pseudorandom files which have been used in this thesis are present on the attached DVD. In order to characterize the input files, we have used several commonly used text characteristics, namely:

- the size of the alphabet, denoted by $|\Sigma|$
- inverse probability of matching, or simply IPM
- the average and maximum length of the longest common prefix, or simply LCP, of any two lexicographically adjacent suffixes

The inverse probability of matching was defined by Ferragina and Navarro [FN05] as the inverse of the probability that any two arbitrarily selected text characters match. Its purpose is to estimate the *effective* size of the alphabet.

The average and maximum lengths of the longest common prefix of any two lexicographically adjacent suffixes provide an estimation of the expected length of the text's repeated substrings.

The size of the alphabet is easily determined by the selected method of the pseudorandom file's generation. In order to determine inverse probability of matching, we have used our own program called *IPM Utility* [Baš12a]. It is Unicode-aware, which means that it can determine the IPM of all the generated pseudorandom files.

For computation of the average or maximum length of the longest common prefix of any two lexicographically adjacent text suffixes, we have used an implementation based on the *sdsl* library [Gog12]. But unfortunately, this implementation is not Unicode-aware, which means that we were not able to compute these values for any file which uses characters of size more than one byte.

The properties of all the pseudorandom files can be found on the attached DVD. In Table 6.1 we present the properties of a small sample of pseudorandom files.

Except for the pseudorandom input files, we have also used the standard input files from the *Lightweight Corpus* [Man03] as well as from the *Pizza&Chili Corpus* [FN05]. Their properties can be found in the tables 6.2 and 6.3, as well as on the attached DVD.

Table 6.1: Properties of the selected pseudorandom files

File				LCP size	
name	size [MiB]	$\|\Sigma\|$	IPM	average	maximum
PR_2C_262144.1	0.25	2	2.00	16.88	31
PR_2C_1048576.1	1.00	2	2.00	18.88	45
PR_2C_4194304.1	4.00	2	2.00	20.89	44
PR_10C_262144.1	0.25	10	9.53	4.81	11
PR_10C_1048576.1	1.00	10	9.53	5.41	11
PR_10C_4194304.1	4.00	10	9.53	6.02	13
PR_62C_262144.1	0.25	62	61.49	2.39	6
PR_62C_1048576.1	1.00	62	61.49	2.81	6
PR_62C_4194304.1	4.00	62	61.50	3.08	7

Table 6.2: Properties of files in Pizza & Chili Corpus

File				LCP size	
name	size [MiB]	$\|\Sigma\|$	IPM	average	maximum
dblp.xml	282.42	97	28.73	44.91	1084
dna	385.22	16	3.91	2420.73	1 378 596
english.400M	400.00	226	15.25	5771.85	987 770
pitches	53.25	133	39.75	262.00	25 178
proteins.400M	400.00	26	17.19	654.36	263 313
sources	201.10	230	24.77	371.80	307 871

Unfortunately, we were not able to directly use all the files from the *Pizza&Chili Corpus*, because some of them are too large. Since we have used a computer with 16 GiB of memory, we could only use input files of size up to approximately 400 MiB. With respect to this limitation, we have shortened the files "proteins" and "english" and used their 400 MiB prefixes only.

Table 6.3: Properties of files in Lightweight Corpus

File				LCP size	
name	size [MiB]	$\|\Sigma\|$	IPM	average	maximum
chr22.dna	32.95	5	4.24	1979.25	199 999
etext99	100.40	146	15.74	1108.63	286 352
gcc-3.0.tar	82.62	150	21.76	8603.21	856 970
howto	37.60	197	14.29	267.56	70 720
jdk13c	66.50	113	35.24	678.94	37 334
linux-2.4.5.tar	110.87	256	27.12	479.00	136 035
rctail96	109.40	93	23.27	282.07	26 597
rfc	111.03	120	10.20	93.02	3445
sprot34.dat	104.54	66	15.41	89.08	7373
w3c2	99.37	256	38.05	42 299.75	990 053

Additionally, we have also used special input files. In particular, we have generated files containing a single repeated character, repeated small letters of English alphabet and the adversary strings $A_{i^2,i}$ (see Definition 5.1) of various lengths.

Table 6.4 lists the properties of all the used special files. The prefix of a file's name

Table 6.4: Properties of special files

File				LCP size	
name	size [MiB]	$\|\Sigma\|$	IPM	average	maximum
SA_20	0.00	2	1.07	133.25	399
SA_100	0.01	2	1.01	3333.14	9999
SA_500	0.36	2	1.00	83 333.12	249 999
SA_2500	8.94	2	1.00	2 083 333.11	6 249 999
SA_10000	143.07	2	1.00	33 333 333.11	99 999 999
SA_16721	399.98	2	1.00	93 197 280.11	279 591 840
SRA_1048580	1.00	26	26.00	524 265.00	1 048 554
SRC_1048576	1.00	1	1.00	524 288.00	1 048 575

before the underscore "_" character indicates the type of this file's content. Its suffix indicates the file size, but it is not necessarily expressed in the number of characters.

If the prefix equals to "SA", it means that this file contains the adversary string $A_{i^2,i}$ as defined in Definition 5.1. The value of i is stored in the second part of this file's name, after the underscore character.

If the file's name has a prefix of "SRA", it contains the repeated small characters of the English alphabet. Similarly, if the file's name has a prefix of "SRC", it consists of a single repeated character "c" only. The number of characters in these files is given in the second part of their names.

We have not used all of these special input files in all of our benchmarks. The reason is that several algorithms and implementation techniques are extremely slow on some of these files. Therefore, the number of results from the benchmarks on special input files is very limited.

6.2 Results

In this section we present some of the most important results of our benchmarks.

At first, we compare our implementation of the PWOTD algorithm with the original implementation by Tata and Patel [TP04]. The running times and memory consumption of these implementations are presented in Table 6.5.

The last two columns of this table contain relative time and memory consumption of our implementation with respect to the implementation by Tata, Hankins, and Patel [THP04]. As we can see, both implementation are comparably fast. Usually, our implementation is a little faster, on average about 14%. But there are also some input files, on which the original implementation is faster.

However, the memory consumption of the original implementation is considerably higher than the memory consumption of our implementation. We suppose that it is caused by more aggressive memory allocation strategy of the original implementation.

Table 6.5: Comparison of the original implementation of the PWOTD algorithm by Tata, Hankins, and Patel and our implementation on pseudorandom files.

File	original		ours		difference [%]	
name	time [s]	space [MiB]	time [s]	space [MiB]	time	space
PR_2C_262144	0.2	17.09	0.15	7.55	75.00	44.18
PR_2C_1048576	0.9	66.5	0.69	34.17	76.67	51.38
PR_2C_4194304	3.97	264.12	3.05	76.73	76.83	29.05
PR_2C_16777216	18.61	1054.62	15.75	304.69	84.63	28.89
PR_2C_67108864	86.98	4216.62	87.84	1152.58	100.99	27.33
PR_4C_262144	0.12	17.09	0.1	6.99	83.33	40.90
PR_4C_1048576	0.54	66.5	0.42	26.24	77.78	39.46
PR_4C_4194304	2.39	264.12	2.16	62.28	90.38	23.58
PR_4C_16777216	11	1054.63	10.83	247.23	98.45	23.44
PR_4C_67108864	50.8	4216.62	57.13	930.28	112.46	22.06
PR_10C_262144	0.08	17.09	0.06	4.78	75.00	27.97
PR_10C_1048576	0.35	66.5	0.26	16.88	74.29	25.38
PR_10C_4194304	1.53	264.12	1.16	51.65	75.82	19.56
PR_10C_16777216	6.91	1054.62	5.95	205.48	86.11	19.48
PR_10C_67108864	32.13	4216.58	29.07	799.4	90.48	18.96
PR_26C_262144	0.05	17.09	0.04	4.27	80.00	24.99
PR_26C_1048576	0.28	66.5	0.2	16.36	71.43	24.60
PR_26C_4194304	1.12	264.13	0.82	45.61	73.21	17.27
PR_26C_16777216	5.32	1054.62	4.72	195.61	88.72	18.55
PR_26C_67108864	24.38	4216.62	22.81	717.66	93.56	17.02
PR_62C_262144	0.05	17.09	0.04	4.53	80.00	26.51
PR_62C_1048576	0.22	66.5	0.17	15.59	77.27	23.44
PR_62C_4194304	0.88	264.12	0.82	42.66	93.18	16.15
PR_62C_16777216	4.72	1054.62	4.95	183.88	104.87	17.44
PR_62C_67108864	19.1	4216.62	20.96	700.91	109.74	16.62
average					86.01	25.77

Our next benchmark compares the running times of many variations of the Ukkonen's suffix tree construction algorithm implemented by Senft and Dvořák [SD12]. We have tried all the available suffix link simulation techniques as well as all the available linked list management types. The results are presented in the Table 6.6.

Table 6.6: Implementation by Senft and Dvořák — running times in seconds on pseudo-random files

	rescan		climb		climbscan		
File name	simple	mtf	simple	mtf	simple	mtf	best
PR_2C_262144	0.03	0.02	0.02	0.03	0.02	0.02	0.02
PR_2C_1048576	0.19	0.2	0.17	0.18	0.18	0.18	0.17
PR_2C_4194304	0.94	0.98	0.82	0.81	0.85	0.81	0.81
PR_2C_16777216	4.61	4.83	4.11	4.08	4.07	3.99	3.99
PR_2C_67108864	22.04	22.26	18.46	18.33	19.08	18.48	18.33
PR_4C_262144	0.03	0.04	0.03	0.03	0.03	0.04	0.03
PR_4C_1048576	0.25	0.27	0.24	0.25	0.24	0.25	0.24
PR_4C_4194304	1.52	1.58	1.29	1.33	1.28	1.34	1.28
PR_4C_16777216	7.39	7.57	6.15	6.37	6.19	6.39	6.15
PR_4C_67108864	36.35	37.04	30.24	31.18	29.84	30.6	29.84
PR_10C_262144	0.05	0.05	0.06	0.05	0.06	0.05	0.05
PR_10C_1048576	0.43	0.43	0.41	0.43	0.4	0.43	0.4
PR_10C_4194304	2.72	2.69	2.36	2.42	2.36	2.43	2.36
PR_10C_16777216	14.05	13.88	11.52	11.82	11.52	11.81	11.52
PR_10C_67108864	70.42	69.26	58.45	59.77	58.28	60.47	58.28
PR_26C_262144	0.11	0.11	0.12	0.11	0.11	0.12	0.11
PR_26C_1048576	0.91	0.9	0.89	0.93	0.9	0.93	0.89
PR_26C_4194304	5.92	5.87	5.21	5.44	5.17	5.38	5.17
PR_26C_16777216	31.66	31.31	26.74	27.27	26.69	27.41	26.69
PR_26C_67108864	141.18	137.44	121.39	125.77	122.85	124.54	121.39
PR_62C_262144	0.17	0.18	0.19	0.2	0.2	0.2	0.17
PR_62C_1048576	1.6	1.65	1.67	1.74	1.62	1.73	1.6
PR_62C_4194304	9.95	10.19	9.49	9.88	9.46	9.87	9.46
PR_62C_16777216	62.36	62.78	53.95	53.67	53.17	54.68	53.17
PR_62C_67108864	283.63	278.34	249.76	253.09	248.66	262.39	248.66

The same tests can be conducted using our implementation. Since we have implemented slightly different types of implementation techniques with slightly higher number of their variations, we present these results in multiple tables. At first, Table 6.7 contains the running times of the implementation technique SLLI.

Table 6.7: Our implementation of SLLI — running times in seconds on pseudorandom files

	McCreight's		Ukkonen's		
File name	top-down	bottom-up	top-down	bottom-up	best
PR_2C_262144	0.04	0.04	0.06	0.06	0.04
PR_2C_1048576	0.24	0.22	0.29	0.26	0.22
PR_2C_4194304	1.21	1.03	1.41	1.22	1.03
PR_2C_16777216	5.66	4.85	6.44	5.59	4.85
PR_2C_67108864	26.26	22.6	29.58	25.47	22.60
PR_4C_262144	0.04	0.04	0.06	0.06	0.04
PR_4C_1048576	0.31	0.27	0.35	0.32	0.27
PR_4C_4194304	1.72	1.45	1.93	1.62	1.45
PR_4C_16777216	8.47	6.95	9.3	7.64	6.95
PR_4C_67108864	40.31	33.31	43.75	36.19	33.31
PR_10C_262144	0.06	0.05	0.07	0.06	0.05
PR_10C_1048576	0.43	0.4	0.47	0.44	0.40
PR_10C_4194304	2.71	2.32	2.89	2.45	2.32
PR_10C_16777216	14.21	11.67	14.83	12.2	11.67
PR_10C_67108864	68.46	56.88	70.98	59.27	56.88
PR_26C_262144	0.09	0.08	0.09	0.09	0.08
PR_26C_1048576	0.74	0.71	0.79	0.77	0.71
PR_26C_4194304	4.79	4.36	4.91	4.47	4.36
PR_26C_16777216	26.98	22.55	27.73	23.17	22.55
PR_26C_67108864	129.3	111.73	131.57	113.28	111.73
PR_62C_262144	0.14	0.14	0.16	0.16	0.14
PR_62C_1048576	1.32	1.36	1.39	1.43	1.32
PR_62C_4194304	8.15	7.87	8.3	8.04	7.87
PR_62C_16777216	50.2	43.26	51.06	44.2	43.26
PR_62C_67108864	259.54	231.91	266.15	235.59	231.91

As we can see, the running times on pseudorandom files of our SLLI implementation are comparable to the running times of the implementation by Senft and Dvořák [SD12]. But since we have also implemented another implementation techniques, we would like to test their performance as well. Therefore, Table 6.8 contains benchmark results of our implementation of SHTI implementation technique with the hash table collision resolution technique of *double hashing*.

Table 6.8: Our implementation of SHTI with double hashing — running times in seconds on pseudorandom files

	McCreight's		Ukkonen's		
File name	top-down	bottom-up	top-down	bottom-up	best
PR_2C_262144	0.12	0.11	0.13	0.12	0.11
PR_2C_1048576	0.9	0.85	0.96	0.92	0.85
PR_2C_4194304	5.49	4.97	5.8	5.31	4.97
PR_2C_16777216	23.1	21.16	24.39	22.4	21.16
PR_2C_67108864	114.55	108.12	120.18	112.57	108.12
PR_4C_262144	0.08	0.08	0.1	0.09	0.08
PR_4C_1048576	0.49	0.48	0.55	0.52	0.48
PR_4C_4194304	4.93	4.36	5.37	4.77	4.36
PR_4C_16777216	22.83	19.22	24.78	21.02	19.22
PR_4C_67108864	88.24	73.62	92.84	78.25	73.62
PR_10C_262144	0.06	0.06	0.07	0.07	0.06
PR_10C_1048576	0.38	0.38	0.42	0.45	0.38
PR_10C_4194304	5.79	5.5	6.21	5.94	5.5
PR_10C_16777216	39.43	36.22	42.35	39.32	36.22
PR_10C_67108864	163.21	132.2	171.65	140.31	132.2
PR_26C_262144	0.05	0.05	0.06	0.06	0.05
PR_26C_1048576	0.32	0.32	0.36	0.36	0.32
PR_26C_4194304	6.01	5.98	6.27	6.23	5.98
PR_26C_16777216	53.01	52.19	56.96	56.25	52.19
PR_26C_67108864	506.7	471.7	544.21	509.41	471.7
PR_62C_262144	0.07	0.07	0.08	0.08	0.07
PR_62C_1048576	0.37	0.38	0.39	0.4	0.37
PR_62C_4194304	6.88	6.93	7.13	6.88	6.88
PR_62C_16777216	51.68	51.06	54.75	53.96	51.06
PR_62C_67108864	1277.96	1240.5	1398.91	1369.06	1240.5

Here we can see a considerable increase in running time on many inputs. This is caused by the usage of the double hashing combined with tight memory allocation strategy. We know that double hashing can always reach the hash table load factor of 1 and can never fail to resolve a hash collision, provided that the hash table is not full. In our implementation, we are strongly relying on that fact and use hash table of the exact size necessary to store *every* edge that can possibly be present in a suffix tree. Therefore, if a suffix tree contains many edges, this approach significantly increases the time complexity.

On the other hand, this rather sparing memory allocation strategy allows this implementation technique to achieve its theoretical worst-case space complexity. This is not possible with the cuckoo hashing, because it usually can not utilize 100% of the hash table. Based on our experimental evaluation, our implementation of the cuckoo hashing can achieve a load factor of about 85% in average case before an insolvable hashing collision is encountered. That is why we have to keep the hash table larger than the maximum possible number of edges a suffix tree can have. The benchmark results of the SHTI

implementation technique with the cuckoo hashing are presented in the Table 6.9.

Table 6.9: Our implementation of SHTI with cuckoo hashing — running times in seconds on pseudorandom files

	McCreight's		Ukkonen's		
File name	top-down	bottom-up	top-down	bottom-up	best
PR_2C_262144	0.53	0.48	0.52	0.52	0.48
PR_2C_1048576	3.11	2.75	3.03	2.96	2.75
PR_2C_4194304	13.61	15.05	14.8	13.88	13.61
PR_2C_16777216	60.53	64.08	63.87	54.78	54.78
PR_2C_67108864	295.17	288.28	297.62	289.31	288.28
PR_4C_262144	0.3	0.28	0.32	0.31	0.28
PR_4C_1048576	1.38	1.23	1.36	1.33	1.23
PR_4C_4194304	6.03	5.7	6.23	5.8	5.7
PR_4C_16777216	25.08	23.26	25.69	24.79	23.26
PR_4C_67108864	118.47	111.5	125.46	124.48	111.5
PR_10C_262144	0.2	0.18	0.22	0.2	0.18
PR_10C_1048576	0.94	1.25	1.03	1.04	0.94
PR_10C_4194304	5.01	4.59	4.91	4.8	4.59
PR_10C_16777216	21.3	28.86	21.61	20.18	20.18
PR_10C_67108864	91.24	89.58	97.23	89.32	89.32
PR_26C_262144	0.18	0.17	0.18	0.18	0.17
PR_26C_1048576	0.91	0.91	0.99	0.95	0.91
PR_26C_4194304	4.31	4.39	4.63	4.55	4.31
PR_26C_16777216	19.7	19.3	21.04	20.42	19.3
PR_26C_67108864	88.23	103.9	87.2	98.31	87.2
PR_62C_262144	0.24	0.24	0.26	0.25	0.24
PR_62C_1048576	1.08	1.05	1.14	1.12	1.05
PR_62C_4194304	4.97	4.62	5.54	4.86	4.62
PR_62C_16777216	21.44	20.54	22.47	34.25	20.54
PR_62C_67108864	99.7	96.15	117.5	97.98	96.15

In order to be able to better see the differences, we have compiled all the best results achieved by any of our implementation techniques as well as the best results achieved by the implementation of Senft and Dvořák [SD12] into the Table 6.10. For reference, we have also tested an implementation by Kurtz [Kur99] on the same pseudorandom files. Its results are added to this table as well. Note that the results of our implementation of the PWOTD algorithm are also contained in this table, for comparison.

Table 6.10: Comparison of the best running times of all our implementation techniques, as well as the SLLI implementation technique by Kurtz relative to to the best variations of the implementation by Senft and Dvořák (the first column) on pseudorandom files

	S&D	our implementation								Kurtz	
		SLLI		SHTI				PWOTD			
				double hashing		cuckoo hashing					
File name	time[s]	time [s]	diff [%]	time [s]	diff [%]	time [s]	diff [%]	time [s]	diff [%]	time [s]	diff [%]
PR_2C_262144	0.02	0.04	200.00	0.11	550.00	0.48	2400.00	0.15	750.00	0.05	250.00
PR_2C_1048576	0.17	0.22	129.41	0.85	500.00	2.75	1617.65	0.69	405.88	0.31	182.35
PR_2C_4194304	0.81	1.03	127.16	4.97	613.58	13.61	1680.25	3.05	376.54	1.52	187.65
PR_2C_16777216	3.99	4.85	121.55	21.16	530.33	54.78	1372.93	15.75	394.74	6.96	174.44
PR_2C_67108864	18.33	22.60	123.30	108.12	589.85	288.28	1572.72	87.84	479.21	31.61	172.45
PR_4C_262144	0.03	0.04	133.33	0.08	266.67	0.28	933.33	0.10	333.33	0.06	200.00
PR_4C_1048576	0.24	0.27	112.50	0.48	200.00	1.23	512.50	0.42	175.00	0.38	158.33
PR_4C_4194304	1.28	1.45	113.28	4.36	340.63	5.70	445.31	2.16	168.75	2.10	164.06
PR_4C_16777216	6.15	6.95	113.01	19.22	312.52	23.26	378.21	10.83	176.10	10.04	163.25
PR_4C_67108864	29.84	33.31	111.63	73.62	246.72	111.50	373.66	57.13	191.45	46.90	157.17
PR_10C_262144	0.05	0.05	100.00	0.06	120.00	0.18	360.00	0.06	120.00	0.08	160.00
PR_10C_1048576	0.40	0.40	100.00	0.38	95.00	0.94	235.00	0.26	65.00	0.44	110.00
PR_10C_4194304	2.36	2.32	98.31	5.50	233.05	4.59	194.49	1.16	49.15	2.80	118.64
PR_10C_16777216	11.52	11.67	101.30	36.22	314.41	20.18	175.17	5.95	51.65	15.01	130.30
PR_10C_67108864	58.28	56.88	97.60	132.20	226.84	89.32	153.26	29.07	49.88	73.21	125.62
PR_26C_262144	0.11	0.08	72.73	0.05	45.45	0.17	154.55	0.04	36.36	0.12	109.09
PR_26C_1048576	0.89	0.71	79.78	0.32	35.96	0.91	102.25	0.20	22.47	0.85	95.51
PR_26C_4194304	5.17	4.36	84.33	5.98	115.67	4.31	83.37	0.82	15.86	5.24	101.35
PR_26C_16777216	26.69	22.55	84.49	52.19	195.54	19.30	72.31	4.72	17.68	28.89	108.24
PR_26C_67108864	121.39	111.73	92.04	471.70	388.58	87.20	71.83	22.81	18.79	137.38	113.17
PR_62C_262144	0.17	0.14	82.35	0.07	41.18	0.24	141.18	0.04	23.53	0.21	123.53
PR_62C_1048576	1.60	1.32	82.50	0.37	23.13	1.05	65.63	0.17	10.63	1.50	93.75
PR_62C_4194304	9.46	7.87	83.19	6.88	72.73	4.62	48.84	0.82	8.67	8.63	91.23
PR_62C_16777216	53.17	43.26	81.36	51.06	96.03	20.54	38.63	4.95	9.31	52.63	98.98
PR_62C_67108864	248.66	231.91	93.26	1240.50	498.87	96.15	38.67	20.96	8.43	273.53	110.00
average			104.74		266.11		528.87		158.34		139.97

The table 6.10 shows that the linked list implementation by Senft and Dvořák [SD12], as well as our SLLI implementation and the implementation by Kurtz [Kur99] are comparatively fast on pseudorandom files. On smaller files, the implementation by Senft and Dvořák [SD12] is slightly faster. On larger files and more importantly on larger alphabet sizes, the running times of all the implementations get pretty close.

Our SLLI implementation is showing small advantage over the other two linked list implementations on files with larger alphabets. The most probable cause is that our linked list implementation uses *sorted* linked lists, which help to decrease the running time on average.

When considering the hash table implementation, we can see the already mentioned high time complexity of the double hashing on some input files. This is caused by the tight memory allocation strategy. But on the other hand, there are input files on which the implementation technique SHTI with double hashing is comparatively very fast. In general, we can observe that the SHTI implementation technique gets to be comparatively faster on files with larger alphabet.

When comparing the running times of the PWOTD algorithm, we can conclude that it is undoubtedly very fast on pseudorandom files with larger alphabets. However, its running time on files with smaller alphabets is usually worse than the running time of the standard linked-list-based implementation techniques. This result partially supports the claims of its authors, Tata, Hankins, and Patel, who claim that it can be faster than the usual suffix tree construction algorithms. On pseudorandom files with large alphabets, this is indeed true.

Our next benchmark uses the input files form the *Pizza&Chili Corpus*. Unfortunately, we were not able to run the original implementation of the PWOTD algorithm on the files from this corpus. The program simply exited with "Invalid Character!" error. Moreover, we have experienced the same error also on every file from *Lightweight Corpus*. We have not been able to resolve this problem and therefore we do not present a comparison of original and our implementations of the PWOTD algorithm on these files.

Instead, we present the comparison of the running times of all the variations of the Ukkonen's suffix tree construction algorithm implemented by Senft and Dvořák [SD12]. The results of this comparison are organized in Table 6.11.

Table 6.11: Implementation by Senft and Dvořák — running times in seconds on files from *Pizza&Chili Corpus*

	rescan		climb		climbscan		
File name	simple	mtf	simple	mtf	simple	mtf	best
dblp.xml	91.87	78.82	73.22	70.34	73.14	70.55	70.34
dna	228.61	231.28	185.59	189.26	183.96	190.08	183.96
english.400M	364.33	305.89	262.48	250.44	263.52	251.60	250.44
pitches	58.82	41.52	42.82	35.17	43.16	34.93	34.93
proteins.400M	481.69	419.97	344.99	333.38	348.16	336.29	333.38
sources	109.47	81.83	80.46	71.08	81.23	71.46	71.08

Similarly, we can run the same tests using our implementation techniques. Tables 6.12, 6.13 and 6.14 present the running times of all the implementation techniques we have implemented on the files from *Pizza&Chili Corpus*.

Table 6.12: Our implementation of SLLI — running times in seconds on files from *Pizza&Chili Corpus*

	McCreight's		Ukkonen's		
File name	top-down	bottom-up	top-down	bottom-up	best
dblp.xml	84.84	69.32	101.80	85.15	69.32
dna	1147.22	1105.80	2078.30	2041.26	1105.80
english.400M	2456.35	2417.35	4651.50	4494.80	2417.35
pitches	56.66	45.32	70.60	59.52	45.32
proteins.400M	608.50	536.11	856.21	758.62	536.11
sources	161.46	135.20	232.39	225.96	135.20

Table 6.13: Our implementation of SHTI with double hashing — running times in seconds on files from *Pizza&Chili Corpus*

	McCreight's		Ukkonen's		
File name	top-down	bottom-up	top-down	bottom-up	best
dblp.xml	93.08	83.94	112.70	104.60	83.94
dna	934.76	840.68	1560.39	1436.31	840.68
english.400M	1720.78	1934.17	3131.45	3648.01	1720.78
pitches	32.51	30.22	43.36	41.05	30.22
proteins.400M	386.27	316.87	550.41	464.85	316.87
sources	139.69	129.47	203.70	192.54	129.47

Table 6.14: Our implementation of SHTI with cuckoo hashing — running times in seconds on files from *Pizza&Chili Corpus*

	McCreight's		Ukkonen's		
File name	top-down	bottom-up	top-down	bottom-up	best
dblp.xml	761.05	384.73	428.13	418.49	384.73
dna	1398.81	1255.19	2181.10	2118.04	1255.19
english.400M	2300.27	2091.26	3800.90	4229.71	2091.26
pitches	83.93	85.28	96.27	90.72	83.93
proteins.400M	904.66	758.18	1141.14	958.72	758.18
sources	391.39	573.80	477.60	448.32	391.39

When comparing the running times on *Pizza&Chili Corpus* of our implementation and the implementation by Senft and Dvořák, we can clearly see that our implementation is considerably slower on many files. We have not expected such results. Nonetheless, we have found a possible reason for this behavior.

It turns out that our implementation of SLLI and SHTI implementation techniques does not take full advantage of the active point's position. The problem is that when the active point is implicit, its position inside an edge is *not* remembered between the

algorithm's iterations. This means that we have to repeatedly scan the part of an edge between the insertion point and the active point.

On pseudorandom files, this deficiency did not show up, because these files tend to have short average length of LCP. This means that the expected number of characters between the insertion point and the active point is small. But the average length of LCP for the real-life input files present in corpuses tends to be higher. And that is why this issue has much more visible consequences on these input files. Therefore, we can conclude that this problem is mostly apparent on input files with large average LCP.

We continue by presenting a comparison of the running times of the best of our implementation techniques with the best of implementation techniques by Senft and Dvořák [SD12] and the implementation technique by Kurtz [Kur99] on the input files from *Pizza&Chili Corpus* and from *Lightweight Corpus*. This comparison is compiled in Table 6.15. As we have already mentioned, we could not include the benchmark results of the original implementation of the PWOTD algorithm, because we were not able to run it on these input files.

We also omit the detailed results of the benchmarks on the input files from *Lightweight Corpus*, but they can be found on the attached DVD. Instead, we just point out that there are some files (e.g. "dblp.xml" or "pitches") in this corpus, on which the performance of our implementation of SLLI implementation technique and SHTI implementation technique with double hashing is still relatively fast when compared to the implementation by Senft and Dvořák [SD12].

Note that implementation by Kurtz failed to complete on input file "dna" from *Pizza&Chili Corpus*, because it has received a `SIGSEGV` signal. Also, our implementation of PWOTD algorithm was too slow to complete in 10 hours on input files "chr22.dna" and "gcc-3.0.tar" from *Lightweight Corpus*, so we have stopped its execution.

The reason why PWOTD algorithm is so slow on these files is supported by the fact that they have relatively large average LCP. The average length of LCP is very important for the speed of this algorithm, because it often determines the length of the longest common prefix of currently processed suffixes. If the average length of this longest common prefix is high, it results in longer average time necessary for its determination and this results in overall longer running time of this algorithm.

Table 6.15: Comparison of the best running times of all our implementation techniques, as well as the SLLI implementation technique by Kurtz relative to to the best variations of the implementation by Senft and Dvořák (the first column) on files from *Pizza&Chili Corpus* and *Lightweight Corpus*

	S&D	our implementation								Kurtz	
		SLLI		SHTI				PWOTD			
				double hashing		cuckoo hashing					
File name	time[s]	time [s]	diff [%]	time [s]	diff [%]	time [s]	diff [%]	time [s]	diff [%]	time [s]	diff [%]
dblp.xml	70.34	69.32	98.55	83.94	119.33	384.73	546.96	268.04	381.06	102.73	146.05
dna	183.96	1105.80	601.11	840.68	456.99	1255.19	682.32	1120.81	609.27	—	—
english.400M	250.44	2417.35	965.24	1720.78	687.10	2091.26	835.03	2884.76	1151.88	384.93	153.70
pitches	34.93	45.32	129.75	30.22	86.52	83.93	240.28	77.06	220.61	54.39	155.71
proteins.400M	333.38	536.11	160.81	316.87	95.05	758.18	227.42	810.76	243.19	488.83	146.63
sources	71.08	135.20	190.21	129.47	182.15	391.39	550.63	2265.11	3186.71	118.76	167.08
average			357.61		271.19		513.77		965.45		153.83
File name	time[s]	time [s]	diff [%]	time [s]	diff [%]	time [s]	diff [%]	time [s]	diff [%]	time [s]	diff [%]
chr22.dna	11.42	24.20	211.91	25.88	226.62	57.34	502.10	>36 000.00	—	18.87	165.24
etext99	55.11	163.73	297.10	125.74	228.16	229.29	416.06	183.87	333.64	83.94	152.31
gcc-3.0.tar	21.46	541.47	2523.16	372.65	1736.49	452.06	2106.52	>36 000.00	—	38.27	178.33
howto	16.24	24.06	148.15	20.34	125.25	52.85	325.43	30.02	184.85	26.70	164.41
jdk13c	6.56	23.79	362.65	28.81	439.18	101.33	1544.66	106.93	1630.03	13.48	205.49
linux-2.4.5.tar	36.03	90.08	250.01	73.18	203.11	184.99	513.43	1741.26	4832.81	61.69	171.22
rctail96	27.75	36.97	133.23	40.26	145.08	152.41	549.23	116.70	420.54	45.98	165.69
rfc	38.42	43.07	112.10	44.85	116.74	160.09	416.68	91.02	236.91	62.96	163.87
sprot34.dat	44.20	45.24	102.35	41.47	93.82	144.67	327.31	88.64	200.54	63.52	143.71
w3c2	14.86	3426.36	23 057.60	2338.47	15 736.68	2413.48	16 241.45	3846.66	25 886.00	28.24	190.04
average			2719.83		1905.11		2294.29		4215.67		170.03

Table 6.15 also shows that our implementation of the SLLI and SHTI implementation techniques is definitely much slower than the implementations by Kurtz [Kur99] or by Senft and Dvořák [SD12]. Its running times keep up with the alternative implementations only on files which have low average LCP (see tables 6.2 and 6.3 for reference).

The most notable examples of corpus files which have low average LCP are "dblp.xml", "pitches", "sprot34.dat" and "rfc". On these files, the running times of our implementation of SLLI implementation technique and SHTI implementation technique with double hashing are comparable to the running times of alternative implementations. However, our implementation is usually a little slower even on these files.

On the other hand, our implementation is extremely slow on files whose average LCP is high. These files include "dna", "english.400M", "gcc-3.0.tar" and "w3c2".

The results of this benchmark confirm that our implementations of SLLI and SHTI implementation techniques indeed suffer form the problem with active point implementation which causes them to perform poorly on files with large average LCP. Additionally, this benchmark reveals another interesting result. It turns out that PWOTD algorithm is not as fast on real-life input files as it was on pseudorandom files.

The reason has already been explained — it is the longer average LCP of real-life input files, which causes the PWOTD algorithm to spend too much time on determining the length of the longest common prefix of its currently processed suffixes. Therefore, its overall execution time increases as well.

The benchmark of our implementation techniques on special input files is not expressive, because they suffer from the problem with active point implementation we have mentioned earlier. Moreover, since the average LCP of some special input files is very large, the running time of some of our implementations would be very high. Similarly, since the speed of the PWOTD algorithm is also highly dependent on the average length of the LCP, its running time would also be very high. Therefore, the tests of these algorithms on special files would not reveal any interesting results and that is why we do not present them in this thesis. However, they are available on the attached DVD.

Our next benchmark compares the running times of the suffix tree construction over a sliding window. We have implemented the Ukkonen's algorithm with two different edge label maintenance methods — *batch update* by Senft [Sen05b] and *percolating update* by Fiala and Greene [FG89]. Also, we have implemented both top-down, as well as bottom-up types of suffix link simulation.

Despite the fact that our implementation contains both SLLI and SHTI implementation techniques, we have only tested the running times of the SLLI implementation technique. The reason is that SHTI implementation technique is not suitable for suffix tree construction over a sliding window, because it does not support effective lookups for the remaining children of a certain vertex, as mentioned in the section 4.3.

In addition, we have also tested an alternative implementation of the suffix tree construction over a sliding window — namely the one by Senft and Dvořák [SD12]. Their implementation uses the *batch update* edge label maintenance method by Senft [Sen05b]. Every possible variation supported by this implementation has been tested.

At first, we present the results on pseudorandom files. Table 6.16 shows the running times of all the variations of the suffix tree construction over a sliding window implemented by Senft and Dvořák [SD12]. The size of a sliding window has been set to 8 MiB.

Table 6.16: Implementation by Senft and Dvořák over a sliding window of size 8 MiB — running times in seconds on pseudorandom files

	rescan		climb		climbscan		
File name	simple	mtf	simple	mtf	simple	mtf	best
PR_2C_262144	0.07	0.07	0.05	0.06	0.06	0.05	0.05
PR_2C_1048576	0.39	0.41	0.34	0.33	0.34	0.34	0.33
PR_2C_4194304	1.76	1.76	1.55	1.52	1.56	1.50	1.50
PR_2C_16777216	8.74	8.68	7.32	7.19	7.35	7.12	7.12
PR_2C_67108864	39.80	40.25	35.09	34.91	34.36	33.20	33.20
PR_4C_262144	0.07	0.07	0.06	0.06	0.07	0.06	0.06
PR_4C_1048576	0.44	0.45	0.38	0.38	0.39	0.38	0.38
PR_4C_4194304	2.28	2.29	1.91	1.94	1.95	1.95	1.91
PR_4C_16777216	10.77	10.89	8.93	8.96	9.18	9.12	8.93
PR_4C_67108864	51.71	53.02	43.56	43.93	44.70	44.63	43.56
PR_10C_262144	0.08	0.08	0.07	0.07	0.07	0.07	0.07
PR_10C_1048576	0.61	0.60	0.53	0.52	0.54	0.53	0.52
PR_10C_4194304	3.46	3.40	2.87	2.95	2.96	2.99	2.87
PR_10C_16777216	17.09	17.03	13.80	14.04	14.08	14.19	13.80
PR_10C_67108864	86.54	85.88	70.09	71.20	71.84	72.31	70.09
PR_26C_262144	0.16	0.16	0.14	0.14	0.14	0.15	0.14
PR_26C_1048576	1.15	1.15	1.04	1.04	1.05	1.05	1.04
PR_26C_4194304	6.75	6.73	5.89	5.97	5.92	6.02	5.89
PR_26C_16777216	35.88	35.93	29.70	29.79	29.87	30.20	29.70
PR_26C_67108864	160.88	160.73	135.90	136.51	134.22	137.68	134.22
PR_62C_262144	0.24	0.23	0.22	0.22	0.22	0.23	0.22
PR_62C_1048576	1.94	2.00	1.85	1.84	1.88	1.85	1.84
PR_62C_4194304	11.14	11.14	10.07	10.11	9.97	10.07	9.97
PR_62C_16777216	64.81	65.75	55.73	56.27	56.15	56.35	55.73
PR_62C_67108864	317.70	319.68	273.27	271.02	271.53	274.67	271.02

The same tests have also been run using our implementation of the suffix tree construction over a sliding window. Its results are summarized in Table 6.17. The same table also contains the best times achieved by the implementation of Senft and Dvořák [SD12] (in the next to last column). The last column in this table represents the relative running time of the best of our variations with respect to the running time of the best of variations by Senft and Dvořák [SD12].

Table 6.17: Our implementation with SLLI implementation technique over a sliding window of size 8 MiB — running times in seconds on pseudorandom files

	batch update		percolating update		best		
File name	top-down	bottom-up	top-down	bottom-up	ours	S&D	diff [%]
PR_2C_262144	0.11	0.09	0.12	0.10	0.09	0.05	180.00
PR_2C_1048576	0.55	0.47	0.61	0.54	0.47	0.33	142.42
PR_2C_4194304	2.45	2.11	2.80	2.42	2.11	1.50	140.67
PR_2C_16777216	17.59	15.93	13.40	11.86	11.86	7.12	166.57
PR_2C_67108864	95.18	87.52	61.01	53.83	53.83	33.20	162.14
PR_4C_262144	0.11	0.09	0.11	0.11	0.09	0.06	150.00
PR_4C_1048576	0.56	0.50	0.62	0.56	0.50	0.38	131.58
PR_4C_4194304	2.96	2.58	3.26	2.90	2.58	1.91	135.08
PR_4C_16777216	17.66	15.54	15.80	13.97	13.97	8.93	156.44
PR_4C_67108864	85.19	76.83	69.18	60.79	60.79	43.56	139.55
PR_10C_262144	0.14	0.12	0.14	0.13	0.12	0.07	171.43
PR_10C_1048576	0.71	0.62	0.76	0.74	0.62	0.52	119.23
PR_10C_4194304	4.69	4.10	5.01	4.42	4.10	2.87	142.86
PR_10C_16777216	24.00	20.90	22.99	20.60	20.60	13.80	149.28
PR_10C_67108864	113.65	98.48	104.09	88.96	88.96	70.09	126.92
PR_26C_262144	0.16	0.14	0.16	0.15	0.14	0.14	100.00
PR_26C_1048576	1.29	1.15	1.30	1.20	1.15	1.04	110.58
PR_26C_4194304	6.30	5.68	6.47	5.83	5.68	5.89	96.43
PR_26C_16777216	37.16	32.61	36.75	32.46	32.46	29.70	109.29
PR_26C_67108864	168.31	148.79	164.00	143.99	143.99	134.22	107.28
PR_62C_262144	0.27	0.25	0.28	0.26	0.25	0.22	113.64
PR_62C_1048576	1.67	1.56	1.72	1.59	1.56	1.84	84.78
PR_62C_4194304	9.15	8.36	9.27	8.49	8.36	9.97	83.85
PR_62C_16777216	62.50	56.26	61.85	55.45	55.45	55.73	99.50
PR_62C_67108864	288.61	259.90	293.09	257.95	257.95	271.02	95.18
average							128.59

From these results, we can see that our implementation of the suffix tree construction over a sliding window is competitively fast with respect to the implementation by Senft and Dvořák [SD12] on pseudorandom files. Its speed is undoubtedly lower on input files with small alphabets, but it is a little faster on input files with larger alphabets.

Also, we can observe that the percolating update by Fiala and Greene [FG89] is usually somewhat faster than the batch update by Senft [Sen05b]. This result conforms to our estimation presented in Chapter 5.

The last benchmark presented in this chapter compares the running times of the suffix tree construction algorithms over the sliding window on the input files from *Pizza&Chili Corpus* and *Lightweight Corpus*. At first, we present the running times of all the variations of the implementation of the suffix tree construction over a sliding window by Senft and Dvořák [SD12]. The size of a sliding window has been set to 8 MiB. These results are arranged in tables 6.18 and 6.19.

Table 6.18: Implementation by Senft and Dvořák over a sliding window of size 8 MiB — running times in seconds on files from *Pizza&Chili Corpus*

	rescan		climb		climbscan		
File name	simple	mtf	simple	mtf	simple	mtf	best
dblp.xml	190.27	185.11	170.30	167.33	173.09	168.60	167.33
dna	348.00	355.75	300.30	305.26	301.20	298.96	298.96
english.400M	501.65	472.86	412.02	415.84	421.93	410.18	410.18
pitches	77.50	60.55	60.27	53.92	60.79	54.23	53.92
proteins.400M	610.85	588.76	511.22	512.27	529.88	510.74	510.74
sources	191.87	170.35	161.23	154.79	161.72	155.58	154.79

Table 6.19: Implementation by Senft and Dvořák over a sliding window of size 8 MiB — running times in seconds on files from *Lightweight Corpus*

	rescan		climb		climbscan		
File name	simple	mtf	simple	mtf	simple	mtf	best
chr22.dna	27.09	27.10	23.23	23.60	23.14	23.42	23.14
etext99	123.09	114.01	100.15	99.99	103.27	103.22	99.99
gcc-3.0.tar	65.29	59.27	55.48	54.17	56.01	54.43	54.17
howto	44.33	37.69	34.38	32.79	35.30	33.09	32.79
jdk13c	25.48	24.52	23.18	23.14	23.59	23.50	23.14
linux-2.4.5.tar	100.62	88.50	82.42	79.00	84.02	80.14	79.00
rctail96	85.73	81.24	72.83	72.48	73.28	72.62	72.48
rfc	105.66	97.66	89.81	85.89	88.04	88.24	85.89
sprot34.dat	104.38	90.76	80.82	81.54	86.97	84.88	80.82
w3c2	58.12	50.54	46.98	46.30	47.75	46.72	46.30

Similarly to the previous benchmarks, we are interested in comparison of our implementation technique with the implementation technique by Senft and Dvořák [SD12]. Therefore, in the tables 6.20 and 6.21 we present the results of all the available variations of our implementation of the SLLI implementation technique over a sliding window. Again, we have used a sliding window of size 8 MiB.

For comparison, the best results of the suffix tree construction over a sliding window by Senft and Dvořák [SD12] are also present in this table, namely in its next to last column. The last column contains the relative running time of the best of our implementations with respect to the best implementation by Senft and Dvořák [SD12].

Note that no test of our implementation on the input file "w3c2" from the *Lightweight Corpus* has been completed, because these tests failed to finish in 2 hours. After that time, we have stopped them. Since this file has very high length of its average LCP, the reason why it took so long was probably caused by the above mentioned problem with the active point management.

From the tables 6.20 and 6.21 we can confirm that our implementation of a suffix tree construction over a sliding window is indeed slower than the implementation by Senft and Dvořák [SD12]. But we can also see that its relative speed is not as dramatically slow as it was in case of constructing the entire suffix tree in memory. We suppose that the reason is smaller overall size of the suffix tree, which makes the problem with active point in our implementation harder to exploit.

We can also see that percolating update proved to be faster even on real-life files from

Table 6.20: Our implementation with SLLI implementation technique over a sliding window of size 8 MiB — running times in seconds on files from *Pizza&Chili Corpus*

	batch update		percolating update		best		
File name	top-down	bottom-up	top-down	bottom-up	ours	S&D	diff [%]
dblp.xml	291.24	266.13	231.01	203.89	203.89	167.33	121.85
dna	3925.03	3897.50	3931.70	3611.17	3611.17	298.96	1207.91
english.400M	3403.59	3314.04	3294.24	3231.86	3231.86	410.18	787.91
pitches	109.19	95.40	99.04	85.56	85.56	53.92	158.68
proteins.400M	1351.35	1246.41	1194.47	1100.18	1100.18	510.74	215.41
sources	505.40	471.80	438.10	405.98	405.98	154.79	262.28
average							459.01

Table 6.21: Our implementation with SLLI implementation technique over a sliding window of size 8 MiB — running times in seconds on files from *Lightweight Corpus*

	batch update		percolating update		best		
File name	top-down	bottom-up	top-down	bottom-up	ours	S&D	diff [%]
chr22.dna	3518.96	3528.79	123.94	109.41	109.41	23.14	472.82
etext99	502.80	482.20	479.06	491.44	479.06	99.99	479.11
gcc-3.0.tar	3573.67	3507.34	2304.42	2240.72	2240.72	54.17	4136.46
howto	51.05	49.35	45.38	44.95	44.95	32.79	137.08
jdk13c	124.35	120.80	111.31	107.75	107.75	23.14	465.64
linux-2.4.5.tar	312.12	296.09	281.33	273.06	273.06	79.00	345.65
rctail96	167.34	161.77	147.78	134.99	134.99	72.48	186.24
rfc	168.20	152.38	142.51	123.91	123.91	85.89	144.27
sprot34.dat	145.88	129.75	123.26	104.94	104.94	80.82	129.84
w3c2	>7200.00	>7200.00	>7200.00	>7200.00	—	46.30	—
average							721.90

corpuses. This is expected, since it has also been estimated in Chapter 5.

Finally, we would like to note that the running times of the *bottom-up* suffix link simulation proved to be faster than the running times of the *top-down* suffix link simulation in almost all benchmarks we have conducted. Obviously, this difference depends on the properties of the input files, as well as on the implementation technique used. Thanks to the constant worst-case time complexity of the branching operation when using the SHTI implementation technique with *cuckoo hashing*, its results are the least affected by the selected type of suffix link simulation. On the other hand, the results of the SLLI implementation technique are affected the most.

For the SLLI implementation technique on real-life texts from the corpuses we have observed an average reduction of the construction time by about 10% when using the *bottom-up* suffix link simulation. Similar results have been experienced on the pseudorandom files.

On the contrary, when using the SHTI implementation technique with *cuckoo hashing*, there was almost no reduction in the construction time on the pseudorandom files. However, the construction time on real-life texts from corpuses has still been reduced by about 5%.

This all means that we can support the results by Senft and Dvořák [SD12] and recommend the usage of this type of suffix link simulation as well.

The remaining results of our experiments are not directly presented in this thesis. As we have already mentioned, the complete benchmark results along with all the scripts used to run them and extract their results can be found on the attached DVD.

Having presented the most important of our results, we can conclude that they support the estimated theory well.

Chapter 7

Conclusions

In this thesis, we have analyzed many of the common algorithms for the suffix tree construction. We have started from the very beginning and introduced the theory necessary for the explanation of these algorithms. Then we have described the following three suffix tree construction algorithms in detail — the McCreight's algorithm in section 3.1, the Ukkonen's algorithm in section 3.2 and the PWOTD algorithm in section 3.4. For the McCreight's and Ukkonen's algorithms, we have presented two alternative types of suffix link simulation — the traditional *top-down* approach and relatively new *bottom-up* approach introduced by Senft and Dvořák [SD12].

We have also explained an enhancement by Fiala and Greene [FG89] which enables the construction of a suffix tree over a sliding window. Two alternative methods of edge label maintenance have been presented — the *percolating update* by Fiala and Greene [FG89] and the *batch update* by Senft [Sen05b].

In addition to the theoretical explanation of these algorithms and their variations, we have also provided details on how they can be implemented in practice. We have explained SLLI and SHTI implementation techniques by Kurtz [Kur99] and the original implementation technique used to implement the PWOTD algorithm by Tata, Hankins, and Patel [THP04].

Each algorithm and implementation technique has been described using a single, unified terminology. Moreover, the implementation techniques have been enhanced so that they are easier to understand and simpler to implement.

Instead of simply presenting the existing implementation techniques in a more comprehensible language, some of them have also been extended. In particular, we have extended SLLI and SHTI implementation techniques so that they can also be used for the implementation of a suffix tree over a sliding window.

Besides presenting and explaining the algorithms and implementation techniques, we have also made simple recommendations on when it is advisable to use a particular algorithm and implementation technique and why. These recommendations can be used as a basic outline for the selection of the appropriate suffix tree construction algorithm and implementation technique in various situations.

In addition to the theoretical recommendations, we have also performed practical experiments which prove that our recommendations work in practice. We have made a lot of tests of many different algorithms, their variations and implementation techniques. During this testing, we have used various kinds of different input files — namely the pseudorandom files, corpus files and also some special files.

Except for testing our own implementation, we have also tested alternative imple-

mentations, namely various variations of the Ukkonen's algorithm by Senft and Dvořák [SD12], implementation of McCreight's algorithm by Kurtz [Kur99] and implementation of the PWOTD algorithm by Tata and Patel [TP04]. The benchmarks of these alternative implementations were helpful in order to further support our recommendations.

The main contribution of this work is the unified and expressive presentation of many different algorithms for the construction of a suffix tree. We have explained many algorithms and implementation techniques in a great detail.

Appendix A

Attached DVD

There is a `README` file in all the important directories on the DVD. Basic instructions can always be found there. The following is an overview of the content of the attached DVD.

- PDF version of this thesis (`stcmb.pdf`)
- source code of our implementation of the suffix tree construction algorithms and some other utilities (directory `src`)
- generated documentation for this source code (directory `doc`)
- the entire benchmark setup (directory `benchmarks`) containing:
  - scripts used to perform the benchmarks themselves
  - source code of alternative implementations (directory `benchmarks/alternative_implementations`)
  - pseudorandom input files (in compressed tar archive `benchmarks/pseudorandom_input_files.tar.gz`)
  - input files from Pizza&Chili corpus (in compressed tar archive `benchmarks/pizza_chili_corpus.tar.gz`)
  - input files from Manzini's Lightweight corpus (in compressed tar archive `benchmarks/manzini_lightweight_corpus.tar.gz`)
  - raw properties of all the input files (outputs from the appropriate programs) (directory `benchmarks/raw_properties_of_the_input_files`)
  - properties of all the input files (raw properties compiled into a compact form) (directory `benchmarks/properties_of_the_input_files`)
  - raw benchmark results (program outputs from each test run) (directory `benchmarks/raw_results`)
  - extracted benchmark results (raw results compiled into a compact form) (directory `benchmarks/results`)

Bibliography

[Baš12a] P. Bašista. *IPM Utility. Unicode-aware utility which computes the inverse of the probability that any two arbitrarily chosen characters in the input file match.* 2012. URL: `https://github.com/pbasista/IPM-utility` (visited on 07/14/2012).

[Baš12b] P. Bašista. *rsgen. Simple Unicode-aware utility to generate pseudorandom strings using Mersenne twister as PRNG.* 2012. URL: `https://github.com/pbasista/rsgen` (visited on 07/14/2012).

[BM77] R. S. Boyer and J. S. Moore. "A fast string searching algorithm." In: *Communications of the ACM* 20.10 (Oct. 1977), pp. 762–772. DOI: `10.1145/359842.359859`.

[Con12] The Unicode Consortium. *The Unicode Standard, Version 6.1.0. A computing industry standard for the consistent encoding, representation and handling of text expressed in most of the world's writing systems.* 2012. URL: `http://www.unicode.org/standard/standard.html` (visited on 07/14/2012).

[Far97] M. Farach. "Optimal suffix tree construction with large alphabets." In: *Proceedings of the 38th Annual Symposium on Foundations of Computer Science.* FOCS '97. Washington, DC, USA: IEEE Computer Society, 1997, pp. 137–143. ISBN: 0-8186-8197-7. DOI: `10.1109/SFCS.1997.646102`.

[FG89] E. R. Fiala and D. H. Greene. "Data compression with finite windows." In: *Communications of the ACM* 32.4 (Apr. 1989), pp. 490–505. DOI: `10.1145/63334.63341`.

[FN05] P. Ferragina and G. Navarro. *Pizza&Chili Corpus. A collection of texts for experimenting and validating compressed indexes.* 2005. URL: `http://pizzachili.dcc.uchile.cl/texts.html` (visited on 07/14/2012).

[Fog10] A. Fog. *randomc. A C++ class library containing various random number generators.* 2010. URL: `http://www.agner.org/random/` (visited on 07/14/2012).

[GKS99] R. Giegerich, S. Kurtz, and J. Stoye. "Efficient Implementation of Lazy Suffix Trees." In: *Algorithm Engineering.* Ed. by J. Vitter and C. Zaroliagis. Vol. 1668. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1999, pp. 30–42. ISBN: 978-3-540-66427-7. DOI: `10.1007/3-540-48318-7_5`.

[Gog12] S. Gog. *sdsl. An open source C++ library of efficient succinct data structures.* 2012. URL: `https://github.com/simongog/sdsl` (visited on 07/14/2012).

[Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Computer Science and Computational Biology. Cambridge University Press, Jan. 1997. ISBN: 978-0-521-58519-8.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. "Fast Pattern Matching in Strings." In: *SIAM Journal on Computing* 6.2 (1977), pp. 323–350. DOI: 10.1137/0206024.

[Kur99] S. Kurtz. "Reducing the space requirement of suffix trees." In: *Software: Practice and Experience* 29.13 (Nov. 1999), pp. 1149–1171. DOI: 10.1002/(SICI)1097-024X(199911)29:13<1149::AID-SPE274>3.0.CO;2-O.

[Lar99] N. J. Larsson. "Structures of String Matching and Data Compression." PhD thesis. Box 118, S-22100 Lund, Sweden: Department of Computer Science, Lund University, Sept. 17, 1999. ISBN: 91-628-3685-4. URL: http://www.larsson.dogma.net/thesis.pdf (visited on 07/14/2012).

[Man03] G. Manzini. *Lightweight Corpus. A collection of files used in experiments with deep-shallow suffix array and BWT construction algorithm.* 2003. URL: http://people.unipmn.it/manzini/lightweight/corpus/ (visited on 07/14/2012).

[McC76] E. M. McCreight. "A Space-Economical Suffix Tree Construction Algorithm." In: *J. ACM* 23.2 (Apr. 1976), pp. 262–272. DOI: 10.1145/321941.321946.

[PR01] R. Pagh and F. F. Rodler. "Cuckoo Hashing." In: *Algorithms — ESA 2001*. Ed. by F. auf der Heide. Vol. 2161. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2001, pp. 121–133. ISBN: 978-3-540-42493-2. DOI: 10.1007/3-540-44676-1_10.

[SD12] M. Senft and T. Dvořák. "On-line suffix tree construction with reduced branching." In: *Journal of Discrete Algorithms* 12 (Jan. 2012), pp. 48–60. DOI: 10.1016/j.jda.2012.01.001.

[Sen05a] M. Senft. "Suffix Tree Based Data Compression." In: *SOFSEM 2005: Theory and Practice of Computer Science*. Ed. by P. Vojtáš et al. Vol. 3381. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, pp. 350–359. ISBN: 978-3-540-24302-1. DOI: 10.1007/978-3-540-30577-4_38.

[Sen05b] M. Senft. "Suffix Tree for a Sliding Window: An Overview." In: *WDS 2005 - Proceedings of Contributed Papers*. Ed. by J. Šafránková. Prague, Czech Republic: ***matfyz****press*, 2005, pp. 41–46. ISBN: 80-86732-59-2. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.1447&rep=rep1&type=pdf (visited on 07/14/2012).

[THP04] S. Tata, R. A. Hankins, and J. M. Patel. "Practical suffix tree construction." In: *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*. VLDB Endowment, 2004, pp. 36–47. ISBN: 0-12-088469-0. URL: http://portal.acm.org/citation.cfm?id=1316695 (visited on 07/14/2012).

[TP04] S. Tata and J. M. Patel. *TDD. Top-Down disk-based approach for very fast suffix tree construction*. 2004. URL: http://www.eecs.umich.edu/tdd/index.html (visited on 04/12/2009).

[Ukk95] E. Ukkonen. "On-line construction of suffix trees." In: *Algorithmica* 14.3 (Sept. 1995), pp. 249–260. DOI: 10.1007/BF01206331.

[Wei73] P. Weiner. "Linear pattern matching algorithms." In: *SWAT '73: Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*. Washington, DC, USA: IEEE Computer Society, 1973, pp. 1–11. DOI: 10.1109/SWAT.1973.13.