FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Enhanced suffix arrays implementation and its usage
Student:	Bc. Minh Trieu Quang
Supervisor:	Ing. Jan Trávníček, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	Until the end of summer semester 2020/21

Instructions

Study the definitions and algorithms of the construction and usage of enhanced suffix array from [1] as a replacement for suffix trees.
Propose data structures in C++ representing the enhanced suffix array.
Implement a construction algorithm of the enhanced suffix array [1].
Implement algorithms of your choice from [1] that simulates at least three different suffix tree traversals with enhanced suffix array.
Implement the chosen algorithms using the standard suffix tree traversals.
Test your implementation appropriately and compare the effectiveness of the algorithms of your choice.

References

[1] Abouelhoda, M. I., Kurtz, S., & Ohlebusch, E. (2004). Replacing suffix trees with enhanced suffix arrays. *Journal of discrete algorithms*, *2*(1), 53-86.

doc. Ing. Jan Janoušek, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 14, 2020

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Enhanced suffix arrays implementation and its usage

Bc. Minh Trieu Quang

Department of Theoretical Computer Science
Supervisor: Ing. Jan Trávníček, Ph.D.

April 17, 2020

Acknowledgements

I would like to thank my supervisor Ing. Jan Trávníček, Ph.D. for his incredible guidance, patience, and support during the entire time on writing this thesis.

My thanks also go to my dear friends with whom I have been studying alongside for five years.

Last but not least, I would like to give a big thanks to my family: my parents and my brother for supporting me throughout writing this thesis, not to mention through my studies on CTU FIT.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on April 17, 2020 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Trieu, Quang Minh. *Enhanced suffix arrays implementation and its usage.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2020.

Abstrakt

Hlavní nevýhodou suffixového stromu je velká paměťová náročnost. Jedna z paměťově efektivnějších struktur je *suffixové pole*, a nedávno se ukázalo, že každý algoritmus řešený suffixovým stromem lze nahradit stejně časově efektivním algoritmem využívajícím suffixového pole, pokud jej rozšíříme o další informace a struktury.

Řešení navrhuje datovou strukturu vylepšeného suffixového pole (`ESA`) v C++ a implementaci vybraných algoritmů, které simulují tři odlišné průchody suffixového stromu. Toto řešení je důkladně otestováno, vyzkoušeno a proběhlo experimentální vyhodnocení algoritmů využívající suffixový strom a navrhovanou datovou strukturu.

Klíčová slova suffixové pole, rozšířené sufixové pole, suffixový strom, lcp tabulka, vyhledávání v textu, zpracovávání textu

Abstract

The suffix tree has a major drawback having a large space consumption. The more space efficient data structure than suffix tree is a *suffix array*, and recently it was shown that every algorithm using a suffix tree can be replaced with an algorithm based on a suffix array in the same time complexity if the suffix array is enhanced with additional information and structures.

The result is a proposed data structure of the enhanced suffix array (`ESA`) in C++ and implementations of the chosen algorithms that simulates three different suffix tree traversals. This solution is thoroughly tested, experimented and compared with the algorithms using the suffix tree and its standard traversals.

Keywords suffix array, enhanced suffix array, suffix tree, lcp table, pattern matching, text processing

Contents

List of Figures

List of Tables

Chapter 1

Introduction

String processing might seem like a basic task at first glance, but it is not that trivial and plays an important key in a lot of human fields. The most prominent field is bioinformatics, for example a genome analysis, but it can also be found in compression or musicology.

It is heavily studied for the last few decades and a lot of algorithms and data structures are being invented to support the run time or space requirements. The *suffix tree* is undoubtedly one of the most powerful data structure in string processing, for it can be used to efficiently solve many string processing problems.

Nonetheless, suffix trees suffer from a huge drawback, which makes them not as widespread as one should expect [1]. First, the space requirements of a suffix tree is quite large. Second, suffix trees have a poor locality of memory reference, causing a significant efficiency loss on caching architectures.

There are more space efficient data structures than suffix tree, and the most prominent one is the *suffix array* introduced by Manber and Myers [2]. It can be constructed in $\mathcal{O}(n)$ in the worst case by constructing the suffix tree [3] and placing the leaf numbers in order. However, this still require a huge amount of space to construct the suffix tree. Fortunately, the suffix arrays can be directly constructed in linear time [4, 5, 6].

Abouelhod et al. [1] showed that every algorithm using a suffix tree can be replaced with an algorithm using a suffix array by enhancing it with additional data structures. This is very beneficial since the data are only getting larger.

First, we go through the definition and construction of a suffix tree, break the chosen algorithm into the types of traversal, then we focus on the suffix arrays and enhancing it with additional information, and finally describe the equivalent algorithm which keeps the same time complexity, but improving the space requirements.

Last but not least, the experimental results and comparison between the two data structures and the algorithms are presented in the Chapter *7 Experimental results*.

Chapter 2

Preliminaries

This chapter presents the basic notions which are further to be used on building the foundation of the *enhanced suffix array*. More specific notions are defined in the following chapters. The basic definitions are based on [1, 7, 8].

2.1 Alphabet, String

Definition 2.1 (Alphabet)**.** *Alphabet* is a non-empty finite ordered set of characters denoted by the Σ.

Note 1. We will suppose that the size of the alphabet is a constant and that $n < 2^{32}$. Therefore the integer in the range $[0, n]$ can be stored in 4 bytes.

Definition 2.2 (String)**.** *String* is a finite sequence of characters over Σ. The set of *all strings over* Σ is denoted as Σ^*. The set of non-empty strings $\Sigma^* \setminus \{\epsilon\}$ is denoted Σ^+.

A *length* n of a string S is defined as the length of the sequence of characters of the S and is denoted by $|S|$. A character at index i of S, for $0 \leq i < n$, is denoted as S$[i]$.

Furthermore, we denote S$[i..j]$ as a *substring* starting with the character at the position i and ending with the character at position j. It is also sometimes denoted by a *pair* (i, j) *of positions*.

Definition 2.3 (Lexicographic order)**.** The *lexicographic ordering*, denoted by $\leq$, is an ordering on strings induced by an ordering on the characters. It is defined as follows. Let $x = x[0..n-1], y = y[0..m-1] \in \Sigma^*$. The $x < y$, iff one of the condition holds:

- $n < m$ and $x[0..n-1] = y[0..n-1]$,
- $x[0..i-1] = y[0..i-1]$ and $x[i] < y[i]$, for $0 < i \leq \min(n, m)$.

Definition 2.4 (Concatenation)**.** The *concatenation* of two strings x and y is the string composed of the characters of x followed by the characters of y. It is denoted as xy or $x \cdot y$.

Definition 2.5 (Sentinel character)**.** A *sentinel character* | is assumed to be an element of Σ and is considered to be the largest of all other elements, i.e. $c <$ | for every $c \in \Sigma \setminus \{|\}$, but does not occur in S.

Definition 2.6 (Prefix)**.** A *prefix* u of a string S if there exists a word v (possibly empty) such that $S = uv$

Definition 2.7 (Suffix)**.** A *suffix* u of a string S if there exists a word v (possibly empty) such that $S = vu$

2.2 Graph

Before diving deeper, let's present some basic notions of graph theory that builds up the suffix tree. The definitions are based on [9].

Definition 2.8 (Graph)**.** A *graph* G is a pair (V, E), where V is a non-empty finite set of nodes and E is a set of edges. An *edge* in a graph is denoted as a set $\{u, v\}$. As for a *directed* graph, an edge is a pair (u, v) such that the edge is leaving u and entering v.

Definition 2.9 (Path, cycle)**.** A *path* P is a sequence of nodes $(v_0, v_1, \ldots, v_n)$, $n \geq 1$ if $\{v_i, v_i + 1\}$ is an edge for every $v_i \in P$, where $1 \leq i \leq n$. A *cycle* is such a path $(v_0, v_1, \ldots, v_n)$ that starts and ends with the same node.

Definition 2.10 (Directed acyclic graph)**.** A *directed acyclic graph* (DAG) is a graph that doesn't contain any cycle.

Definition 2.11 (Degree)**.** Let $G = (V, E)$ be a graph and $v \in V$ its node. A *degree* $\deg(v)$ of a node v denotes the number of edges in graph G incident with v.

Definition 2.12 (Tree, rooted tree)**.** A *tree* is an acyclic connected graph. Any node of a tree can be selected as a *root*. Such a tree is called *rooted tree*.

Definition 2.13 (Labelled tree)**.** A *labelled tree* is a tree, where every node n is labelled by a symbol $c \in \Sigma$.

Definition 2.14 (Leaf)**.** A *leaf* in a tree $T = (V, E)$ is a node $n \in V$ that has $deg(n) = 1$. Otherwise, it is called an *internal node*.

Chapter 3

Suffix Tree

A suffix tree is a data structure that is built upon a string. It is considered one of the most important data structures in string processing, as it can be used to solve many string problems in linear time, not just exact matching which can be solved by Knuth-Morris-Pratt [10] or Boyer-Moore algorithms [11]. The most prominent field of usage is genome analysis in bioinformatics since the human genome is very large and do not change. The applications are described at the end of the chapter. In the following sections, we'll have a look at the definition and the construction of a suffix tree. The proofs, lemmas and theorems are based on [3].

3.1 Definition

A suffix tree is a compacted *suffix trie*. We can look at suffix trie as a simple deterministic automaton that recognizes the suffixes of a string S [12], where terminal states are corresponding to the suffixes of the string S.

However, a suffix trie can lead to a quadratic memory space according to the length of the string S. This drawback is avoided in suffix tree by compressing the nodes of degree 1 that are not terminal. Here's the formal definition:

Definition 3.1 (Suffix Tree)**.** A *suffix tree T of a string S* is a *directed rooted tree* with exactly n leaves numbered 0 to $n-1$. Each internal node, other than the root, has at least two children and each edge is labelled with a non-empty substring of S.

Yet there is no guarantee that suffix tree of any arbitrary string S exists. The problem occurs if any suffix is a prefix of another suffix. For example a string $S =$`aacaaca`, where the suffix $S[3..|S|-1] =$ `aca` is a prefix of a suffix $S[1..|S|-1] =$ `acaaca`.

To prevent this problem, the last character of the string should not occur anywhere else in the string. This is usually solved by appending the *sentinel* character to the string (see Definition 2.5).

In order to avoid any further confusion in the future, string S will be implicitly meant as $S|$.

Definition 3.2 (Path label of a node)**.** A *path label of a node* v is a concatenation of the labels of every edge leading from the root of T to the node v. Thus, every suffix $S_i = S[i..|S| - 1]$ corresponds to each of the leaf i of the suffix tree T.

Remark 3.1. No two edges out of a node can have edge-labels beginning with the same character.

The internal nodes can be viewed as a common prefix of two or more suffixes. An example of a suffix tree can be seen in Figure 3.1.

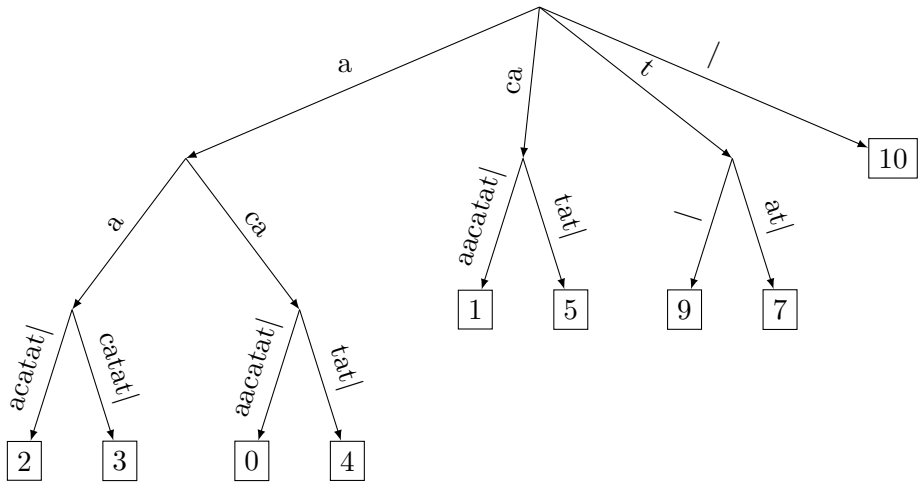

Figure 3.1: The suffix tree for S = `acaaacatat|`.

Proposition 3.1 (Number of nodes)**.** *The suffix tree T of a string S of length n has at least $n + 1$ nodes and at most $2n - 1$ nodes.*

Proof of minimum. Let S contain only unique characters. That means there is no need for internal nodes, i.e. no need for any branches. There are n suffixes and for every suffix there is a leaf. Therefore, the minimum is n leaves and one root node, which makes it $n + 1$. □

Proof of maximum. Every internal node propose a branch. When a new branch is created, it must eventually lead to an extra new leaf. Suppose we have n internal nodes, then there must be $n + 1$ leaves. From the suffix tree properties, there is only n leaves, so the internal nodes are bounded by

the $n-2$. Hence, the maximum is one root, n leaves and at most $n-2$ internal nodes, which makes it $1 + n + (n-2) = 2n - 1$ nodes. □

The difference can be seen on the figure 3.2, where the left suffix tree (a.) reaches the minimum number of nodes and the left (b.) reaches the maximum.

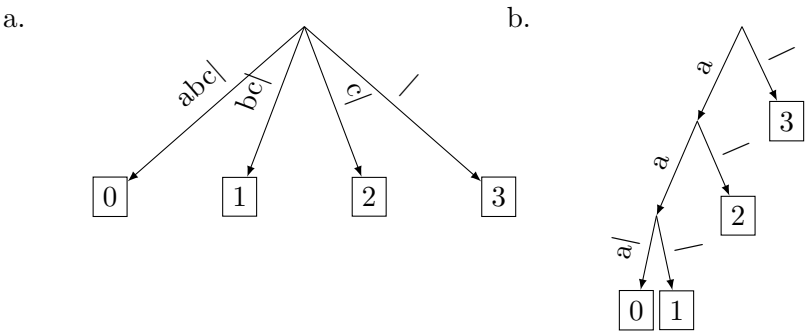

Figure 3.2: The suffix tree for string $S =$ `abc|` and string $S =$ `aaa|`

3.2 Construction

Naive algorithm

A naive algorithm is very straight-forward. Suppose we have a string S of length n. We start by inserting a suffix $S_0 = S[0...n-1]$ into the tree. For every suffix $S_i = S[i...n-1]$, where $0 < i < n$, we insert into a tree by matching the characters of the suffix with the existing path from root until no further matches are possible. This matching path is effectively unique (Remark 3.1) and ends at some point. It can be either a node w or in the middle of an edge between (u, v).

- If it's the node, create a new edge (w, i) ending to a leaf labelled with i representing the suffix S_i.

- If it ends in the middle of an edge, we break it into two edges (u, w) and (w, v) by inserting a new node w. The edge (u, w) would be labelled with the part of (u, v) that matched with the suffix S_i and (w, v) would be the rest. Afterwards, we create a new edge and a leaf as described in the previous case.

As we can see, building the tree with the naive method above will take in the worst case $\mathcal{O}(n^2)$.

Linear-time algorithm

The first linear-time algorithm was given by Weiner [13], or at least explicitly introduced. Following this work, a more space efficient algorithm was given by McCreight [14] a few years later. Afterwards, Ukkonen [15] developed an algorithm based on different intuitive ideas, that has all the advantages of McCreight's algorithm, although these two methods are relatively very close to each other. Recently, Kurtz [16] presented an improved implementation of linear time construction that reduces the space requirements .

Nonetheless, let's take a look at Ukkonen's algorithm in the next section. It was implemented for the comparison with the enhanced suffix array, for a simple reason. It's easy to understand and uses less memory in practice than McCreight's algorithm.

Despite the fact that the original algorithm is described in terms of automata, the following algorithm will be described as Gusfield [3] does.

3.3 Ukkonen's algorithm

Ukkonen at first presented an inefficient algorithm running in $\mathcal{O}(m^3)$, which is then being optimized with certain observations and tricks to obtain the linear solution. We are going to present the algorithm in the same manner and gradually introduce the tricks to understand how Ukkonen reached the claimed linear time.

To build a suffix tree $\mathcal{T}$ for string S of length m, we split the process into m phases. Each phase for each character of the S. Let i be the current phase, and let's denote the $\mathcal{T}_i$ as a suffix tree of $S[0..i]$. At the end of phase i, we'll have a tree $\mathcal{T}_i$. For every phase, we'll proceed i extensions, one for each character in the current prefix. At the end of every extension j, it is ensured that $S[j..i]$ is in the tree $\mathcal{T}_i$. The pseudocode can be seen below in the Algorithm 3.3.1.

3.3.1 Suffix extensions

Let $\beta = S[j...i-1]$. In the extension j, when the algorithm finds the end of the β, it extends this suffix according to one of the rules described in [3]:

Rule 1 In the current tree, the path β ends at a leaf. Update the leaf edge by appending a character $S[i]$.

Rule 2 No path from the end of string β starts with character $S[i]$. Split the edge and create a new internal node if necessary, then add a new leaf with an edge labelled with character $S[i]$.

Rule 3 The path $\beta \cdot S[i]$ exists in the current tree, no actions are taken.

Algorithm 3.3.1: High Level Ukkonen's algorithm

Input : string S of length n
Output: tree $\mathcal{T}_n$

```
1 construct tree T_0 which consists of a root and first character
2 for i = 1 to n − 1 do
3 |  begin phase i
4 |  for j = 0 to i do
5 |  |  begin phase j
6 |  |  in the current tree find the end of the path from the root
  |  |   labelled S[j..i − 1]. If necessary, extend the path by adding
  |  |   character S[i], thus ensuring that string S[j..i] is in the tree.
7 return T_n
```

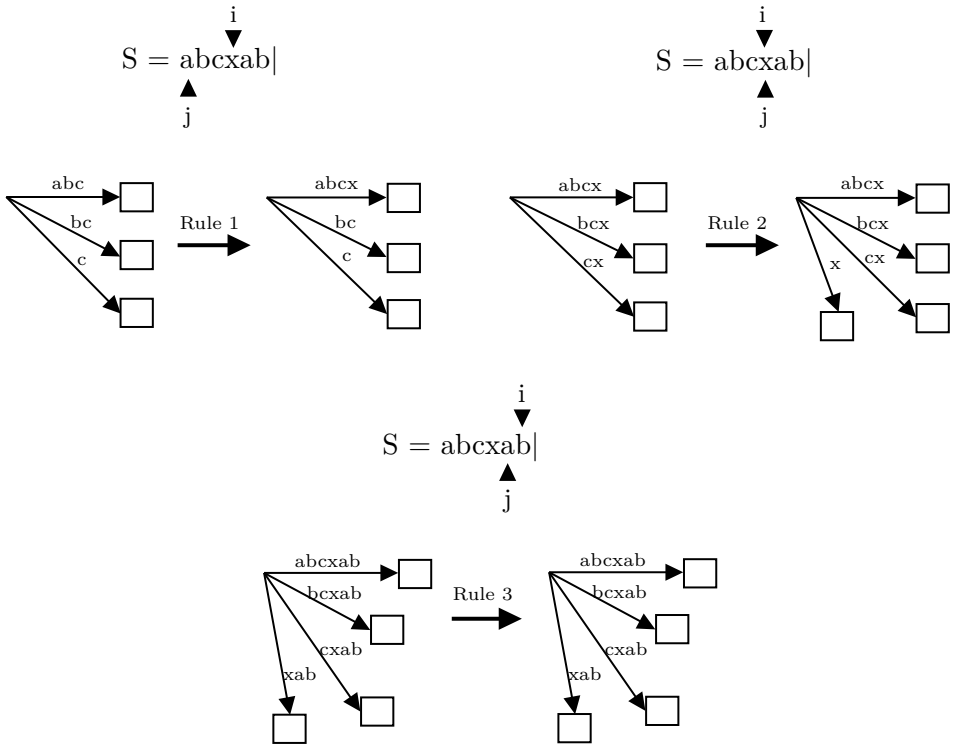

Figure 3.3: An example of extending by the suffix extensions rules

3.3.2 Suffix links

The most important element of speeding up the algorithm is to use the suffix links. This allow the algorithm to lower the time complexity of phase i to $\Theta(i)$, thus making the whole algorithm $\Theta(n^2)$.

Definition 3.3 (Suffix link)**.** Let $x\alpha$ be a string, where x is a single character and α the (possible empty) substring. Furthermore, let's denote a node u in the suffix tree by $\overline{\alpha}$ if its path label is α.

A *suffix link* is a pointer from $\overline{x\alpha}$ to $\overline{\alpha}$. The suffix link of a node v is often denoted as a function $s(v)$.

If α is an empty substring, then the suffix link $s(\overline{x})$ is the root of a tree. The root node itself is not considered an internal and therefore doesn't have any suffix link.

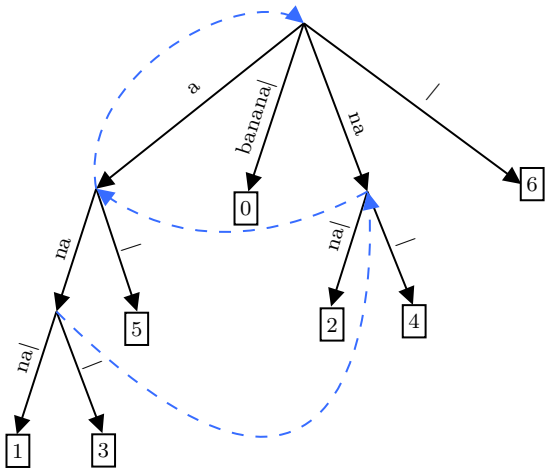

Figure 3.4: An example of suffix tree of $S =$ `banana|` with suffix links.

The definition alone doesn't imply that every internal node has a suffix link, or even, have only one. This is established in the following lemmas.

Lemma 3.1. *Assume the algorithm is processing an extension j of phase $i+1$. If a new internal node v with path label $S[j..i+1]$ is created, then either the path label $S[j+1..i+1]$ exists in the current tree or an internal node at the end of string $S[j..i+1]$ will be created in extension $j+1$ of the same phase $i+1$.*

Proof. New internal node is created only if the extension Rule 2 is applied. In the current extension j of phase $i+1$, it has to find $S[j..i]$ and insert a new internal node v after such path. This means, that there is some character c, which is different from $S[i+1]$ and continues the path. Such continuation must exist, otherwise the Rule 2 could not have been applied.

Consider now extension $j+1$ of phase $i+1$. The $S[j+1..i]$ was already added in extension $j+1$ of phase i, therefore the path exists and has a continuation with character c. If c is the only continuation in the path, then a new internal node w will be created at the end of $S[j+1..i]$. If there is another continuation, then the path $S[j+1..i]$ must have ended at an internal node. □

Lemma 3.2. *Assume the algorithm is processing an extension j of phase $i+1$, then any newly created internal node will have a suffix link from it by the end of the next extension.*

Proof. By the induction on number of phases.

Base case: $\mathcal{T}_0$ doesn't contain any internal nodes.

Inductive case: suppose that at the end of phase i, every internal node has a suffix link, and consider phase $i+1$. By Lemma 3.1, when a new node v is created in extension j, the node $s(v)$ will be found or created in extension $j+1$. The last extension of a phase will not create any new internal nodes, so all suffix links created in phase $i+1$ are known by the end of the phase, and therefore $\mathcal{T}_{i+1}$ has all its suffix links. □

With these two lemmas, we have proved the existence of suffix links for each internal node.

Theorem 3.1. Given a suffix tree $\mathcal{T}_i$, for any internal node v with a label $x\alpha$, there always exists one node w of $\mathcal{T}_i$ with path label α.

Next, let's show how to use suffix links to reduce the time complexity when performing the extensions. Let's assume the algorithm is starting at phase $i+1$. We use the same description from [17] which is called **extension algorithm**:

Extension 0**, phase** $i+1$**:** The first suffix to be inserted in the tree is $S[0..i+1]$. To insert the character $S[i+1]$, we have to locate the $S[0..i]$, which is so far the longest string in the tree, therefore it must end at a leaf. By keeping a pointer to the leaf containing the full string, this extension can be done in constant time. Recall that *Rule 1* always extends the suffix from $S[0..i]$ to $S[0..i+1]$.

Extension 1**, phase** $i+1$**:** In order to find $S[1..i]$, we walk up one node to internal node v. Let α be label forming the edge between the the node v and the leaf. From there, we follow the *suffix link* of v. Once in $s(v)$ we walk down the path labelled with α until we reach the end of the path $S[1..i]$. Here we perform the relevant suffix extension (it is guaranteed the path exists but doesn't have to be composed of one single edge) and insert $S[i+1]$.

Extension $j > 1$, phase $i + 1$: The procedure is the same as in the extension 1. The only difference is, that the end of $S[j..i]$ may be at an internal node having a suffix link. In such case, the algorithm will follow this suffix link.

3.3.3 Trick: Skip/count

With just reducing the work from the root to $s(v)$ is not enough to achieve the linear-time bound yet. Even with the proper implementation of suffix links, it would still be cubic. In each extension, the algorithm makes the comparisons equal to the length of the string which is very time consuming. By applying this trick, it will help us find the place to append the new character $S[i+1]$ in phase $i + 1$ very quickly. Therefore, it will reduce the complexity of walking on the tree to be equal to the number of nodes on the path, rather than the number of characters on the path.

For this trick to work properly, assume these two operations: (1) retrieval of the length of the edge-label, (2) extraction of any character from S at any given position. Both of the operations is assumed to take constant time.

Suppose we are at extension j of phase $i + 1$ at node $s(v)$. Let $S[j..i]$ be α and length be n. Recall, there must be a path α from $s(v)$ and from the properties of suffix tree, there cannot be more edges out of this node starting with the same character. Let p be the number of characters on this edge. We compare the length p and n. If $p < n$, simply skip to the node at the end of the edge and set $n = n-p$, and the next character on the string to be matched is $q = p + 1$. At the following node find similarly the next outgoing edge. In general, when the algorithm finds the next edge on the path, it compares the length. When $p < n$, it skips to the node at the end of the edge, sets $n = n-p$ and $q = q + p$, and search for the next edge starting with the character q and this continues in the same manner.

Eventually, the p will be larger or equal to n, then the algorithm skips to the character n on the edge and quits, assuring the α ends on that edge exactly n characters down its label.

Before showing how does this trick reduce the running time, let's define the *node-depth*.

Definition 3.4 (Node-depth)**.** Node-depth of a node v is a number of nodes on the path from the root to v. The **current node-depth** of the algorithm is the node-depth of the node most recently visited by the algorithm.

Lemma 3.3. *Let $(v, s(v))$ be a suffix link traversed in algorithm at extension j of phase $i + 1$. Then node-detph of v is greater than the node-depth of $s(v)$ at most by one.*

Proof. Consider the path from v and $s(v)$. Let the path label of v be $x\beta$, then the path label to $s(v)$ must be β. Because $x\beta$ is a prefix of path to v and β to $s(v)$, it follows that the suffix link from any internal ancestor of v goes to an ancestor of $s(v)$. If β is a non-empty string, then $s(v)$ is an internal node, otherwise it is root. By the definition of suffix link, no two ancestor internal nodes of v can receive the same suffix link. Thus, the only extra ancestor v (without those corresponding to ancestors of $s(v)$) can have is an internal node whose path label is x. Therefore, the node-depth of v is at most one more than $s(v)$. □

Theorem 3.2. Utilizing the skip/count technique, any phase of Ukkonen's algorithm takes $\mathcal{O}(n)$ time.

Proof. During the extension j of phase i, the algorithm walks up to a node v, traverses the suffix link to $s(v)$ and from there walks down some number of nodes applying the suffix extension rules. It was already established, that every operation takes constant time except for the down-walking. Let's examine the change in the node-depth over the phase.

When the algorithm performs an up-walk, the current node-depth decreases by one. After following the suffix link, it decreases by one again. In the down-walk step, the current node-depth is increased by some number bounded by the height of the current tree $\mathcal{T}_i$, which cannot exceed n. Hence, the total amount of work processing the phases in algorithm is $\mathcal{O}(n)$. □

To summarize what was shown, by applying appropriatly the suffix links and the skip/count trick, the Ukkonen's algorithm can be done in $\mathcal{O}(m^2)$ time.

3.3.4 Edge-label compression

There is an important obstacle which cannot be taken lightly. As of this moment, the suffix tree may require $\Theta(n^2)$ space.

Consider a string $S = \{\texttt{abcdefghijklmnopqrstuvwxyz}\}$. The suffix tree for S will have 26 edges from the root, which makes the sum of characters of the edge labels proportional to n^2.

To reduce the space usage, instead of storing the whole substring in the edge, store the pair of indices pointing to the start and the end position in string S. This labelling technique is called **edge-label compression** and by doing so, each edge will have a constant amount of memory, thus reducing it asymptotically to $\Theta(n)$. This scheme will also help the next tricks to shrink down the time complexity. The visualization of a compressed edge-label can be seen in Figure 3.5.

With this scheme, the suffix extension would change accordingly:

Rule 1 Adding a character $S[i+1]$ will just update the edge label (j, i) to $(j, i+1)$.

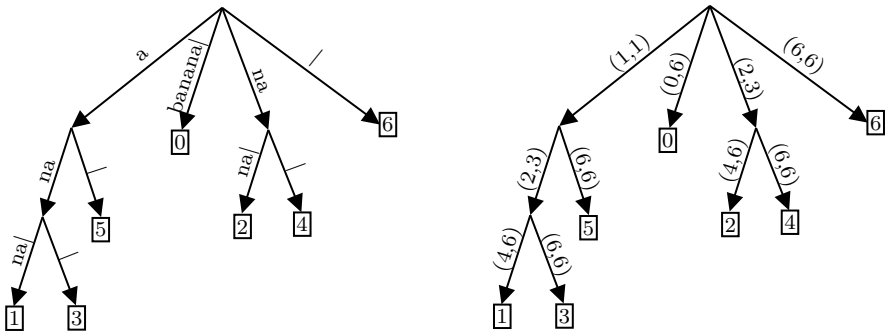

Figure 3.5: An example of edge-label compression on the suffix tree with $S =$ `banana|`.

Rule 2 Consider an edge (u_1, u_2) with edge label (i, j), where a new internal node v will be inserted at position k. *Rule 2* will create three new edges: (u_1, v) with label (j, k), (v, u_2) with label $(k + 1, i)$ and (v, j) with label $(i + 1, i + 1)$.

Rule 3 No changes are made.

Finally, two more speed-up tricks in the combination with the previous ones, and we'll get immediately the desired linear time bound. It could seem that in a given phase, the extensions are being applied in unordered manner, however it is not true. Application of suffix extensions rules actually happens in order, i.e. first the *Rule 1* is applied, then *Rule 2* and finally *Rule 3*. This fact is very important to state, because the following concepts are built upon this observation.

3.3.5 Trick: Once a leaf, always a leaf

Once the leaf j is created, it will remain being a leaf throughout the algorithm. Recall the suffix extensions – no rules modify the leaf label. In phase $i + 1$, according to the *Rule 1*, we just extend the leaves by $S[i + 1]$. Based on this observation, we can reduce the work further. Eventually, the label *end number* of the leaf would be eventually updated to n. Hence there is no need of explicitly updating the label of a leaf node in every phase. We can just set the end point to ∞, i.e., every edge leading to a leaf can be labelled as (j, ∞) in the intermediate phase.

3.3.6 Trick: Halt condition

The *halt condition* or "**Rule 3 is a show stopper**" [3] reads as follows: If *Rule 3* is applied in extension j of phase $i + 1$, then it will also be applied in the next extensions of the same phase, i.e. if the suffix $S[j..i+1]$ is in the tree,

therefore the string $S[j+1..i+1]$, $S[j+2..i+1]$,...,$S[i+1..i+1]$ must also be in the tree. We simply can just stop the current phase of the algorithm whenever the Rule 3 is applied. The phase is said to be built **implicitly**. **Explicit extensions** are then the one built by the algorithm (*Rule 2*).

3.3.7 Putting it together

Suppose that *Rule 1* is applied to l extensions of phase $i+1$ and after the extension l, the *Rule 2* will be applied. This is being applied until some extension r, when *Rule 3* is finally being applied from extension $r+1$ to extension $i+1$. The value l is non-decreasing and is equal to the number of current leaves in the tree. Hence, the value r is equal to l plus the number of current internal nodes, and the next phase can start at extension r because the $r-1$ previous extensions were all implicit. The key point is that when starting a new phase $i+2$, we already know where r ends, and because r was the last explicit extension computed in phase $i+1$, we can execute the suffix extension rule for repeated extension r in the phase $i+2$ without any up-walking, suffix link traversals or node skipping, so the first explicit extension in any phase takes only constant time.

With all the tricks correctly used, Ukkonen's algorithm builds a suffix tree in $\mathcal{O}(n)$ total time.

3.4 Applications

The suffix tree has a wide variety of usage in solving the string problems, as mentioned before. Gusfield [3] devoted a whole chapter to the applications of suffix tree in his book. Furthermore, these applications can be classified into the different types of traversals:

Bottom-up traversal a traversal which goes from leaves to root

Top-down traversal a traversal which goes in top-down manner

Traversal with suffix links a traversal utilizing the suffix links

Next, we'll present some chosen algorithms and show what kind of traversal they use.

Finding maximal repeated pairs

The computation of maximal repeated pairs plays a very important key in the study and analysis of genomes. The study shows that 50% of the 3 billion of human genome consist of repeats. 11% of mustard weed genome contains repeats, 7% of the worm genome and 3% of the fly genome are repeats [18].

Definition 3.5 (Repeated pair)**.** A pair $((i_1, j_1), (i_2, j_2))$ is a *repeated pair* iff $(i_1, j_1) \neq (i_2, j_2)$ and $S[i_1..j_1] = S[i_2..j_2]$. The length of such pair is $j_1 - i_1 + 1$. A repeated pair is called *left maximal* if $S[i_1 - 1] \neq S[i_2 - 1]$ and *right maximal* if $S[j_1 + 1] \neq S[j_2 + 1]$. A repeated pair is called *maximal* if it's left and right maximal.

Definition 3.6 (Repeat)**.** A substring ω of S is a *(maximal) repeat* if there is a (maximal) repeated pair $((i_1, j_1), (i_2, j_2))$ such that $\omega = S[i_1..j_1]$. A *supermaximal repeat* is a maximal repeat that never occurs as a substring of any other maximal repeat.

For example, consider the string $S =$ `xabcyiizabcqabcyr`, where there are three occurences of the substring `abc` at positions `1, 8` and `12`. The first and the second occurrence form a maximal pair, the second and third also form a maximal pair, whereas the first and third don't, because the occurrences are not right maximal. The occurrences of `abcy` also form a maximal pair.

Note 2. The definition allows two substrings in a maximal pair to overlap each other. For example the string `aabaabaa` which contains a maximal repeat `aabaa`.

Using a suffix tree and bottom-up traversal, it is possible to find all the maximal repeated pairs in $\mathcal{O}(n)$ for a string of length n. The following lemma states a necessary condition for a substring to be a maximal repeat.

Lemma 3.4. *Let $\mathcal{T}$ be the suffix tree for string S. If a string α is a maximal repeat in S, then α is the path label of a node v in $\mathcal{T}$.*

Proof. If α is a maximal repeat then there must be at least two occurrences of α in the string S, where the character to the right of the first occurrence differs from the right character of the second occurrence. Hence, α is a path label of a node v in $\mathcal{T}$. □

Now we know that to find maximal repeated pairs, we only need to consider the internal nodes of the suffix tree $\mathcal{T}$. However, it is not yet clear which nodes corresponds to the maximal repeat.

Definition 3.7 (Left character)**.** Consider a string S of length n. For each position i, $0 < i < n$ in the string, character $S[i - 1]$ is a *left character* of i.

Definition 3.8 (Left diverse)**.** A node v of $\mathcal{T}$ is *left diverse* if at least two leaves in v's subtree have different left characters. By definition, a leaf cannot be left diverse. Note that left diversity propagates upward, so if v is diverse, so are all v's ancestors.

Theorem 3.3. The node v with a path label α is a maximal repeat iff v is left diverse.

Proof. We'll show both implications:

1. ($\Rightarrow$): Suppose α is a maximal repeat. Then it participates in a maximal pair, therefore the occurrences of α must have distinct left characters. Hence v must be left diverse.

2. ($\Leftarrow$): Suppose v is left diverse. That means there are substrings $x\alpha$ and $y\alpha$ in S, where x and y are distinct characters. Let p be a character that follows the first substring and p' a character that follows the second.

   If $p' \neq p$, then α is a maximal repeat and the theorem is proved.

   Suppose two substrings are $x\alpha p$ and $y\alpha p$. Since v is a (branching) node, there must be a substring αq in S for some character $q \neq p$. If this occurence is preceded by character x, then it forms of maximal pair with string $y\alpha p$, if it is preceded by y, then it forms in a maximal pair with $x\alpha p$. In either case, α cannot be preceded by both x and y, because v is left diverse. Hence, α must be part of a maximal pair.

□

The general idea is to record the left character for each leaf. When traversing the tree in a bottom-up manner, for each node v and character c, record the leaf numbers below v whose left character is c. The pseudocode of the algorithm is seen in Algorithm 3.4.1.

Algorithm 3.4.1: Find all maximal repeated pairs

Input : suffix tree $\mathcal{T}$ for string S
Output: list of all maximal repeated pairs

```
1 pairs ← ∅
2 Traverse T in a bottom-up manner
3 for each node v with path label α do
4 |   for each pair of children x ≠ y and pair of characters c1 ≠ c2 do
5 |   |   pairs ← append the cartesian product of lists (x, c1) and (y, c2)
6 |   create the list of left characters for node v by linking the lists of
  |     v's children.
7 return pairs
```

Ziv-Lempel decomposition

As the second application, we have chosen the *Ziv-Lempel decomposition* [19, 20]. This plays an important role in data compression and is widely used. The following algorithm is based on Gusfield's [3] with the change that it keeps matching until no further matches are possible (overlapping cases).

Definition 3.9. Let S be a string of length n. We define substring $Prior_i$ as the longest prefix of $S[i..n-1]$ that also occurs as a substring of $S[0..i-1]$.

Definition 3.10. Let S be a string of length n. For any position i in S, define l_i as the length $Prior_i$. For $l_i > 0$, define s_i as the starting position of the leftmost copy of $Prior_i$.

Implementation using the suffix tree can compute l_i and s_i in $\mathcal{O}(n)$ time. Before the compression, the algorithm builds a suffix tree $\mathcal{T}$ for S and assign to each node v the number c_v. The number represents the smallest suffix number of any leaf in v's subtree and it gives the starting position of the leftmost copy of the substring that labels the path from the root to v. Suffix tree can be built in $\mathcal{O}(n)$ time, and computing the numbers can be obtained in $\mathcal{O}(n)$ by bottom-up propagation.

Algorithm 3.4.2: Compute the s and l array

Input : suffix tree $\mathcal{T}$ with assigned c numbers
Output: array s and l

```
1 for i in 0..n − 1 do
2     p ← current point
3     v ← first node at or below p
4     while string_depth(p) + c_v ≤ i or no further matches are
        possible do
5         traverse the unique path in T that matches a prefix of
            S[i..n − 1]
6     l_i ← string_depth(p)
7     s_i ← c_v
8 return s, l
```

When the algorithm needs to compute (s_i, l_i) for position i, it traverses the unique path in $\mathcal{T}$ that matches a prefix of $S[i..n-1]$. The traversal ends at point p, which isn't necessarily a node, either when i equals to `string_depth(`p`)`$+c_v$, where v is the first node at or below p or no further matches are possible.

In either case, the path label to p represents the longest prefix of $S[i..n-1]$ that also occurs in $S[0..i]$. Taking advantage of the fixed alphabet, the time to find (s_i, l_i) is $\mathcal{O}(l_i)$.

Exact string matching

First problem, which we are going to be concerned about and is usually solved by the top-down traversal, is *exact string matching*. Exact string matching is being heavily studied due to its applications in various fields. The problem

states as follows: given a pattern P of length m and a text T of length n, find all occurences of P in T.

We can categorize the exact string matching into three cases. When the text and pattern is known to the algorithm at the same time, then the suffix tree achieves the same time complexity $\mathcal{O}(n+m)$ as the Knuth-Morris-Pratt or Boyer-Moore algorithms. However, the case when the text T is known first and kept for a long time, and then comes a long input sequence of pattern is much more often. Using a suffix tree for T, all occurrences can be found in $\mathcal{O}(m+k)$ time, where k is the number of occurrences of P in T. The prime motivation for developing suffix trees was that any pattern (unknown at the preprocessing phase) can be found in time proportional to its length, after initial linear preprocessing T. In contrast, the mentioned methods such as Knuth-Morris-Pratt or Booyer-Moore, require $\mathcal{O}(m)$ for processing, and $\mathcal{O}(n)$ for the search.

The third case, where the pattern is first fixed and being preprocessed before the text is known, is the classic situation handled by the mentioned methods earlier. This case will be discussed in the next section along with the topic of traversing with the suffix links.

For the top-down traversal, we'll have a look at the second case and present some algorithms. The first exact string matching problem is very straightforward. It matches the characters of P along the path label in $\mathcal{T}$ until either P is exhausted or no matches are possible. The former condition can be seen in Algorithm 3.4.3. If no matches are possible, then P doesn't appear anywhere in T.

Algorithm 3.4.3: Exact string matching

Input : suffix tree $\mathcal{T}$ of string S and pattern P
Output: array of occurrences

```
1  EXACT_MATCHING(POS, NODE):
2      do
3          match the path label of the current node with the P.
4      until either the pattern ends or the path label ends;
5      if P is not exhausted then
6          pos ← the position of the P where it stopped matching
7          node ← next node starting with the next symbol of pattern P
8          EXACT_MATCHING(POS, NODE)
9      else
10         TRAVERSE_LEAVES(NODE)        ◁ traverse and store the leaves
```

The key point to understand why it works, is to see that P starting at position j occurs in T if and only if P occurs as a prefix of $T[j..n-1]$. This happens if and only if the pattern P labels an initial part of the path label from the root to leaf j. This is then being followed by the matching algorithm.

Recall that no two edges out of a node can start with the same character, therefore it is guaranteed that the matching path is unique. Assuming a finite alphabet, the work at each node takes constant time and the total time is proportional to the length of P, making it $\mathcal{O}(m)$.

To collect all the occurrences of P, traverse the subtree at the end of the matching path, and store all the leaves encountered. This can be done in any linear-time traversal, thus making it $\mathcal{O}(m+k)$, where k is the number of occurrences, i.e. leaves.

Finding all shortest unique substrings

As the second application of the top-down traversal, we will briefly describe how to *find all the shortest unique substrings*. The problem is relevant when designing primers for DNA sequences [1].

A substring of S is called *unique* if it occurs only once in S. To find all the unique shortest substrings we traverse the tree in the top-down manner. Let n be the internal node just before the leaves. When we reach any internal node n, it is guaranteed that the string path is unique, because no two edges out of an internal node can have the same following characters. Hence, the string path of the node n plus the following character forms a unique substring. Note that, the substrings ending with the sentinel character are not considered as unique substrings, since the sentinel character is not part of the string.

Algorithm 3.4.4: Find all the shortest unique substrings

Input : suffix tree $\mathcal{T}$ of string S
Output: set M of shortest unique substrings

```
1  TRAVERSE(DEPTH, NODE):
2      if node is leaf and next character ≠ | then
3          if depth+1 < len then
4              M ← {⟨depth + 1, leaf_index⟩}
5              len ← depth+1
6          else if depth+1 = len then
7              M ← M ∪ {⟨depth + 1, leaf_index⟩}
8      else
9          if depth > len then return
10         depth ← string_depth(node)
11         for child in node.children do
12             TRAVERSE(depth, child)
```

To find all the *shortest* unique substrings, we keep a set M and the length of shortest unique substrings, and whenever an internal node is being processed, it checks if the string-depth is less than or equal to the currently shortest

unique substrings. If it is equal, then we insert the substring into the set. If it is less, we clear the set M, set the new shortest length and insert the substring. Because there are $\mathcal{O}(n)$ internal nodes, the algorithm runs overall in the $\mathcal{O}(n)$ time.

Computing matching statistics

As stated in the previous section, there are sometimes situations when pattern(s) will be given first while the text varies which is a reverse role of the normal usage of suffix trees. Naturally, the question would be, whether it is possible solve those problems by building a suffix tree for the patterns, not the text, and still achieve the same time and space bounds as in the Knuth-Morris-Pratt or Aho-Corasick [21] methods. This was solved in a general algorithm by Chang and Lawler [22] called *matching statistics*.

Definition 3.11. Let $ms(i)$ be the length of the longest substring of T starting at position i that matches a substring *somewhere* in P. These values are called the *matching statistics*.

The $ms(i)$ alone doesn't indicate the location of any such match in P. For some applications, it is required to know, the location of at least one such substring for each i. Therefore, The $ms(i)$ is often modified to contain this information.

Definition 3.12. Let $p(i)$ be the number specifing a location in P such that the substring starting at $p(i)$ matches a substring starting at position i of T with length $ms(i)$.

Matching statistics can be used to reduce the overall size of the suffix tree needed in solutions to more complex problems than just exact matching. For example, a problem of *longest common substring*. This would normally require to build a *generalized suffix tree* (a tree representing the suffixes of multiple strings) for string S_1 and S_2. Each internal node v would be marked with an additional symbols denoting the presence of a leaf in the substree of v representing a suffix from the given string. To find the longest common substring, find the node with the greatest string-depth marked by the both symbols. However, this solution uses $\mathcal{O}(|S_1| + |S_2|)$ time and space, and the solution using a suffix tree of a smaller string while still achieving the same worst-case space and time bounds would be much more desirable. Clearly, the longest common substring can be found as the longest matching statistic $ms(i)$. The position in the longer string would be i and the shorter $p(i)$.

Matching statistics also provide a bridge between exact matching methods and problems of approximate string matching and are central to a fast approximate matching method devised for rapid database searching [3].

The suffix links are used to speed up the entire algorithm, similarly to the way it speeds up the construction of a suffix tree in Ukkonen's algorithm.

The naive way would be to match from left to right the characters $T[i..n-1]$, for every $0 \leq i < n$ against the suffix tree by following the unique path label until no further matches are possible. Certainly, this would not achieve the linear-time bound.

Suppose the algorithm has just followed a matching path to learn $ms(i)$ for $0 \leq i < n$. That means, it has located a point b in the suffix tree such that the path to b matches a prefix of $T[i..n-1]$, and no further matches are possible. Now the algorithm proceed to compute $m(i+1)$.

If b is an internal node v of the suffix tree, then the algorithm can follow its suffix link to node $s(v)$. Otherwise, it walks up to the node v just above b. The node v can be either the root or an internal node. If v is root, then the search for $ms(i+1)$ starts at root. For the latter case, it follows the suffix link from v to $s(v)$. Let $x\alpha$ is the path label of v, which is a prefix of $T[i..n-1]$. It follows that α must be a prefix of $T[i+1..n-1]$. From the properties of suffix links, the $s(v)$ has path label α, hence the search for $ms(i+1)$ can start at node $s(v)$ instead of the root.

Algorithm 3.4.5: Matching statistics

Input : suffix tree of string P of length m and string T of length n
Output: the array ms and p

```
1  offset ← 0
2  node ← root
3  for i in 0..n-1 do
4      len, pos, v ← COMPUTE_MS(node, T[i..n-1], offset)
5      ms(i), p(i) ← len, pos
6      offset← len − 1
7      if ms(i) = 0 or ms(i) = 1 and T[i+1] is not in P then
8          ms(i + 1) ← 0
9          skip next iteration
10     if v != root then
11         node = s(v)
12 return ms, p
```

Let β denote the string between node v and point b. Then $x\alpha\beta$ is the longest substring in P matching a substring of T at position i. Thus $\alpha\beta$ is string matching a substring at position $i+1$. If $s(v)$ has path label α, there is guaranteed the path β out of $s(v)$ must exist. From this, we use the *skip/count* trick (described in Section 3.3.3) to traverse the nodes, instead of traversing by matching every character on it.

When it reaches the end of the β path, the algorithm continues to match single characters from T against the characters in the tree until no further matches are possible. The $ms(i+1)$ is then the string-depth of the ending

position. Depending on the result of the search for $ms(i)$ – either there is a mismatch or it ends at the leaf – the character comparisons done after reaching the β path in $ms(i+1)$ begin either with the same character in T that ended the search for $ms(i)$ or with the next character.

There is a special case when computing $ms(i+1)$. If $ms(i)$ is 0 or 1, and $T[i+1]$ is not in P, then $ms(i+1) = 0$.

Algorithm 3.4.6: Compute matching statistics

```
1  COMPUTE_MS(node, T, offset):
2      skips the offset characters          ◁ this corresponds to the β path
3      ▷ match single characters after β path until no further match
4      v ← a node at or above the point where the matching stopped
5      len ← the string-depth of the ending position
6      if it stopped on an edge (u,v) then
7      |   pos ← one of the v's leaf
8      else if it's not on a leaf then
9      |   pos ← one of the u's leaf
10     else
11     |   pos ← a leaf where algorithm stopped
12     return len, pos, v
```

All that remains, is to show the correctness of the algorithm.

Theorem 3.4. Let there be a suffix tree for pattern P of length m and let T be a string of length n be the text. All the matching statistics can be found in $\mathcal{O}(n)$ time.

Proof. The search for any $ms(i+1)$ begins by walking up at most one edge and following the suffix link. This is known to be constant for every i, so this takes $\mathcal{O}(n)$ over the entire algorithm. As for the traversing the β path, this can be proved similarly to the proof of Ukkonen's algorithm. Recall that, the link traversal reduces the current depth by at most one, therefore the time for traversing the β is bounded by $\mathcal{O}(n)$.

It remains to show the comparisons done after applying the skip/count trick of the β path. When computing the $ms(i+1)$, it shares at most one chracter in common with $ms(i)$. It follows that at most $\mathcal{O}(n)$ comparisons are performed in total after the β path. All the work of the algorithm for computing the matching statistics are bounded by $\mathcal{O}(n)$, hence the theorem is proved. □

Chapter 4

Enhanced Suffix Array

Suffix arrays are very closely related to the suffix trees. Manber and Myers [2] introduced suffix arrays as a more space-efficient alternative to suffix trees in 1990. Since then, the study of algorithms for suffix array construction and its application in bioinformatics has attracted a lot of attention.

The first direct construction for suffix arrays required $\mathcal{O}(n \log n)$ time, thus leaving a huge difference between the time complexity of constructing the suffix trees. While suffix arrays can be induced from suffix trees, which would imply linear time construction, it would not achieve the purpose of saving the space.

Fortunately, this gap is closed by numerous researchers. One of the algorithms to achieve the optimal space and time complexity is the *SA-IS* algorithm [23] of Nong, Zhang & Chan devised in 2009. It is considered as one of the fastest known suffix array construction algorithms. The currently fastest algorithm is *DivSufSort* which wasn't documented until Fischer and Kurpicz [24] gave a concise description of it in 2017.

In this chapter, we will present additional data structures that will assist in solving the problems that are usually solved by the suffix tree. Specifically, it will be shown on a few chosen algorithms from each category of the tree traversals to show that it is possible to replace the suffix tree with ESA. These to be presented concepts are based on the [1].

Before we begin with the definition of a suffix array, a little reminder needs to be put in place. As mentioned in the previous chapter, the string S is terminated with the *sentinel* character.

Definition 4.1 (Suffix Array)**.** Let S be a string of length n. The *suffix array* of S, often denoted as `suftab`, is an array of integers indicating the lexicographic order of suffixes of the string S.

Note 3. Assuming that $n < 2^{32}$, the most primitive form of suffix array needs only $4n$ bytes.

4.1 BWT table

The ESA can also contain a bwt table, which contains the *Burrows and Wheeler* transformation [25], known primarily in the data compression. It is denoted as `bwttab` and is an array of size n. For every i, $0 \leq 0 < n$, $\texttt{bwttab}[i] = S[\texttt{suftab}[i] - 1]$ if $\texttt{suftab}[i] \neq 0$. The table can be constructed in $\mathcal{O}(n)$ time by scanning over the suffix array. One of the usage in the thesis is in the application to solve the problem of maximal pairs. Instead of storing the whole text, we can just store the `bwttab`.

4.2 LCP array

The first data structure to be presented, often used in association with suffix array, is called *LCP array*.

Definition 4.2 (LCP array)**.** The LCP array stores the lengths of the longest common prefixes between two adjacent suffixes. The LCP array is often denoted as `lcptab`.

We define $\texttt{lcptab}[0] = 0$, and since the last character is unique in the whole string, the $\texttt{lcptab}[n-1] = 0$.

Manber and Myers, along with the suffix array, presented an algorithm for constructing the *LCP array* in $\mathcal{O}(n \log n)$ worst-case time and $\mathcal{O}(n)$ expected time, until Kasai et al. [26] showed that *LCP array* can be constructed directly from the suffix array in a linear time. However, assuming every entry of an array takes 4 bytes, this algorithm needs an additional array called *rank array* and it would occupy total $8n$ bytes, in contrast to the output which would be only $4n$ bytes. Thereafter, a lot of improvements were devised, which would decrease space usage [27] or improve time efficiency [28]. Recently, it was shown, that it is possible to combine the inducing suffix array algorithms to produce the LCP, which in return gives a very fast algorithms to produce both of the needed structures [29, 24].

4.2.1 Lcp-intervals

The first fundamental concept to be presented is the *lcp-intervals* and their *lcp-intervals tree* form.

Definition 4.3 (lcp-interval)**.** An interval $[i..j]$, $0 \leq i < j < n$, is called *lcp-interval of lcp-value* ℓ if

1. $\texttt{lcptab}[i] < \ell$,
2. $\texttt{lcptab}[k] > \ell$ for all k where $i + 1 \leq k \leq j$,
3. $\texttt{lcptab}[k] = \ell$ for at least one k where $i + 1 \leq k \leq j$,
4. $\texttt{lcptab}[j + 1] < \ell$.

Suffix array and LCP			
i	sa	lcp	$S_{\texttt{suftab[i]}}$
0	2	0	aaacatat|
1	3	2	aacatat|
2	0	1	acaaacatat|
3	4	3	acatat|
4	6	1	atat|
5	8	2	at|
6	1	0	caaacatat|
7	5	2	catat|
8	7	0	tat|
9	9	1	t|
10	10	0	|

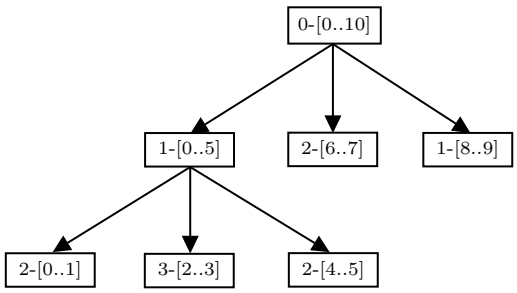

Figure 4.1: An example of lcp-interval tree of string $S =$ `acaaacatat|`.

The lcp-interval $[i..j]$ of lcp-value ℓ can also be denoted as ℓ-interval or ℓ-$[i..j]$. Every index k that holds $\texttt{lcptab}[k] = \ell$ for $i + 1 \leq k \leq j$, is called ℓ-index. Such a set of ℓ-indices of an ℓ-$[i..j]$ is denoted *ℓIndices(i, j)*.

If ℓ-$[i..j]$ is such an interval that $\omega = S[\texttt{suftab}[i]..\texttt{suftab}[i] + \ell - 1]$ is the longest common prefix of suffixes $S_{\texttt{suftab[k]}}$ for every $i \leq k \leq j$, then $[i..j]$ is called ω-interval.

An example can be seen on Figure 4.1. Consider an interval [0..5]. This interval is 1-interval, because $\texttt{lcptab}[0] = 0 < 1$, $\texttt{lcptab}[k] \geq 1$, for $0 < k \leq 5$ and $\texttt{lcptab}[6] = 0 < 1$. The *ℓIndices*$(0,\ 5) = [2, 4]$, and the interval is a-interval.

4.2.2 Lcp-interval tree

The second essential concept is a representation of the lcp-intervals. The lcp-interval tree is only a conceptual scheme, so it is not really built by the algorithm and it allows us to simulate the traversal very efficiently.

Definition 4.4. We say that an m-interval $[l..r]$ is *embedded* in an ℓ-interval $[i..j]$ if it is a sub-interval of $[i..j]$, i.e. $i \leq l < r \leq j$ and the lcp-value of the $[l..r]$ is greater than the lcp-value of $[i..j]$, i.e. $m > \ell$. We also say that $[i..j]$ *encloses* $[l..r]$ in this case.

If $[i..j]$ encloses $[l..r]$ and there is no another interval embedded in $[i..j]$ which would also enclose $[l..r]$, then $[l..r]$ is called a *child interval* of $[i..j]$. Conversely $[i..j]$ is called *parent interval* of $[l..r]$.

The idea behind ℓ-intervals is that they correspond to the internal nodes of the suffix tree. The leaves are being deliberately left in this concept, but every leaf in the suffix tree can be represented by a *singleton interval* $[l..l]$.

Kasai et al. [26] showed that it is possible to simulate every bottom-up traversal of a suffix tree with just a suffix array and an LCP array.

In order to perform a bottom-up traversal, we keep the nested lcp-intervals with the help of a stack (operations `push`, `pop` and `top`). The elements on the stack are the lcp-intervals represented by a tuple $\langle lcp,\ lb,\ rb\rangle$: *lcp* is the lcp-value of the interval, *lb* is its left boundary and *rb* is its right boundary.

Algorithm 4.2.1: Process lcp-intervals

```
1  push(⟨0, 0, ⊥⟩)
2  for i in 1..n − 1 do
3      lb ← i − 1
4      while lcptab[i] < top.lcp do
5          top.rb ← i − 1
6          interval ← pop()
7          process(interval)
8          lb ← interval.lb
9      if lcptab[i] > top.lcp then
10         push(⟨lcptab[i], lb, rb⟩)
```

With the Algorithm 4.2.1, we can generate all the lcp-intervals in bottom-up manner. But in order to perform a meaningful traversal, the information from the children of currently processed lcp-intervals needs to be known. The next theorem shows how the relationship of the lcp-intervals can be determined from the stack operations and has an important consenquence for the correctness of the derived algorithm which takes the information of the child nodes in account.

Theorem 4.1. Let *top* be the top-most interval on the stack and top_{-1} be the one next to it on the stack ($top_{-1} < top.lcp$). If $\texttt{lcptab}[i] < top.lcp$, then before *top* will be popped off the stack in the while loop, the following holds:

1. If $\texttt{lcptab}[i] \leq top_{-1}.lcp$, then *top* is the child interval of top_{-1}
2. If $top_{-1}.lcp < \texttt{lcptab}[i] < top.lcp$, then *top* is the child interval of the `lcptab`[i]-interval that contains i.

Proof. Both cases are very similar. The for-loop of Algorithm 4.2.1 is maintaining this: if $\langle \ell_1,\ lb_1,\ rb_1\rangle, ..., \langle \ell_k,\ lb_k,\ rb_k\rangle$ are intervals on the stack, where top $= \langle \ell_k,\ lb_k,\ rb_k\rangle$, then $lb_i \leq lb_j$ and $\ell_i < \ell_j$ for all $1 \leq i < j \leq k$. From the stack properties, the $\langle \ell_j,\ lb_j,\ rb_j\rangle$ will be popped before $\langle \ell_i,\ lb_i,\ rb_i\rangle$. It follows that $rb_j \leq rb_i$, thus the ℓ_j-interval $[lb_j..rb_j]$ is embedded in the ℓ_i-interval $[lb_i..rb_i]$, i.e. *top* is embedded in top_{-1}.

Assume *top* was not the child of top_{-1}. That would mean, there is such an lcp-interval top', that encloses *top* and is embedded in top_{-1}. This can happen only if top' is on the stack above the top_{-1}. □

Thus we can extend the lcp-interval by a child information so the element of the stack would be represented by a quadruple $\langle lcp, lb, rb, childList \rangle$, where *childList* is a list of its child intervals. In case (1), we add *top* to the child list of top_{-1} and top_{-1} is popped next. Otherwise (case (2)), the while loop is left without assigning a parent for *top*. The modified algorithm can be seen at Algorithm 4.2.2.

Algorithm 4.2.2: Process lcp-intervals with child information

```
1  lastInterval ← ⊥
2  push(⟨0, 0, ⊥, [ ]⟩)
3  for i in 1..n − 1 do
4      lb ← i − 1
5      while lcptab[i] < top.lcp do
6          top.rb ← i − 1
7          lastInterval ← pop()
8          process(lastInterval)
9          lb ← lastInterval.lb
10         if lcptab[i] ≤ top.lcp // case (1)
11          then
12             top.childList ← [top.childList, lastInterval]
13             lastInterval ← ⊥
14     if lcptab[i] > top.lcp then
15         if lastInterval ≠ ⊥ // case (2)
16          then
17             push(⟨lcptab[i], lb, rb, [lastInterval]⟩)
18             lastInterval ← ⊥
19         else
20             push(⟨lcptab[i], lb, rb, [ ]⟩)
```

Many problems solved by the suffix tree bottom-up traversal can be solved by merely specifying the function `process` called on line 7. This function accepts the interval with the information about its child intervals. The specifications of the function are presented in the section 4.5 *Applications*.

4.3 Child-table

It would be convenient to determine all child intervals of any ℓ-interval $[i..j]$ in a constant time. This can be achieved by enhancing the suffix array (along with the lcp table), with an additional table: the *child-table*, often denoted as `childtab`. Furthermore, this will solve the problems which are usually solved by the top-down traversal of a suffix tree. Each entry of the child-table contains three values: *up*, *down* and *nextℓIndex*.

For an ℓ-interval $[i..j]$ with ℓ-indices $i_1 < i_2 < ... < i_k$, the first ℓ-index can be obtained from $\texttt{childtab}[i].\texttt{down} = i_1$ or $\texttt{childtab}[j+1].\texttt{up} = i_1$. The rest ℓ-indices are defined as $\texttt{childtab}[i_p].\texttt{next}\ell\texttt{Index} = i_{p+1}$ for all p in $1 \leq p < k$.

Definition 4.5. The entries of `childtab` are defined as follows (undefined values are set to $\perp$):

$$\texttt{childtab}[i].\texttt{up} = min\left\{ q \in [0..i-1] \;\middle|\; \begin{array}{l} \texttt{lcptab}[q] > \texttt{lcptab}[i], \\ \forall k \in [q+1..i-1] : \texttt{lcptab}[k] \geq \texttt{lcptab}[q] \end{array} \right\}$$

$$\texttt{childtab}[i].\texttt{down} = max\left\{ q \in [i+1..n-1] \;\middle|\; \begin{array}{l} \texttt{lcptab}[q] > \texttt{lcptab}[i], \\ \forall k \in [i+1..q-1] : \texttt{lcptab}[k] > \texttt{lcptab}[q] \end{array} \right\}$$

$$\texttt{childtab}[i].\texttt{next}\ell\texttt{Index} = max\left\{ q \in [i+1..n-1] \;\middle|\; \begin{array}{l} \texttt{lcptab}[q] = \texttt{lcptab}[i], \\ \forall k \in [i+1..q-1] : \texttt{lcptab}[k] > \texttt{lcptab}[i] \end{array} \right\}$$

Once the ℓ-indices are known, the child intervals can be found according to the lemma below.

			childtab			
i	sa	lcp	up	down	nextℓ...	$S_{\texttt{suftab[i]}}$
0	2	0		2	6	aaacatat|
1	3	2				aacatat|
2	0	1	1	3	4	acaaacatat|
3	4	3				acatat|
4	6	1	3	5	2	atat|
5	8	2				at|
6	1	0	2	7	8	caaacatat|
7	5	2				catat|
8	7	0	7	9	10	tat|
9	9	1				t|
10	10	0	9			|

Table 4.1: Suffix array of string S = `acaaacatat|` enhanced with the lcp array and child-table

Lemma 4.1. *Assume we have an ℓ-interval $[i..j]$ with ℓ-indices $i_1 < ... < i_k$. Then the child intervals of $[i..j]$ are $[i..i_1 - 1]$, $[i_1..i_2 - 1]$,..., $[i_k..j]$.*

Proof. Let $[l..r]$ be one of the child interval of $[i..j]$. It can be either a singleton interval or an m-interval. If $[l..r]$ is a singleton interval, then it is a child interval of $[i..j]$. Suppose the latter case, where $[l..r]$ is an m-interval. Since $[l..r]$ does not contain an ℓ-index, it means that $[l..r]$ is embedded in $[i..j]$. Because $\texttt{lcptab}[i_1] = \texttt{lcptab}[i_2] = ... = \texttt{lcptab}[i_k] = \ell$, there is no interval embedded in $[i..j]$ that is enclosing the $[l..r]$. Hence, $[l..r]$ is a child interval of $[i..j]$. □

The example of the child-table can be seen on the Table 4.1. For instance, the interval 1-[0..5] has 1-indices 2 and 4. The first 1-index is stored in `childtab`[0].`down` and `childtab`[6].`up`, the second 1-index is stored in `childtab`[2].`next`ℓ`Index`. Thus, all the child intervals of the interval 1-[0..5] are [0..1], [2..3], and [4..5].

4.3.1 Construction

Child-table can be constructed in a bottom-up manner by using the lcp-interval tree in linear time. The Algorithm 4.3.1 is very similar to the Algorithm 4.2.2. To show the algorithm works corretly, we need the following lemma.

Algorithm 4.3.1: Construction of *up/down* and *nextℓIndex* values

```
1  lastIndex ← −1
2  push(0)
3  for i in 1..n − 1 do
4      while lcptab[i] < lcptab[top] do
5          lastIndex ← pop()
6          if (lcptab[i] ≤ lcptab[top]) and (lcptab[top] ≠
            lcptab[lastIndex]) then
7              childtab[top].down ← lastIndex
       /* now lcptab[i] ≥ lcptab[top] holds                    */
8      if lcptab[i] = lcptab[top] then
9          childtab[i].nextℓIndex ← i
10     if lastIndex ≥ −1 then
11         childtab[i].up ← lastIndex
12         lastIndex ← −1
13     push(i)
```

Lemma 4.2. *Let $i_1, ..., i_p$ be the indices on the stack (where i_p is the topmost element), then $i_1 < ... < i_p$ and $\texttt{lcptab}[i_1] \leq ... \leq \texttt{lcptab}[i_p]$. Moreover, if $\texttt{lcptab}[i_j] < \texttt{lcptab}[i_j + 1]$, then for all k with $i_j < k < i_j + 1$, we have $\texttt{lcptab}[k] > \texttt{lcptab}[i_j + 1]$.*

Proof. By induction.

Before the algorithm executes the for-loop for the first time, this lemma holds. Assume it holds after the for-loop has been executed for m times. Consider now $(m+1)$-th execution. Suppose there is an index q with $1 \leq q < p$ such that $\texttt{lcptab}[i_1] \leq ... \leq \texttt{lcptab}[i_q] \leq \texttt{lcptab}[m+1] < \texttt{lcptab}[i_{q+1}] \leq ... \leq \texttt{lcptab}[i_p]$. It follows, that in the while-loop, $i_{q+1}, ..., i_p$ are popped from the stack and afterwards the $m + 1$ is pushed onto the stack, thus being now the topmost element. Clearly, the conditions $i_1 < ... < i_q < m + 1$ and $\texttt{lcptab}[i_1] \leq ... \leq \texttt{lcptab}[i_q] \leq \texttt{lcptab}[m + 1]$ hold. Suppose $\texttt{lcptab}[i_j] < \texttt{lcptab}[i_{j+1}]$. By the inductive hypothesis, for $\forall j \in \{1, ..., p\}$ with $\texttt{lcptab}[i_j] < \texttt{lcptab}[i_{j+1}]$, we have $\texttt{lcptab}[k] > \texttt{lcptab}[i_{j+1}]$ for $\forall k$ with $i_j < k < i_{j+1}$. It is not difficult to see that this is a consequence of the while-loop. □

Let's summarize it with the following theorem and show that the algorithm correctly fills up the values.

Theorem 4.2. Algorithm 4.3.1 correctly fills the up/down values of the child-table

Proof. Consider the line `7` (filling the down value) in the Algorithm 4.3.1 is executed. Then the following holds: $\texttt{lcptab}[i] \leq \texttt{lcptab}[top] < \texttt{lcptab}[lastIndex]$ and $top < lastIndex < i$. The down field is the maximum of the set M (defined followingly from the definition 4.5). Surely, $lastIndex \in [top{+}1..n]$ and $\texttt{lcptab}[lastIndex] > \texttt{lcptab}[top]$. Moreover, according to Lemma 4.2, *lastIndex* is an element of M. Suppose *lastIndex* is not the maximum of M. Then there is an element q' in M with $lastIndex < q' < i$ and it follows that $\texttt{lcptab}[lastIndex] > \texttt{lcptab}[q']$. This is a contradiction, because it implies the *lastIndex* would be popped from the stack when q' was being processed. Hence, *lastIndex* is the maximum of M.

Now consider the line `11` (filling the up value) is executed. Then $\texttt{lcptab}[top] \leq \texttt{lcptab}[i] < \texttt{lcptab}[lastIndex]$ and $top < lastIndex < i$. The up field is the minimum of the set M' (defined followingly from the definition 4.5). Clearly, the $lastIndex \in [0..i-1]$ and $\texttt{lcptab}[lastIndex] > \texttt{lcptab}[i]$. Moreover, $lastIndex \in M'$, otherwise it would have been popped from the stack earlier. Now suppose *lastIndex* is not the minimum of M'. Then there is such $q' \in M'$ with $top < q' < lastIndex$, that $\texttt{lcptab}[lastIndex] \geq \texttt{lcptab}[q'] > \texttt{lcptab}[i] \geq \texttt{lcptab}[top]$, hence this is a contradiction. □

4.3.2 Determining child intervals

The first step to locate all the child intervals of ℓ-interval $[i..j]$ is to find the first ℓ-index. The first ℓ-index is either in the *up* or *down* field of the child-table. This will be more clear with the following lemma.

Lemma 4.3. *For every ℓ-interval $[i..j]$ the following statements hold:*

(1) $i < \texttt{childtab}[j+1].\texttt{up} \leq j$ or $i < \texttt{childtab}[j+1].\texttt{up} \leq j$.

(2) $\texttt{childtab}[j+1].\texttt{up}$ is the first ℓ-index if $i < \texttt{childtab}[j+1].\texttt{up} \leq j$.

(3) $\texttt{childtab}[i].\texttt{down}$ is the first ℓ-index if $i < \texttt{childtab}[i].\texttt{down} \leq j$

Proof. (1) Suppose $\texttt{lcptab}[j+1] = \ell'$ $(\ell > \ell')$ and let I' be the corresponding ℓ'-interval. If $[i..j]$ is a child interval I', then $\texttt{lcptab}[i] = \ell'$ and because there is no ℓ-index in $[i+1..j]$, it follows that $\texttt{childtab}[j+1].\texttt{up} = \min\ell\mathit{Indices}(i,j)$. Therefore $i < \texttt{childtab}[j+1].\texttt{up} \leq j$. Otherwise, if $[i..j]$ is not a child interval of I', then suppose the $\texttt{lcptab}[i] = \ell''$ and let I'' be the corresponding ℓ''-interval. Surely, $\ell > \ell'' > \ell' = \texttt{lcptab}[j+1]$, which means that $[i..j]$ is a child interval of I''. Hence, $\texttt{childtab}[i].\texttt{down} = \min\ell\mathit{Indices}(i,j)$, hence $i < \texttt{childtab}[i].\texttt{down} \leq j$.

(2) It follows from:

$$\begin{aligned}\texttt{childtab}[j+1].\texttt{up} &= \mathit{min}\left\{q \in [i+1..j] \;\middle|\; \begin{array}{l}\texttt{lcptab}[q] > \texttt{lcptab}[j+1],\\ \forall k \in [q+1..j] : \texttt{lcptab}[k] \geq \texttt{lcptab}[q]\end{array}\right\}\\ &= \mathit{min}\left\{q \in [i+1..j] \mid \forall k \in [q+1..j] : \texttt{lcptab}[k] \geq \texttt{lcptab}[q]\right\}\\ &= \mathit{min}\ell\mathit{Indices}(i,j)\end{aligned}$$

(3) Let i_1 be the first ℓ-index of interval $[i..j]$. Then $\texttt{lcptab}[i_1] = \ell > \texttt{lcptab}[i]$ and for $\forall k \in [i+1..i_1-1]$: $\texttt{lcptab}[k] > \texttt{lcptab}[i_1]$. Furthermore, $\forall q \in [i+1..j]$: $\texttt{lcptab}[q] \geq \ell > \texttt{lcptab}[i]$ but *not* $\texttt{lcptab}[i_1] > \texttt{lcptab}[q]$. □

Once the first ℓ-index of an ℓ-interval $[i..j]$ is found, the rest can be obtained from the field *nextℓIndex* of every other ℓ-index. See the Algorithm 4.3.2. The algorithm runs in constant time $\mathcal{O}(|\Sigma|)$, where $|\Sigma|$ is the number of child intervals of the ℓ-interval currently being processed. With the algorithm, it is possible to simulate any top-down traversal of a suffix tree with the ESA. Thanks to the Lemma 4.3, it is also very simple to implement a function to return the lcp-value of an lcp-interval $[i..j]$ in constant time.

4.4 Suffix link table

In this section, we focus on the suffix link from the suffix tree and add the similar concept into the ESA. We start with some basic definitions and lemmas established from [1]. We denote inverse suffix array as $\texttt{suftab}^{-1}$ such that $\texttt{suftab}^{-1}[\texttt{suftab}[q]] = q$ for every $0 \leq q < n$.

Algorithm 4.3.2: Get child intervals of an ℓ-interval

```
1  intervalList ← [ ]
2  if i = 0 and j = n-1 // for the ℓ-interval [0..n − 1]
3   then
4      while childtab[i1].nextℓIndex ≠ ⊥ do
5          i2 ← childtab[i1].nextℓIndex
6          intervalList ← [intervalList, (i1, i2 − 1)]
7          i1 ← i2
8      intervalList ← [intervalList, (i1, n − 1)]
9  else
10     i1 ← getFirstℓIndex(i, j)
11     intervalList ← [intervalList, (i, i1 − 1)]
12     while childtab[i1].nextℓIndex ≠ ⊥ do
13         i2 ← childtab[i1].nextℓIndex
14         intervalList ← [intervalList, (i1, i2 − 1)]
15         i1 ← i2
16     intervalList ← [intervalList, (i1, j)]
17 return intervalList
```

Definition 4.6. Let $S_{\texttt{suftab}[i]} = a\omega$. If j, $0 \leq j < n$, satisfies $S_{\texttt{suftab}[i]} = \omega$, then we denote j by $\texttt{link}[i]$ and call it the suffix link (index) of i.

Lemma 4.4. *If* $\texttt{suftab}[i] < n$, *then* $\texttt{link}[i] = \texttt{suftab}^{-1}[\texttt{suftab}[i]+1]$.

Proof. Let $S_{\texttt{suftab}[i]} = a\omega$. Since $S_{\texttt{suftab}[i]+1} = \omega$, $\texttt{link}[i]$ must satisfy $\texttt{suftab}[\texttt{link}[i]] = \texttt{suftab}[i] + 1$. □

Definition 4.7 (Suffix link interval)**.** Given ℓ-interval $[i..j]$, the smallest lcp-interval $[l..r]$ satisfying $l \leq \texttt{link}[i] < \texttt{link}[j] \leq r$ is called the *suffix link interval* of $[i..j]$.

Lemma 4.5. *Given the* $a\omega$*-interval* ℓ*-*$[i..j]$*, its suffix link interval is the* ω*-interval, which has lcp-value* $\ell - 1$.

Proof. Consider an lcp-interval $[i..j]$ and let $[l..r]$ be its suffix link interval. Because the lcp-interval $[i..j]$ is the $a\omega$-interval, $a\omega$ is the longest common prefix of the suffixes in the interval $[i..j]$. It follows, that ω is the longest common prefix of the suffixes in the interval $[l..r]$, because $[l..r]$ is the smallest lcp-interval that $l \leq \texttt{link}[i] < \texttt{link}[j] \leq r$. Thus, $[l..r]$ is the ω-interval and its lcp-value is $\ell - 1$. □

			childtab			suflink			
i	sa	lcp	up	down	nextℓ...	l	r	sa^{-1}	$S_{\texttt{suftab[i]}}$
0	2	0		2	6			2	aaacatat|
1	3	2				0	5	6	aacatat|
2	0	1	1	3	4	0	10	0	acaaacatat|
3	4	3				6	7	1	acatat|
4	6	1	3	5	2			3	atat|
5	8	2				8		7	at|
6	1	0	2	7	8			4	caaacatat|
7	5	2				0	5	8	catat|
8	7	0	7	9	10			5	tat|
9	9	1				0	10	9	t|
10	10	0	9					10	|

Table 4.2: ESA of string S = `acaaacatat|` with additional suflink table

4.4.1 Construction

In order to integrate the suffix links, we compute the suffix link interval $[l..r]$ for every ℓ-interval $[i..j]$ and store the right boundaries l and r at the first ℓ-index of $[i..j]$. Such a table of these values is denoted `suflink`. The example of the `suflink` can be seen on the Table 4.2.

To compute the table, the lcp-interval tree is traversed in breadth first left-to-right manner. For every lcp-value, we create a list of intervals having such lcp-value, so whenever an ℓ-interval is computed, it is added to the list corresponding with its lcp-value. This list is called ℓ-list and is automatically sorted by the left-boundary of the intervals in ascending order. Taking the example from Figure 4.1, we have:

$$0 - \text{list} : [0..10]$$
$$1 - \text{list} : [0..5], [8..9]$$
$$2 - \text{list} : [0..1], [4..5], [6..7]$$
$$3 - \text{list} : [2..3]$$

For every lcp-value $\ell > 0$ and every ℓ-interval $[i..j]$ in ℓ-list, compute the `link`$[i]$ according to the Lemma 4.4 and then search in the $(\ell - 1)$-list for the interval $[l..r]$ such that l is the biggest left boundary of all $(\ell - 1)$-intervals with $l \leq$ `link`$[i]$. As the total number of intervals in the ℓ-list is at most n, the search can done in $\mathcal{O}(\log n)$ time by using the binary search. The search is done for every ℓ-interval (at most n), therefore the whole process would take $\mathcal{O}(n \log n)$ time. The table along with the inverse suffix array require $\mathcal{O}(n)$ space and can be deleted after the preprocessing phase. This is however very slow and lose the optimal time complexity to construct the ESA.

Fortunately, the construction can be done in a linear time. We reduce the problem of constructing the suffix link intervals to the problem of answering range minimum queries (RMQ). But, in contrast to the previous method, we store the boundaries of an ℓ-interval $[i..j]$ at *every* ℓ-index. With the following lemma, it will be clear that it is possible to compute the suffix link interval $[l..r]$ of an ℓ-interval $[i..j]$ in constant time.

Lemma 4.6. *Let $[i..j]$ be an ℓ-interval and let $[l..r]$ be its suffix link interval. Since there is an ℓ-index q, $i+1 \leq q \leq j$, there is also an index k such that k is $(\ell-1)$-index of $[l..r]$ and* $\texttt{link}[i] + 1 \leq k \leq \texttt{link}[j]$.

Proof. Follows from the proof of Lemma 4.5. □

Since $l \leq \texttt{link}[i] + 1 \leq \texttt{link}[j] \leq r$ and lcp-value of $\texttt{link}[i]$ and $\texttt{link}[j]$ is $\ell - 1$, the minimum value of the lcp-table in the range $[\texttt{link}[i]{+}1..\texttt{link}[j]]$ is $\ell - 1$. Therefore, we can locate an $(\ell-1)$-index k of $[l..r]$ by answering RMQ in the range $[\texttt{link}[i]{+}1..\texttt{link}[j]]$.

Definition 4.8 (Range Minimum Query)**.** Let L be an integer array of size n. Given two positions $0 \leq i < j < n$, the problem of *range minimum query RMQ(i, j)* is to find the position k of an element such that $i \leq k \leq j$ and $\mathrm{L}[k] = \min\{\, L[q] \mid i \leq q \leq j \,\}$.

An *RMQ* can be answered in constant time provided that the array L is properly preprocessed. For the computation of the suffix link intervals, the L array for the *RMQ* problem is set to be the `lcptab`. We begin the process by storing the boundaries of every ℓ-interval $[i..j]$ at every ℓ-index of the ℓ-interval. Afterwards, we traverse the lcp-interval tree in breadth-first order in the left-to-right manner. Suppose the ℓ-interval $[i..j]$ is being processed. First, we compute $\texttt{link}[i]$ and $\texttt{link}[j]$ according to Lemma 4.4 and then evaluate $k = \texttt{RMQ}(\texttt{link}[i]{+}1, \texttt{link}[j])$, which is an $(\ell-1)$-index of the suffix link interval of $[i..j]$. Let l and r be the boundaries of the suffix link interval at index k. Finally, we store the l and r in the suffix link table at the *first* ℓ-index of $[i..j]$. Every step in the procedure takes constant time and space, thus the overall complexity of computing the suffix link intervals is $\mathcal{O}(n)$.

4.5 Applications

In this section, we will look into the problems that were presented in the Chapter 3 *Suffix Tree* and solve them using the ESA framework by simulating the suffix tree traversals.

4.5.1 Bottom-up traversals

Finding maximal repeated pairs

The implementation using the ESA considerably reduces the space requirements and in consenquence, much larger genomes can be searched for repetitive elements. The algorithm requires tables `suftab`, `lcptab` and `btwtab`. The access to the three tables are in sequential order, which is verified to reduce the running time [1].

We begin by introducing some notions: The undefined character is denoted by $\perp$ and we assume that it is different from all characters in Σ. Let $[i..j]$ be an ℓ-interval and u their longest common prefix. Next we define $\mathcal{P}_{[i..j]} = \{\, \texttt{suftab}[r] \mid i \leq r \leq j \,\}$, i.e. the set of all positions p such that u is a prefix of S_p. This set is divided into disjoint and possibly empty sets according to the left character of each position: For any $a \in \Sigma \cup \{\perp\}$ define

$$\mathcal{P}_{[i..j]}(a) = \begin{cases} \{\, 0 \mid 0 \in \mathcal{P}_{[i..j]} \,\}, & \text{if } a = \perp \\ \{\, p \mid p \in \mathcal{P}_{[i..j]},\ p > 0,\ \text{and } S[p-1] = a \,\}, & \text{otherwise.} \end{cases}$$

The algorithm processes lcp-interval and computes the position sets in the bottom-up manner. Note that, if the function `process` (Algorithm 4.2.2) is called for an lcp-interval, then all its child intervals are available. On top of that, besides the child intervals, the stack is also maintaining the position sets which forms the fifth component to the quadruples. Suppose the algorithm is processing $[i..j]$.

If $[i..j]$ is a singleton interval, then let $p = \texttt{suftab}[i]$, $\mathcal{P}_{[i..j]} = \{\, p \,\}$ and

$$\mathcal{P}_{[i..j]}(a) = \begin{cases} \{p\}, & \text{if } p > 0 \text{ and } S[p-1] = a \text{ or } p = 0 \text{ and } a = \perp, \\ \emptyset & \text{otherwise.} \end{cases}$$

Now suppose $i < j$. For each character $a \in \Sigma \cup \{\perp\}$, $\mathcal{P}_{[i..j](a)}$ is computed step by step while processing the child intervals of $[i..j]$ in the left-to-right manner. By $\mathcal{P}^q_{[i..j]}(a)$ we denote the subset of $\mathcal{P}_{[i..j]}(a)$ obtained after processing the q-th child interval of $[i..j]$. Let $[l..r]$ be the $(q+1)$-th child interval of $[i..j]$. Due to the bottom-up manner, this interval has been processed and thus the $\mathcal{P}_{[l..r]}(b)$ are available for any character $b \in \Sigma \cup \{\perp\}$. Suppose the interval $[l..r]$ is being processed.

First, we compute the *cartesian product* by combining the set $\mathcal{P}^q_{[i..j]}(a),\ a \in \Sigma \cup \{\perp\}$, with the set $\mathcal{P}_{[l..r]}(b),\ b \in \Sigma \cup \{\perp\}$ which creates a maximal repeated pair $((p, p+\ell-1), (p', p'+\ell-1)),\ p < p'$ for all $p \in \mathcal{P}^q_{[i..j]}(a)$ and $p' \in \mathcal{P}_{[l..r]}(b)$, $a, b \in \Sigma \cup \{\perp\}$ and $a \neq b$. Because u is the longest common prefix of $[i..j]$, the pair is clearly a repeated pair. By construction, it combines only the positions which have different left character. This guarantees the left-maximality of the output. As the right maximality, this is guaranteed because the position sets

$\mathcal{P}^q_{[i..j]}(a)$ were inherited from child intervals of $[i..j]$ which are different from $[l..r]$. Hence the characters to the right of u are different.

Finally, we compute the *union* of $\mathcal{P}^{q+1}_{[i..j]}(c) = \mathcal{P}^q_{[i..j]}(c) \cup \mathcal{P}_{[l..r]}(c)$ so the interval $[i..j]$ can inherit the position sets from interval $[l..r]$.

These two steps together runs in $\mathcal{O}(|\Sigma|n + z)$. Each product of position sets is computed in a constant time making it $\mathcal{O}(z)$, where z is the number of repeats. As for the union operation of the position sets, we implement it with a linked list so it can be achieved in a constant time. That means, for each lcp-interval, we have $\mathcal{O}(|\Sigma|)$ union operations, and since there are $\mathcal{O}(n)$ intervals, it requires $\mathcal{O}(|\Sigma|n)$.

As far as the space consumption is concerned, if the child intervals of $[i..j]$ have been processed, its position sets are redundant. We store only the position sets of lcp-intervals that are maintained on the stack to be processed in the bottom-up traversal of the lcp-interval tree. Hence, the space required is bounded by the maximal size of the stack, so the total space requirements is $\mathcal{O}(|\Sigma|n)$. However, in practice, the stack size is much smaller [1].

Ziv-Lempel decomposition

Ziv-Lempel decomposition can be solved by the bottom-up traversal of the lcp-interval tree. We add another integer value *min* to the quadruples stored on the stack. Suppose the *process* function is being applied to ℓ-interval $[i..j]$. Let $[l_1..r_1]$, $[l_2..r_2]$,...,$[l_k..r_k]$ be the k child intervals of $[i..j]$ stored in the *childList*. Let min_1,...,min_k be the *min*-values of the child intervals. Let

$$M = \{\, min_1, ..., min_k \,\} \cup \{\, \texttt{suftab}[q] \mid q \in [i..j] \text{ and } q \notin [l_p..r_p] \text{ for all } 1 \leq p \leq k \,\}.$$

Set the min $\coloneqq \texttt{min}M$ and assign $s_q \coloneqq min$ and $l_q \coloneqq \ell$ to all $q \in M$ that $q \neq min$. As for the root interval $[0..n-1]$, we assign $s_q \coloneqq 0$ and $l_q \coloneqq 0$ to all $q \in M$.

4.5.2 Top-down traversals

In this section, we are concerned with the problems that are solved by the top-down traversal of the suffix tree. One of the problem is to answer queries of "Is P a substring of S?" in optimal $\mathcal{O}(m)$ time. As for the enumeration queries, the algorithm can answer those in optimal $\mathcal{O}(m + z)$.

Exact string matching

It is convenient for the following algorithm to be able to get the child interval $[l..r]$ of ℓ-interval $[i..j]$ whose suffixes have the character $a \in \Sigma$ at position ℓ. If the interval does not exist, it returns $\perp$. Let the function be called `getInterval`.

Algorithm 4.5.1: Exact string matching with ESA

```
1  c ← 0
2  queryFound ← True
3  (i, j) ← getInterval(0, n, P[c])
4  while (i, j) ≠ ⊥ and c < m and queryFound = True do
5      idx ← suftab[i]
6      if i ≠ j then
7          ℓ ← getLcp(i, j)
8          min ← min{ℓ, m}
9          queryFound ← S[idx + c..idx + min − 1] = P[c..min − 1]
10         c ← min
11         (i, j) ← getInterval(i, j, P[c])
12     else
13         queryFound ← S[idx + c..idx + m − 1] = P[c..m − 1]
14 if queryFound then Report(i, j)
15 else print("Pattern not found")
```

First, the Algorithm 4.5.1 finds the singleton or lcp-interval $[i..j]$ whose suffixes start with the character $P[0]$ on the line 3. Suppose the interval is singleton. Then the pattern P occurs in S if and only if $S[\texttt{suftab}[i]..\texttt{suftab}[i] + m - 1] = P$.

If $[i..j]$ is an lcp-interval, then we determine its lcp-value ℓ. Let $\omega = S[\texttt{suftab}[i]..\texttt{suftab}[i] + \ell - 1]$ be the longest common prefix of the suffixes $S_{\texttt{suftab}[k]}$ for all k in $i \leq k \leq j$. If $m \leq \ell$, then pattern P occurs in S if and only if $\omega[0..m-1]$ is equal to the pattern P. Otherwise, if $\ell < m$, then we check $\omega = P[0..\ell - 1]$. If the substrings are not equal, then the pattern P does not occur in S. If it is equal, then we search with $\texttt{getInterval}(i, j, P[\ell])$ for the ℓ'- or singleton interval $[i'..j']$ whose suffixes start with the prefix $P[0..\ell]$. Note that, the suffixes of $[i'..j']$ surely have $P[0..\ell - 1]$ as a common prefix because $[i'..j']$ is embedded in $[i..j]$. Then we continue in the same manner but with a slight change – we shift the matching by ℓ positions, which is represented by the variable c in the Algorithm 4.5.1.

As for the enumerative queries, all it takes is to list every position by enumerating the interval $[i..j]$ in which the pattern S occurs. The preceding algorithm takes $\mathcal{O}(m)$ time, and because P occurs z times in S, then listing every position of every occurence takes additional $\mathcal{O}(z)$ time.

Finding all shortest unique substrings

As for the shortest unique substrings problem, if u is a shortest unique substring, then there is an ℓ-interval $[i..j]$ and singleton child interval $[k..k]$ such

that u is a prefix of length $\ell+1$ of $S_{\texttt{suftab}[k]}$ and $u[\ell] \neq \texttt{|}$. We also maintain a set M of unique substrings, represented by their length and their start position in S. Additionally, the value `q` which is the length of the unique substrings in M detected so far. Initially, the `q` is set to ∞.

Algorithm 4.5.2: Find all the shortest unique substrings

```
   Output: the set M of shortest unique substrings
 1 queue.push(⟨0, n − 1⟩)
 2 while !queue.empty() do
 3 |  (i, j) ← queue.pop()
 4 |  ℓ ← getLcp(i, j)
 5 |  for every (l, r) of getChildIntervals(i, j) do
 6 |  |  if (l, r) is singleton then
 7 |  |  |  if ℓ + 1 < q then
 8 |  |  |  |  M ← {⟨ℓ + 1, suftab[l]⟩}
 9 |  |  |  |  q ← ℓ + 1
10 |  |  |  else if ℓ + 1 = q then  M ← M⊥{⟨ℓ + 1, suftab[l]⟩} ;
11 |  |  else if getLcp(l, r) +1 ≤ q then  queue.push(⟨l, r⟩) ;
```

Let $[i..j]$ be the current ℓ-interval being processed. We compute the child intervals with the function `getChildIntervals`. The child interval of $[i..j]$ can be either a singleton interval $[k..k]$ with $S_{\texttt{suftab}[k]}[\ell] \neq \texttt{|}$, or ℓ'-interval $[l..r]$. If the interval is an ℓ'-interval, then we add it to the back of the queue, provided the $\ell'+1 \leq \texttt{q}$. Now suppose the interval is singleton, then the prefix of $S_{\texttt{suftab}[k]}$ of length $\ell+1$ is a unique substring of S. If the length $\ell+1$ is less than `q`, then M is updated by $\{\langle \ell+1, \texttt{suftab}[k]\rangle\}$ and `q` is assigned with $\ell+1$. If M is not empty and `q` is equal to $\ell+1$, then we add $\langle \ell+1, \texttt{suftab}[k]\rangle$ to M. The algorithm continues to process the lcp-interval at the front of the queue, until the queue is not empty.

The time to process and verify the uniqueness of the substrings takes proportional time to the number of processed lcp-intervals. Thus the algorithm runs in $\mathcal{O}(n)$ time. However, in practice only a small number of lcp-intervals is processed, which can be seen in the Chapter *7 Experimental results*.

4.5.3 Traversal with suffix links

The last algorithm to be presented will solve the matching statistics by adapting the mechanism to the enhanced suffix array utilizing the suffix link table.

Computing matching statistics

The resulting algorithm is called *greedymatch* [1], which determines the matching statistics in $\mathcal{O}(n+m)$ time. Given the ESA for S, a *location in the ESA*

is a tuple ($[i..j]$, q, $[l..r]$) where $[i..j]$ is an ℓ-interval, and either $q = \ell$ and $[i..j] = [l..r]$ or the following condition holds: $[l..r]$ is a child interval of $[i..j]$ and either $[l..r]$ is an m-interval and $\ell < q < m$ or $[l..r]$ is a singleton interval and $\ell < q \leq n - \texttt{suftab}[l]$. Each location in the ESA corresponds to $S[\texttt{suftab}[l]..\texttt{suftab}[l] + q - 1]$.

To compute the $ms(k)$, `greedymatch` is applied to each suffix $T[k..m-1]$ and in each step it finds an ESA location ($[i..j]$, q, $[l..r]$) that corresponds to the longest prefix of $T[k..m-1]$ occuring as a substring of S. Then we assign $l_j \coloneqq q$ and $p_j \coloneqq \texttt{suftab}[z]$ for some $z \in [l..r]$, and set the `offset` $\coloneqq$ `q`-1. If the algorithm starts at the full length of the T, i.e. $j = 0$, or the length of longest matching prefix is zero, i.e. $l_j = 0$, then we start at location ($[0..n-1]$, q, $[0..n-1]$). Otherwise, we look up the suffix link interval $[i'..j']$ of $[i..j]$ in the $\texttt{suflink}[\texttt{min}\ell\mathit{Indices}(i,j)]$. If $q = \ell$ and $[i..j] = [l..r]$, then it matched the whole suffix and therefore the next suffix, i.e. $T[k+1..m-1]$ can be proceed with $[i'..j']$ with the length $q = \ell - 1$. Otherwise, the $S[\texttt{suftab}[l] + \ell..\texttt{suftab}[l] + q - 1]$ has to be rescanned from location $[i'..j']$.

It is very similar to the Algorithm 4.5.1 with a few modifications:

- it matches a character by character until no further matching is possible,
- it starts matching at any location and delivers a location as a result.

With these modification, *greedymatch* can be achieved in constant time per visited lcp-interval. Therefore, we obtain an algorithm that computes the matching statistics in $\mathcal{O}(n + m)$ time.

Chapter 5

Implementation

In this chapter, we are going to present the implementation details and choises made to construct the enhanced suffix array structure. All the structures were written in C++.

The implementation is dependent on the custom aliases for the data types. The most common which will be seen through the text is (1) `uval_t` – simply an `unsigned int`, (2) `uval_t2` – a pair of `uval_t`, (3) `uval_t3` – a tuple of `uval_t`, (4) `V_NUM` – a `std::vector<uval_t>`.

We consider in our implementation the alphabet of size ≤ 255. Hence every character takes only 1 byte.

5.1 Suffix Tree

For the construction of suffix tree, we used the Ukkonen's algorithm described in 3.3. Each node needs **at least** 20 bytes, consisting of `start`, `end`, `index` (for the leaves), the suffix link pointer, and additionally the pointers to the children. The whole suffix tree also keeps a whole string and a pointer to the root.

All the implementations concerning the suffix tree are located in the namespace `suftree`. As for the inner support functions, the anonymous namespaces approach for the access only within the file was chosen. The current available algorithms:

`suftree::maximalPairs` – computation of the maximal repeated pairs
`suftree::zivLempel` – computation of ziv-lempel decomposition
`suftree::matchingStatistics` – given T, compute the $ms(i)$ of $T[i..n]$
`suftree::getAllOccurrences` –find all occurrences of P in S
`suftree::isSubstring` – decide if pattern P is a substring of string S
`suftree::getUniqueSubstrings` – given S, output all the unique substrings

As for the construction of the suffix tree, simply call `suftree::SuffixTree tree(str)`, where *str* is an input string terminated with the sentinel character `|`.

5.2 Enhanced Suffix Array

We start by presenting the two approaches to construct the basic blocks `suftab` and `lcptab`.

5.2.1 Suffix Array

For the suffix array construction, we have chosen the space efficient linear suffix array construction by almost pure induced-sorting. It is easy to implement and it is considered as one of the fastest algorithm [23]. Also it is possible to incorporate the LCP array as a by-product of the SA-IS inducing algorithm, which is easier than inducing it by the *divSufSort* [24]. We start by giving some basic notions. The following presentation is based on [23, 30].

Definition 5.1 (L/S-type suffix)**.** Let S be a string terminated with a sentinel character. A suffix $S[i..n-1]$ is *S-type* if $S[i..n-1] < S[i+1..n-1]$. Otherwise, if $S[i..n-1] > S[i+1..n-1]$, then it is a *L-type*.

Lemma 5.1. *All the suffixes of S can be classified as S-type or L-type in $\mathcal{O}(n)$ time.*

Definition 5.2. A character $S[i]$ is called leftmost *S-type* (*LMS*) if $S[i]$, $0 < i < n$ is *S-type* and $S[i-1]$ is *L-type*. A suffix $S[i..n-1]$ is denoted as *LMS*-suffix.

Definition 5.3 (*LMS*-substring)**.** A *LMS*-substring is either a substring $S[i..j]$ with both $S[i]$ and $S[j]$ being *LMS* characters and there is no other *LMS* character in between or the sentinel itself.

In `suftab`, all suffixes starting with the same character span consecutively into an sub-array called c-bucket. Further, in the same bucket, the *L*-type suffixes precede the *S*-type suffixes (due to their definition). Hence, each bucket can be sub-divided into *S/L*-type buckets.

Now the induced sorting algorithm is described as follows. These steps are done in linear time.

1. Sort the *LMS*-suffixes and put them in their corresponding *S*-type buckets in `suftab` with their relative orders unchanged.

2. Induce the order of the *L*-type suffixes by scanning `suftab` in *left-to-right* manner. For every i in `suftab`, if $S_{\texttt{suftab}[i]-1}$ is *L*-type, then write $\texttt{suftab}[i]-1$ to the current *head* of the *L*-type c-bucket, where $c = S[\texttt{suftab}[i]-1]$, and forward the current head to the right by one.

3. Induced the order of the S-type suffixes by scanning `suftab` in *right-to-left* manner. For every i in `suftab`, if $S_{\texttt{suftab}[i]-1}$ is L-type, then write $\texttt{suftab}[i]-1$ ti the current *end* of the S-type c-bucket, where $c = S[\texttt{suftab}[i]-1]$, and forward the current head to the left by one.

The main idea is to treat the *LMS*-substrings as the basic blocks of the string and efficiently sort them, so we can replace the *LMS*-substrings with their order index, also called a *name*. As a result, the S can be represented by a shorter string, denoted by S'.

To determine the order of any two *LMS*-substrings, we compare the characters from left to right. First, by the lexicographical values, and if the characters are equal, then next we compare their types, where S-type has a higher priority than L-type.

The naming of *LMS*-substrings process is similar to the inducing the *LMS*-suffixes in the algorithm above, with the difference of putting in the *unsorted* *LMS*-suffixes into their corresponding buckets. Then we assign *names* to the *LMS*-substrings by comparing adjacent *LMS*-suffixes with the index of their bucket. So, the S' is created by joining the *names* of the LMS-substrings in their original positional order. This takes overall linear time.

We build a suffix array $\texttt{suftab}'$ of S' by applying the inducing algorithm *recursively* to S', if the size of buckets are less than the length of the S'. The important property [23] to observe is that the order of the suffixes in S' is the same as the order of the respective LMS-suffixes in S. Hence, $\texttt{suftab}'$ determines the sorting of the LMS-suffixes in S. Furthermore, at most every second suffix in S can be *LMS*, the complete algorithm has worst-case $T(n) = T(\frac{n}{2}) + \mathcal{O}(n) = \mathcal{O}(n)$. As for the space complexity, it is designated by the space needed to store the `suftab` for every reduced sub-problem. Initially, the first iteration is bounded by $n\lceil \log n \rceil$ bits, and decreases at least a half for every next iteration, hence the overall space complexity is $\mathcal{O}(n \log n)$.

5.2.2 LCP Array

For the LCP array, as suggested before, we have chosen the Fischer's [29] enhancement of *SA-IS* to additionally induce the LCP array as a by-product. It was shown that it outperforms most of the LCP array construction and in combination with the *SA-IS* it is very powerful.

Whenever we place two S- or L- suffixes S_{i-1} and S_{j-1} at $k-1$ and k in the final `suftab` (steps 3 and 4 in the SA-IS inducing algorithm), their lcp-value can be induced from the lcp-value of S_i and S_j. As these suffixes are the one that caused the inducing of S_{i-1} and S_{j-1}, their lcp-value ℓ is already known and thus we can set $\texttt{lcptab}[k] = \ell + 1$.

The augmented steps of the induced sorting algorithm:

1. Compute lcp-values of the *LMS*-suffixes and whenever we place an *LMS*-suffix, we also store its lcp-value at the corresponding position in `lcptab`.

2. Suppose that the inducing step just put suffix $S_{\texttt{suftab}[i]-1}$ into its L-type c-bucket at some position k. If it was the first suffix in its bucket, then its lcp-value is 0. Otherwise, suppose that in the previous iteration $j < i$ the inducing step placed suffix $S_{\texttt{suftab}[j]-1}$ at $k-1$ in the same c-bucket. If j and i are not in the same bucket, both the suffixes start with the different character, therefore the $\texttt{lcptab}[k]$ is set to 1 ($S_{\texttt{suftab}[j]-1}$ and $S_{\texttt{suftab}[i]-1}$ share only a common c at the beginning). Otherwise if j and i are in the same c'-bucket, the lcp-value of the suffixes $S_{\texttt{suftab}[j]-1}$ and $S_{\texttt{suftab}[i]-1}$ is given by the *minimum* in the range $[j+1..i]$ of $\texttt{lcptab}$. Let this value be ℓ, then the $\texttt{lcptab}[k] = \ell + 1$.

3. This step is very similar to the previous one. Suppose that in the previous iteration $j > i$, the inducing step put suffix $S_{\texttt{suftab}[j]-1}$ at $k+1$ in the same c-bucket. If k is the last position in its S-bucket, we skip the following step. Otherwise, if j and i are not in the same buckets, we set $\texttt{lcptab}[k+1]$ to 1. If j and i are in the same c'-bucket, then the lcp-value is given by the *minimum* in the range $[i+1..j]$. Hence we set $\texttt{lcptab}[k+1] = \ell + 1$

Note 4. When reaching the last L-suffix and the first S-suffix in the augmented step 3, it has to recompute the lcp-values between the first LMS-suffix in the c-bucket (if it exists) and the last L-suffix in the same bucket. Likewise for the step 4, when placing the first S-suffix in its c-bucket, it has to recompute lcp-values between this induced S-suffix and the largest L-suffix in the same c-bucket. However, this was shown in [29], that a *naive* computation of these cases is sufficient to achieve the linera running time.

When computing the $\texttt{suftab}'$ for S'. The lcp-values refers to the characters in the reduced alphabet which corresponds to *LMS*-substrings R_i in S. Hence we need to *scale* every lcp-value in $\texttt{lcptab}'$. Here with the knowledge of suffixes being lexicographically ordered in S', during the *scaling* $\texttt{lcptab}'[k]$, we know that the first $m = \texttt{min}(\texttt{lcptab}[k-1], \texttt{lcptab}[k])$ *LMS*-substrings match, hence we can compute the *real* lcp-value as

$$\sum_{i=0}^{\texttt{lcptab}[k]-1} |R_{\texttt{suftab}[k]+i}| = \underbrace{\sum_{i=0}^{m-1} |R_{\texttt{suftab}[k]+i}|}_{\text{already computed}} + \sum_{i=m}^{m-1} |R_{\texttt{suftab}[k]+i}|$$

To remove the computation of lcp-values recursively, we used the sparse variant of the Φ-algorithm [31] at the first level of the recursion to compute the lcp-values of the LMS-suffixes in overall linear time.

As for the finding minima in the augmented steps 3 and 4 above, we have chosen the following solution. We keep an array M of the size $|\Sigma|$, where $M[c]$ is the minimum of lcp-value in bucket c. To keep M up-to-date, after every step we set $M[c]$ to $\texttt{lcptab}[i]$, and then update all other elements in M that are larger than $\texttt{lcptab}[i]$ by $\texttt{lcptab}[i]$. This approach runs in $\mathcal{O}(n\sigma)$ time.

The construction algorithm is done only in the first level and operates just over the `lcptab` plus the *MinStack* which is bounded by the size of the alphabet.

The algorithms were chosen based on the Kurpicz's solution [32] and from the experiments [29], it yields the better overall run-time than most existing solutions.

Both the implementation of the SAIS and induced LCP based on this algorithm can be found in */esa/utils/sais{.hpp/.cpp}*.

Space reduction of LCP array

The `lcptab` requires $4n$ bytes in the worst case. However, in practive there usually only few entries that are larger than or equal to 255 and there is a trick how to further optimalize it described from [1]. With this, the `lcptab` can be implemented in little more than n bytes. For the lcp-values ≥ 255, we store them in the special table called `llvtab`. This table consists of pairs $(i, \texttt{lcptab}[i])$ ordered by the i. If $\texttt{lcptab}[i] = 255$, then we find the correct value in the table `llvtab` by performing a binary search using i as the key. Thus, $\texttt{lcptab}[i]$ is achieved in $\mathcal{O}(\log_2 |\texttt{llvtab}|)$ time.

5.2.3 Space reduction of child-table

From the look at the Table 4.1, there is a lot of unused space in the child-table. It is possible to reduce the space requirements just to a single field [1]. The *down* field is necessary only if it does not have the same information as *up* field.

An ℓ-interval $[i..j]$ with k ℓ-indices has at most $k+1$ child intervals. Consider a new space-reduced array $\texttt{childtab}'$. Suppose $[l_1..r_1], ..., [l_{k+1}..r_{k+1}]$ are the child intervals of $[i..j]$ and let i_q be the first ℓ_q-index of the interval ℓ_q-$[l_q..r_q]$ for $1 \leq q \leq k+1$. We store indices $i_1, ..., i_k$ in the $\texttt{childtab}'[r_1 + 1]$,...,$\texttt{childtab}'[r_k + 1]$. The remaining index i_{k+1} is stored in the `down` field of $\texttt{childtab}[r_k + 1]$. This index can be stored in the $\texttt{childtab}'[r_k + 1]$ because $r_k + 1$ is the last ℓ-index, therefore it is guaranteed the field is empty.

However, this raise a question whether it is possible to decide if $\texttt{childtab}'[i]$ contains the next ℓ-index or the *down* value of the $\texttt{childtab}[i]$.

1. If $\texttt{lcptab}[\texttt{childtab}'[i]] = \texttt{lcptab}[i]$, then it contains the *nextℓIndex*
2. If $\texttt{lcptab}[\texttt{childtab}'[i]] > \texttt{lcptab}[i]$, then it contains the *down* value.

As for the *up* field, it is stored in the unused space of $\texttt{childtab}'$. Note that $\texttt{childtab}[i+1].\texttt{up} \neq \perp$ iff $\texttt{lcptab}[i] > \texttt{lcptab}[i+1]$. In this case, it is guaranteed $\texttt{childtab}[i].\texttt{next}\ell\texttt{Index}$ is empty and therefore $\texttt{childtab}'[i]$ can store the value $\texttt{childtab}[i+1].\texttt{up}$. To check whether $\texttt{childtab}'[i]$ contains the value $\texttt{childtab}[i+1].\texttt{up}$, it is sufficient to only test $\texttt{lcptab}[i] > \texttt{lcptab}[i+$

1]. In conclusion, although the child-table theoretically needs three fields, in practice only one field is required to store the child-table.

5.2.4 RMQ for the suffix link table

The *RMQ* problem is heavily studied and nowadays, it is still being explored. Gabow et al. [33] presented an algorithm that reduces the *RMQ* problem to the *least common ancestor* problem by transforming the array into a *Cartesian* tree. This however has a major drawbacks: 1) it uses too much space, 2) it relies on the structures such as tree, which is similar to the suffix tree/suffix array duality.

In this section, we are using the notation from [34] to describe the RMQ solution. We denote an algorithm which preprocess the input in $p(n)$ time and handles queries in $q(n)$ as $\langle p(n), q(n)\rangle$. For example, the naive method with the notation above would be described as $\langle \mathcal{O}(1), \mathcal{O}(n)\rangle$, as it doesn't need preprocessing and it searches the array from i to j.

The recent most notable approach is an improved $\langle \mathcal{O}(n), \mathcal{O}(1)\rangle$ by Fischer and Heun[35], which doesn't need the Cartesian tree construction nor any other dynamic data structures, thus making it an optimal algorithm. However, it was shown that in practice the simple $\langle \mathcal{O}(n), \mathcal{O}(\log n)\rangle$ outperforms the $\langle \mathcal{O}(n), \mathcal{O}(1)\rangle$ solutions (refer to the Fig. 4 and Fig. 5 in [35]).

Therefore for the ESA implementation, we have chosen the $\langle \mathcal{O}(n), \mathcal{O}(\log n)\rangle$ hybrid RMQ structure, which uses two layered approach [36]. The bottom layer requires no preprocessing and consist of the original array. As for the top layer, it is divided into equal sized blocks and the minimum over each block is computed. Hence, a minimum over a range is the minimum of each block fully contained in the range and the indices on both ends of the range, in partially contained blocks.

Suppose we use a $\langle p_1(n), q_1(n)\rangle$ for the block minima and $\langle p_2(n), q_2(n)\rangle$ within each block, with block size b. Total preprocessing time would be $\mathcal{O}(n + p_1(\frac{n}{b}) + (\frac{n}{b})p_2(b))$ and the query time $\mathcal{O}(q_1(\frac{n}{b}) + q_2(b))$. The sparse table has complexity $p_1 = \mathcal{O}(n \log n)$, $q_1 = \mathcal{O}(1)$ and the bottom layer is simply a naive one. If we set the $b = \log n$, the construction time would be:

$$
\begin{aligned}
\mathcal{O}\Big(n + p_1\Big(\frac{n}{b}\Big) + \Big(\frac{n}{b}\Big)p_2(b)\Big) &= \mathcal{O}\Big(n + \frac{n}{b}\log\Big(\frac{n}{b}\Big) + \Big(\frac{n}{b}\Big)\Big) \\
&= \mathcal{O}\Big(n + \frac{n}{\log n}\log\Big(\frac{n}{\log n}\Big) + \Big(\frac{n}{\log n}\Big)\Big) \\
&= \mathcal{O}\Big(n + n + \Big(\frac{n}{\log n}\Big)\Big) \\
&= \mathcal{O}(n)
\end{aligned}
$$

As for the query time:

$$\begin{aligned}\mathcal{O}\Big(q_1\Big(\frac{n}{b}\Big) + q_2(b)\Big) &= \mathcal{O}(1+b)\\ &= \mathcal{O}(1+\log n)\\ &= \mathcal{O}(\log n)\end{aligned}$$

Thus, by setting the $b = \Theta(\log n)$, we can achieve the RMQ with complexity $\langle \mathcal{O}(n), \mathcal{O}(\log n)\rangle$.

The implementation is located in the file */esa/utils/rmq{.hpp/.cpp}*

5.2.5 Space complexity

In this section, we will have a look at the space complexity for the ESA structure. In the fullest form the ESA contains the input string, `bwttab`, `suftab`, `lcptab`, `childtab` and `suflink`. Depending on the application, we can reduce the other unused table. For example, for the maximal repeated pairs problem, we don't need the input string however we do need the knowledge of the left character of every entry of the `suftab`, so here it can be replaced by the more succint representation, the `bwttab`. However, in case we do keep the input string, then `bwttab` can just be induced from the input string and hence we save n bytes.

suftab each entry takes 4 bytes, thus overall $4n$ bytes.

lcptab with the reduced version, this takes a little more than n bytes.

childtab each entry takes 4 bytes (reduced version), so in total $4n$ bytes.

suflink each entry of suflink table takes 8 bytes, thus in total $8n$ bytes.

The total bytes required to store all the structures of the ESA is $18n$ bytes (if we count $1n$ just for input string/`bwttab`). Thus, this implementation still requires less space than even the recently improved implementations of suffix tree, which requires 20 bytes per input character in the worst case [16].

5.2.6 Usage

In this last section, we're going to present the usage of the implemented solutions to each chosen problem. Every algorithm concerning the ESA is located in the namespace `esa` except for the construction itself.

To construct the ESA structure, call `ESA esa`(*str*), where *str* is an input string terminated with the sentinel character `|`. If the string contains the sentinel character elsewhere, it throws an exception. The construction of the ESA is implemented to create all enhancing data structures.

Bottom-up traversal

The bottom-up traversal of every presented algorithm is very similar and as described in the 4.2.2, it is sufficient to just specify the `process` function. Hence, we've chosen the *policy-based design* which is an idiom for a class template (*host class*) taking type parameters as input, each implementing a particular interface called a *policy*. In our case, the *host class* is `LcpInterval<Process>`, where `Process` is a *policy*. The host class can be seen on the Code 1.

```
template <typename Process = Algorithm>
class LcpInterval: private Process {
public:
  typedef typename Process::node_t node;
  /** Constructor accepting the ESA */
  LcpInterval(const ESA & esa): Process(esa) { }
  /** A traverse method which simulates the bottom-up manner.
   * Returns the output of the Process algorithm */
  auto traverse();
};
```

Listing 1: The `LcpInterval` host class

The `Process` policy should be an implementation of the interface `Algorithm`, which defines the behaviour of the process function. The interface contains the reference to the ESA and the basic block (node) of the stack without the child information. Each child class of `Algorithm` will specify their node, if needed.

```
class Algorithm{
public:
  struct node_t;
  /** Constructor accepting the ESA to work with */
  Algorithm(const ESA & esa);
  /** Process a node in a bottom-up manner. Processes a node */
  void process(node_t & node);
  /** The output of the algorithm */
  auto output();
};
```

Listing 2: The `Algorithm` interface of the policy `Process`

Maximal pairs

With the mentioned implementation of the `LcpInterval` above, the problem maximal repeated pairs has the implementation of the `Algorithm` interface in the *esa/algorithms/bottom/maximal_pairs.hpp*.

```
ESA esa("acaaacatat|");
esa::LcpInterval<esa::MaximalPairs> lcpInterval(esa);
std::list<uval_t3> = lcpInterval.traverse();
```

Ziv-Lempel decomposition

Ziv-Lempel decomposition is very similar except the inner element of the stack is slight different (described in the 4.5.1). It returns the pointer to the l and s arrays.

```
uval_t *l, *s;
ESA esa("acaaacatat|");
esa::LcpInterval<esa::ZivLempel> lcpInterval(esa);
std::tie(s, l) = lcpInterval.traverse();
```

Top-down traversal

Top-down traversal is relying on the `childtab`, for which there are three essential functions: `ESA::getFirstLIndex` and `ESA::getNextLIndex`. For the child interval of ℓ-interval $[i..j]$, one can use `ESA::getChildInterval(uval_t i, uval_t j)`. The supporting functions are `ESA::getLcp(uval_t i, uval_t j)`, which returns the lcp-value of the interval $[i..j]$. The last supporting function is to get the child m-interval of ℓ-interval which has character `a` on ℓ position `ESA::getInterval(uval_t i, uval_t j, char a)`.

```
ESA esa("acaaacatat|");
// decision query "Is pattern P a substring of S?"
std::cout << "The string 'aaa' "
    << (esa::isSubstring(esa, "aaa") ? "IS" : "IS NOT")
    << " a substring of string acaaacatat|" << std::endl;
// enumerative query "Where are all occurrences of P in S?"
V_NUM esa_res = esa::getAllOccurences(esa, "aa");
// Get all the shortest unique substrings
std::set<uval_t2> esa_res = esa::getUniqueSubstrings(esa);
```

Traversal with suffix link table

The last usage of the ESA is the computation of *matching statistics*. If the algorithm returns some $\ell-interval$ and the `len` is greater than zero, it returns immediately i as the occurrence.

```
ESA esa("cacaccc|");
std::vector<uval_t2> res =
    esa::greedyMatch(esa, "caacacacca");
```

ESA structure

We present the proposed ESA structure. The documentation comments and variables were removed for the illustration.

```
class ESA{
public:
  ESA(const STRING & str);
  ESA(STRING && str);
  ~ESA();

  const STRING & str() const;
  const uval_t * sa() const;
  const lcptab_t & lcp() const;
  const char * bwt() const;
  const uval_t * childtab() const;
  const interval_t * suflink() const;
  uval_t n() const;

  uval_t2 getInterval(uval_t i, uval_t j, char a) const;
  std::vector<uval_t2> getChildIntervals(uval_t i, uval_t j) const;
  uval_t getLcp(uval_t i, uval_t j) const;
  uval_t getFirstLIndex(uval_t i, uval_t j) const;
  uval_t getNextLIndex(uval_t i) const;
private:
  void constructSALCP();
  void constructBWT();
  void constructChildTab();
  void constructSufLink();
  void setBoundaries(std::queue<uval_t2> &, interval_t *) const;
};
```

Listing 3: The ESA structure for the illustration

Chapter 6

Testing

This chapter will be concerned about testing. The purpose of this is to ensure that the application works correctly with the least errors. We have implemented the set of unit tests with the test library Catch 2 [37].

6.1 Catch2

The main key features are that it's just a header-file, therefore all it is needed is just to include the header file, and it can be used immediately without any external dependencies.

Test cases are written as a functions, which are self-registering, so all it is needed is really just to define how the test case will look like. Each test case can be divided into sections, hence it can be run in a complete isolation without any fixtures. This is especially useful, when the one setup across the multiple methods is needed but sometimes a setup is needed to be slightly changed.

An example of test case taken from [37]:

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
unsigned int Factorial( unsigned int number ) {
  return number > 1 ? Factorial(number-1)*number : 1;
}

TEST_CASE( "Factorials are computed", "[factorial]" ) {
  REQUIRE( Factorial(0) == 1 );
  REQUIRE( Factorial(3) == 6 );
  REQUIRE( Factorial(10) == 3628800 );
}
```

Let's break the example to explain how the test is defined. The test case is registered with the macro `TEST_CASE` which takes two arguments – unique test name in the string and optionally a tag. With the name and the tag, one can run seperately a group of tests either by a tag or by specifying a wildcarded test name. The `REQUIRE` macro is used for the test assertions. Rather than a seperate macro for every type, this is expressed using the classic C/C++ syntax.

When seperating the tests into multiple files, there should be a main one which has the `#include` line along with the `#define CATCH_CONFIG_MAIN`. Otherwise it is just sufficient to include the header.

6.2 Unit testing

For the unit testing, we have implemented 19 test cases, which test most of the available functionalities of the application (14203 assertions when running the tests). Most of the functionality is initially tested with the strings for which we know the values and can be single-handely computed. Afterwards, it is tested with the *Catch*'s `GENERATE` macro which generates the listed values and can be combined with another `GENERATE` to create multiple combinations. For this we also implemented the naive algorithms to test against. An example of testing the correctness of a suffix array:

```
TEST_CASE("Suffix array of a random string", "[sa]") {
  uval_t j = GENERATE(100, 500, 1000), // string len
         k = GENERATE(0, 5, 3, 2, 1); // 0 => full alphabet
  for(uval_t i = 0; i < 5; ++ i) {
    // generates the string with len j and alphabet k
    STRING str = stringGenerator(j, k);
    ESA esa(str);
    uval_t n = str.size();
    V_NUM sa(n), naive_sa(str);
    for(uval_t i = 0; i < n; ++ i) sa[i] = esa.sa()[i];
    REQUIRE(esa.n() == naive.size());
    REQUIRE_THAT(sa, !Catch::VectorContains(n));
    REQUIRE_THAT(sa, Catch::Equals(naive));
  }
}
```

Thankfully, the tests revealed some bugs in the implementation, mostly around the suffix tree algorithms, which were immediately fixed afterwards. All the tests can be run in the command line with the command `make test`. The set of the test cases can be found in the folder `/tests/` and the name of the file suggests what functionality it tests.

Chapter 7

Experimental results

This chapter focuses on the performance of both implemented solutions to the chosen problems. The algorithms solved by ESA are prefixed with `esa` and algorithms by suffix tree are prefixed `suftree`. First, we present the setup environment for our experiments and afterwards the chosen data sets and their characteristics.

7.1 Setup and Environment

All the functionalities were compiled with `g++ -std=c++17 -ffast-math -g -O3 -Wextra` and the measurements were done on Intel Core i3-9100F CPU @ 3.60Ghz, 16.0 GB RAM running in Ubuntu environment on Windows 10.

The experiments were conducted with the help of Benchmark library developed by Google [38]. The library was chosen because it has a great scale of customizing the settings of the benchmark. The most notable is the function `DoNotOptimize` which helps to conduct the experiment without the optimalizations, i.e. without being dependent on the previous iterations. Every experiment were conducted in 5 iterations.

7.2 Data sets

We chose the data sets from multiple sources, Manzini-Ferragina corpora[1], SMART-tool data source[2], Pizza&Chili Corpus[3] and lastly the online database NCBI[4] for the genome assembly. The whole chosen data set can be seen on the Table 7.1. Most of the data are biology-related, but we also included the italian and english texts.

[1] `http://people.unipmn.it/manzini/lightweight/corpus/`
[2] `http://www.dmi.unict.it/ faro/smart/corpus.php`
[3] `http://pizzachili.dcc.uchile.cl/`
[4] `https://www.ncbi.nlm.nih.gov`

Name	Avg. LCP	Max. LCP	File size (in bytes)	Description
ipromes	8.01	61	1,301,484	A snippet of italian text *I promessi sposi*
strep1	25.59	5,885	2,038,615	Genome of *Streptococcus pneumoniae R6*
strep2	35.57	5,985	2,160,842	Genome of *Streptococcus pneumoniae TIGR4*
world192	23.01	559	2,473,400	The CIA world fact book
hs	7.96	3,207	3,295,751	Homo sapiens protein
bible	13.97	551	4,047,392	The King James version of the bible
ecoli1	17.38	2,815	4,638,690	Genome of *E. Coli* bacterium
shaks	15.55	593	5,458,199	Collection of works of William Shakespeare
ecoli2	34.25	5,216	5,594,605	Genome of *E. Coli O157:H7* bacterium
yeast1	42.95	8,375	12,157,105	Genome of *S. cerevisiae* bacterium
yeast2	40.83	10,147	12,747,577	Genome of *S. pombe* bacterium
chr22	1,979.25	199,999	34,553,758	Assembly of human chromosome 22
chr21	23.58	73,034	40,088,619	Assembly of human chromosome 21
protein50	166.19	25,822	52,241,887	Protein sequences from *Swissprot database*

Table 7.1: Data sets used for experiments sorted by the file size

7.3 Construction

In this section we are concerned with the running time and space consumption of the construction of the suffix tree and the ESA. For the construction, we have chosen the files with different size and characteristics. The space requirements were measured with the *resident set size*, which is the memory portion occupied by a process in main memory. The value is accessible from the `/proc/self/statm`. Table 7.2 shows the results. It is clear that ESA structure is superior to the suffix tree in both the running time and especially the space comsumption, where ESA uses almost 10 times less space.

	Running time		Space consumption	
Name	ESA	Suffix Tree	ESA	Suffix Tree
ipromes	**0.62**	1.11	**41,012**	313,832
bible	**2.32**	3.42	**128,284**	986,572
yeast1	**9.64**	13.71	**307,928**	3,134,880
chr22	**29.31**	40.35	**859,476**	9,068,832
protein50	**69.19**	72.08	**1,482,524**	9,598,800

Table 7.2: Running time (in seconds) and space requirements (in kB).

We detected that the most time consuming phase of construction of ESA is the suffix array construction. We compared our implemented SAIS with the very careful optimized implementation of Yuta Mori [39] and tested on three biggest files from our data set. From the results, the Yuta Mori's outperforms ours by at least twice (depending on the file size) the running time. (1) *chr22* – **10.94s** vs. 20.31s, (2) *chr21* – **13.44s** vs. 24.80s, and (3) *protein50* – **19.75s** vs. 45.36s.

7.4 Performance of Algorithms

In this section, we present the experiments done on the individual algorithms and compare the running time of the algorithms using the suffix tree and the ESA. To reflect the running time of each algorithms, the construction time is excluded from the measurements. For every algorithm, we used only a subset of the chosen data set.

7.4.1 Computing maximal repeated pairs

For the computation of maximal repeated pairs we divided the running time for different ℓ, which indicates the search for the repeats of length $\geq \ell$. For the experiment, we have chosen the files *yeast1*, *protein50*, *shaks* and *world192*, each of them possesses different size of alphabet and length. Table 7.3 reveals that ESA is generally faster than suffix tree.

	protein50			*yeast1*			*shaks*			*world192*		
ℓ	#reps	esa	st	#reps	esa	st	#reps	esa	st	#reps	esa	st
18	76,792,738	**23.61**	157.48	306,646	**3.50**	4.15	162,368,777	**8.05**	9.89	5,021,355	**1.40**	1.44
20	65,955,648	**22.91**	119.44	175,562	**3.44**	4.19	125,172,713	**6.33**	8.18	3,534,608	**1.20**	1.38
23	52,031,187	**21.88**	57.67	84,174	**3.24**	4.13	77,042,967	**4.18**	5.06	2,350,986	**1.02**	1.33
25	41,020,350	**21.27**	53.78	56,685	**3.29**	4.26	55,730,383	**3.23**	4.25	1,828,920	**0.93**	1.31
30	23,545,736	**19.99**	31.52	32,202	**3.23**	4.13	24,739,511	**1.91**	2.73	935,168	**0.82**	1.29
40	11,052,382	**18.14**	22.14	20,768	**3.20**	4.13	5,866,133	**1.02**	1.92	491,989	**0.73**	1.28
60	3,690,804	**17.17**	21.55	13,907	**3.17**	4.12	32,705	**0.71**	1.70	108,758	**0.67**	1.26

Table 7.3: Measurement of the maximal repeated pairs computation. The running time is in seconds, as for the columns, *esa* represents the ESA method and *st* is the Suffix Tree. #reps gives the number of repeats of length $\geq \ell$.

7.4.2 Ziv-Lempel decomposition

We implemented the algorithms described in Sections 3.4 and 4.5. The algorithm based on the ESA is far superior because it uses only the properties of the lcp-tree intervals with the SA and in the combination it reaches overall a better running time. For the experiment, we used the texts *bible*, *world192*, *shaks* and the italian text *ipromes*.

file	esa	suffix tree
bible	**0.57**	4.99
world192	**0.25**	2.54
ipromes	**0.12**	1.32
shaks	**0.12**	1.32

Table 7.4: Measurement of Ziv-Lempel decomposition. The running time is in seconds.

7.4.3 Pattern searching

For the third experiment, we have chosen files *ecoli1*, *yeast1*, *bible* and *shaks* to measure the running time of searching for the patterns. We use this strategy: for every iteration we generated one million substrings of the input string. The length of the substrings are categorized into intervals on which we conducted the experiment. To simulate the pattern not occurring in the string, we simply reversed it for every even pattern. The lengths of the substrings are evenly distributed over the intervals $\mathcal{I} = \{ (20, 30), (30, 40), (40, 50) \}$.

$\mathcal{I}$	*(20,30)*		*(30, 40)*		*(40, 50)*	
File	esa	st	esa	st	esa	st
ecoli1	**1.09**	1.80	**1.10**	1.67	**1.08**	1.65
yeast1	**1.37**	2.08	**1.38**	2.06	**1.36**	2.74
bible	1.74	**1.39**	1.73	**1.36**	1.73	**1.35**
shaks	**2.43**	9.73	**2.25**	3.25	2.16	**1.84**

Table 7.5: Measurement of the pattern searching. The running time is in seconds.

The experiment revealed that for the small alphabet, the ESA is faster. However when the alphabet grows the ESA is slightly slower than Suffix Tree. This slowdown is most probably caused by the searching of the interval that contains the character a at position ℓ. However, it is still clear, that it can be compete with other methods.

7.4.4 Shortest unique substrings

The shortest unique substrings is sensible for the genomes, therefore we conducted an experiment on the files: *hs*, *ecoli1*, *yeast1*, *chr21* and *protein50*. Clearly, the ESA is faster than the Suffix tree method. The results can be seen in the Table 7.7. We also measured the number of processed lcp-intervals and the percentage of it to display the ratio of processed and total amount of lcp-intervals in practice.

File	esa	suffix tree	cnt	len	processed	total	percentage
hs	**0.00413**	0.309	1,657	9	7,627	1,251,682	0.60%
ecoli1	**0.00228**	0.560	3	7	11,392	2,978,796	0.38%
yeast1	**0.0167**	1.571	383	9	92,863	7,905,335	1.17%
ch21	**0.0263**	5.609	1468	9	93,114	27,720,199	0.33%
protein50	**8.4e-5**	0.00663	25	2	51	27,439,445	0.0001%

Table 7.6: Measurement of the pattern searching. The running time is in seconds. The column *cnt* indicates the number of shortest unique substrings, *len* indicates the length of the shortest substrings, *processed* indicates the number of processed lcp-intervals and the *total* is the number of lcp-intervals in lcp-intrerval tree.

7.4.5 Computing matching statistics

The last experiment we conducted is computing the matching statistics. For this measurement, we used four pairs of similar genomes: *yeast1* with *yeast2* as *yeast*, *strep1* with *strep2* as *strep*, *chr21* and *chr22* as *chr*, and *ecoli1*, *ecoli2* as *ecoli*. We build the ESA and suffix tree for the smaller input file and proceed the matching statistics against the bigger one.

Genome pair	Total length	esa	suffix tree
strep	4,199,457	0.589	**0.561**
ecoli	10,233,295	0.268	**0.129**
yeast	24,904,682	**5.18**	5.73
chr	74,642,377	16.682	**16.632**

Table 7.7: Measurement of the matching statistics computation. The running time is in seconds.

In most cases the experiment shows that the suffix tree is slightly faster than the ESA. This is probably due to the *rescanning* step and querying for the suffix link interval. However, from the results, the ESA method is still very competitive with the suffix tree method.

Conclusion

In this chapter, we summarize the thesis and conclude the results. We also present some topics for the future improvements.

Evaluation of the thesis

First, the definitions and construction of the enhanced suffix array and suffix tree had to be studied. This was fulfilled in the chapters 3 and 4.

Second goal was to propose a data structure of the enhanced suffix array, which is the main contribution of the thesis and chapter 5 is dedicated to this goal. Furthermore, we studied the multiple implementation of the suffix array construction, lcp-array construction and RMQ algorithms to create an optimal solution.

Along with the data structure, the algorithms to the chosen problems which uses at least three different suffix tree traversals were implemented in C++ using the C++17 standard. These problems were also implemented with the ESA to simulate the suffix tree traversal.

In the chapter 6 with the usage of *Catch2* library, we properly test the functionalities of the enhanced suffix array and the algorithms with the unit tests on fixed inputs and the randomized inputs.

Finally, the chapter 7 covers the conducted experiments to compare and show the effectiveness of the implemented algorithms.

Future Work

As mentioned earlier, the *DivSufSort*, which is widely agreed to be the fastest known algorithm for the suffix array construction, could be thoroughly studied and replace the current implementation of the SAIS. In 2017, Fischer [29] also presented the enhancement for the algorithm with lcp construction and the concise description to understand the algorithm. The experiments revealed

it to be competitive with existing implementations and in some cases even faster.

As the RMQ is heavily studied, the enhanced suffix array structure could be further enhanced by a better RMQ solution. The succinct one can be used to replace and further reduce the space consumption, while retaining the capability to simulate the traversals [40].

Contribution

The thesis is an implementation type, therefore the main contribution of the thesis is the proposed enhanced suffix array, which eventually can be imported to the ALIB[5] framework.

[5]The Algorithm Library Toolkit – https://gitlab.fit.cvut.cz/algorithms-library-toolkit

Bibliography

[1] Abouelhoda, M. I.; Kurtz, S.; et al. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2004: pp. 53–86, ISSN 1570-8667. Available from: `http://www.sciencedirect.com/science/article/pii/S1570866703000650`

[2] Manber, U.; Myers, G. Suffix Arrays: A New Method for On-Line String Searches. *SIAM Journal on Computing*, volume 22, no. 5, 1993: pp. 935–948, doi:10.1137/0222058. Available from: `https://doi.org/10.1137/0222058`

[3] Gusfield, D. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. USA: Cambridge University Press, 1997, ISBN 0521585198.

[4] Kärkkäinen, J.; Sanders, P.; et al. Simple Linear Work Suffix Array Construction. 06 2003, doi:10.1007/3-540-45061-0_73.

[5] Ko, P.; Aluru, S. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, volume 3, no. 2, 2005: pp. 143 – 156, ISSN 1570-8667, doi:https://doi.org/10.1016/j.jda.2004.08.002, combinatorial Pattern Matching (CPM) Special Issue. Available from: `http://www.sciencedirect.com/science/article/pii/S1570866704000498`

[6] Nong, G.; Zhang, S.; et al. Linear Suffix Array Construction by Almost Pure Induced-Sorting. *Proceedings of the Data Compression Conference*, 03 2009: pp. 193–202, doi:10.1109/DCC.2009.42.

[7] Crochemore, M.; Hancart, C.; et al. *Algorithms on Strings*. Cambridge University Press, 2007, doi:10.1017/CBO9780511546853.

[8] Holub, J. *Introduction, basic notions and border array*. 2019, [cit. 2020-02-17].

[9] Mareš, M.; Valla, T. *Průvodce labyrintem algoritmů*. CZ.NIC, z.s.p.o., 2017, ISBN 9788088168195.

[10] Knuth, D. E.; Morris, J. H.; et al. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, volume 6, no. 2, March 1977: pp. 323–350, ISSN 0097-5397.

[11] Boyer, R. S.; Moore, J. S. A Fast String Searching Algorithm. *Commun. ACM*, volume 20, no. 10, Oct. 1977: p. 762–772, ISSN 0001-0782, doi:10.1145/359842.359859. Available from: `https://doi.org/10.1145/359842.359859`

[12] Crochemore, M.; Hancart, C.; et al. *Algorithms on Strings*. USA: Cambridge University Press, 2014, ISBN 1107670993.

[13] Weiner, P. Linear Pattern Matching Algorithm. 11 1973, pp. 1–11, doi:10.1109/SWAT.1973.13.

[14] McCreight, E. M. A Space-Economical Suffix Tree Construction Algorithm. *J. ACM*, volume 23, no. 2, 1976: p. 262–272, ISSN 0004-5411, doi:10.1145/321941.321946. Available from: `https://doi.org/10.1145/321941.321946`

[15] Ukkonen, E. On-Line Construction of Suffix Trees. *Algorithmica*, volume 14, no. 3, 1995: p. 249–260, doi:10.1007/BF01206331. Available from: `https://doi.org/10.1007/BF01206331`

[16] Kurtz, S. Reducing the Space Requirement of Suffix Trees. *Software – Practice and Experience*, volume 29, 1999: pp. 1149–1171.

[17] COMPSCI 260, C. S. D. U. *Suffix Trees and its Construction [online]*. [cit. 2020-03-08]. Available from: `https://www2.cs.duke.edu/courses/fall14/compsci260/resources/suffix.trees.in.detail.pdf`

[18] Lander, E.; Chen, C.; et al. Initial Sequencing and Analysis of the Human Genome. *Nature*, volume 409, 02 2001.

[19] Ziv, J.; Lempel, A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, volume 23, no. 3, May 1977: pp. 337–343, ISSN 1557-9654, doi:10.1109/TIT.1977.1055714.

[20] Ziv, J.; Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, volume 24, no. 5, Sep. 1978: pp. 530–536, ISSN 1557-9654, doi:10.1109/TIT.1978.1055934.

[21] Aho, A.; Corasick, M. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, volume 18, 06 1975: pp. 333–340, doi:10.1145/360825.360855.

[22] Chang, W. I.; Lawler, E. L. Approximate string matching in sublinear expected time. *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, 1990: pp. 116–124 vol.1.

[23] Nong, G.; Zhang, S.; et al. Linear Suffix Array Construction by Almost Pure Induced-Sorting. *Proceedings of the Data Compression Conference*, 03 2009: pp. 193–202, doi:10.1109/DCC.2009.42.

[24] Fischer, J.; Kurpicz, F. Dismantling DivSufSort. *CoRR*, volume abs/1710.01896, 2017, 1710.01896. Available from: `http://arxiv.org/abs/1710.01896`

[25] Burrows, M.; Wheeler, D. J. A block-sorting lossless data compression algorithm. Technical report, 1994.

[26] Kasai, T.; Lee, G.; et al. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. 06 2001, pp. 181–192, doi:10.1007/3-540-48194-X_17.

[27] Manzini, G. Two Space Saving Tricks for Linear Time LCP Array Computation. 07 2004, pp. 372–383, doi:10.1007/978-3-540-27810-8_32.

[28] Kärkkäinen, J.; Manzini, G.; et al. Permuted Longest-Common-Prefix Array. 06 2009, pp. 181–192, doi:10.1007/978-3-642-02441-2_17.

[29] Fischer, J. Inducing the LCP-Array. *CoRR*, volume abs/1101.3448, 2011, 1101.3448. Available from: `http://arxiv.org/abs/1101.3448`

[30] Okanohara, D.; Sadakane, K. A Linear-Time Burrows-Wheeler Transform Using Induced Sorting. 08 2009, pp. 90–101, doi:10.1007/978-3-642-03784-9_9.

[31] Kärkkäinen, J.; Manzini, G.; et al. Permuted Longest-Common-Prefix Array. 06 2009, pp. 181–192, doi:10.1007/978-3-642-02441-2_17.

[32] Kurpicz, F. sais-lite-lcp. `https://github.com/kurpicz/sais-lite-lcp`, 2015, [Cited 2020-04-01].

[33] Gabow, H. N.; Bentley, J. L.; et al. Scaling and related techniques for geometry problems. In *STOC '84*, 1984.

[34] Bender, M.; Farach-Colton, M.; et al. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, volume 57, 11 2005: pp. 75–94, doi:10.1016/j.jalgor.2005.08.001.

[35] Fischer, J.; Heun, V. Theoretical and Practical Improvements on the RMQ-Problem with Applications to LCA and LCE. In *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM'06)*,

Lecture Notes in Computer Science, volume 4009, edited by M. Lewenstein; G. Valiente, Barcelona, Spain, July 5-7, 2006: Springer-Verlag, 2006, pp. 36–48, doi:10.1007/11780441_5.

[36] Schwarz, K.; Leon, A.; et al. *Range Minimum Queries [online]*. [cit. 2020-04-02]. Available from: `https://web.stanford.edu/class/cs166/lectures/00/Slides00.pdf`

[37] CatchOrg. Catch2. `https://github.com/catchorg/Catch2/`, 2013, [Cited 2020-04-05].

[38] Google. Benchmark. `https://github.com/google/benchmark`, 2015, [Cited 2020-04-13].

[39] Mori, Y. An Implementation of the Induced sorting algorithm. `https://sites.google.com/site/yuta256/sais`, 2010, [Cited 2020-04-13].

[40] Fischer, J.; Heun, V. A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array. In *Proceedings of the First International Conference on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies*, ESCAPE'07, Berlin, Heidelberg: Springer-Verlag, 2007, ISBN 3540744495, p. 459–470.

Appendix A

Acronyms

ALIB The Algorithm Library Toolkit

DAG Directed acyclic graph

ESA Enhanced suffix array

LCP Longest common prefix

RMQ Range minimum query

SA Suffix array

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source codes
  - `implementation` ..... implementation sources
  - `text` ..... the directory of LaTeX source codes of the thesis
- `thesis.pdf` ..... the thesis text in PDF format