Karlsruhe Institute of Technology

Master's Thesis

# Engineering Fully-Compressed Suffix Trees

Christian Ocker

Submission Date: 29. May 2015

Supervisors: Prof. Dr. Peter Sanders
Dr. Simon Gog

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 29. Mai 2015

**Zusammenfassung**

Der Suffixbaum ist eine klassische Datenstruktur, die optimale Lösungen für zahlreiche Probleme aus dem Bereich der Stringverarbeitung ermöglicht. Eine zeigerbasierte Repräsentation des Suffixbaums benötigt für einen Text der Länge $n$ einen Speicheraufwand von $\Theta(n \log n)$ Bit, wohingegen kompakte Repräsentationen $\mathcal{O}(n)$ Bit zusätzlich zum Speicherbedarf des komprimierten Textes benötigen. Der Fully-Compressed Suffix Tree (FCST) bietet dieselbe Funktionalität und benötigt $o(n)$ Bit zusätzlich zum Speicherbedarf des komprimierten Textes, wohingegen die Abfragezeit um einen logarithmischen Faktor langsamer ist als bei kompakten Repräsentationen.

Im Rahmen dieser Arbeit wird eine generische Implementierung des FCST angefertigt, einschließlich einer kürzlich veröffentlichten Variante von Navarro und Russo, welche ein abweichendes Sampling verwendet, um eine bessere Zeitkomplexität zu erreichen. Es wird eine Variante des FCST vorgestellt, die einen auf blind search basierenden Ansatz verwendet, um String-Matching sowohl theoretisch als auch praktisch zu verbessern.

Es wird eine ausführliche empirische Evaluation und ein Vergleich der implementierten Datenstrukturen durchgeführt und gezeigt, dass die Implementierung dem vorherigen Prototypen in Platzbedarf sowie Abfrage- und Konstruktionszeit überlegen ist.

**Abstract**

The suffix tree is a classical data structure that provides optimal solutions to countless string processing problems. For a text of length $n$, a pointer-based representation of a suffix tree requires $\Theta(n \log n)$ bits, whereas compact representations use $\mathcal{O}(n)$ bits on top of the size of the compressed text. The fully-compressed suffix tree (FCST) provides the same functionality using $o(n)$ bits on top of the size of the compressed text, whereas queries are slowed down by a logarithmic factor compared to compact representations.

Our contribution is a generic implementation of the FCST, including a recent proposal by Navarro and Russo that uses a different sampling to achieve a better query time complexity. We propose a variant of the FCST that improves pattern matching both in theory and in practice using a blind search approach.

We provide an extensive empirical evaluation and comparison of the implemented data structures and show that our implementation outperforms the previous prototype in both space consumption and query/construction time.

# Acknowledgments

First of all, I would like to express my gratitude for the advice provided by Simon Gog, who guided me through the process of writing the thesis and taught me a lot. I also want to thank Peter Sanders and everyone in his research group for the opportunity to write the thesis and providing me with the necessary facilities. I thank Alexander Hantelmann and Niko Wilhelm for proofreading the thesis and Luís Russo for providing his implementation prototype.

I am grateful to my parents who made it possible for me to pursue a computer science degree and supported me in every way they could. I would also like to thank my family and friends for encouraging me during the writing of the thesis.

# Contents

# List of Figures



# List of Tables

# List of Algorithms

# 1 Introduction

## 1.1 Motivation

String matching is the problem of finding the occurrences of a small string, called the pattern, in a longer string, the text. Online string matching algorithms read the text sequentially and compare it with the pattern to find its occurrences. Although there are techniques like the Boyer-Moore [1] or the Knuth-Morris-Pratt algorithm [2] for speeding up this process compared to a naive solution, online string matching inherently requires time proportional to the size of the input to solve this problem.

If many patterns are to be found in the same text, it is beneficial to separate the process into two phases. In the first phase, the indexing phase, the text is preprocessed and an auxiliary data structure, called the index, is built. In the second phase, the querying phase, the index is used to solve the string matching problem. While the indexing phase may be time-consuming and still requires at least linear time, queries are generally much faster, both asymptotically and in practice.

The suffix tree is a classical data structure that can be used as an index to perform string matching. In his textbook, Gusfield [3] presents more than 20 applications which can be solved efficiently by using suffix trees. One essential aspect of suffix trees is that they can be constructed in linear time [4], so that indexed string matching can be applied without increasing the asymptotic complexity of the algorithm, allowing optimal solutions to many problems. For example, the problem of finding the longest common substring of two strings can be solved in linear time using suffix trees.

Suffix trees are commonly used in bioinformatics to match sequences of DNA or proteins, where natural borders are absent and character-based pattern matching, as the suffix tree provides it, is required. The problem with classical pointer-based suffix tree representations is that they are very large. Using a space-efficient implementation, the suffix tree of the human genome requires 45 GB of main memory [5], which is a lot more than the 700 MB the human genome itself occupies[1]. For practical purposes it is often resorted to the suffix array, a data structure that provides only part of the functionality of the suffix tree but requires less space in classical representations. The suffix array of the human genome requires about 11 GB of main memory[2]. Abouelhoda et al. [6] show how the suffix array can be extended by a longest common prefix array and a child table to provide the same functionality and time complexity as the suffix tree using three times the space of the suffix array.

A general problem of classical representations of both the suffix tree and the suffix array is their asymptotic space complexity of $\Theta(n \log n)$ bits, where $n$ is the length of the text. Its space requirement is thus growing faster than the size of the plain text, which is $\mathcal{O}(n \log \sigma)$ bits, where $\sigma$ is the number of different characters in the text. The large space requirements make suffix trees and suffix arrays impractical for large texts.

[1] The human genome consists of a slightly more than 3 billion base pairs. As there are four different base pairs, each one can be represented using 2 bits. Storing the genome as a sequence thus requires about 6 billion bits or 700 MB.

[2] The suffix array of the human genome consists of $n \approx 3 \cdot 10^9$ integer values with values in the interval $[0, n)$ so that each entry can be stored using $\lceil \log n \rceil = 32$ bits. In total, the $n$ entries require $n\lceil \log n \rceil \approx 11 \cdot 2^{30} = 11GB$.

In 2000, the first succinct indexes appeared. A succinct index is a data structure that provides efficient string matching capability and requires only space close to the size of the compressed text. One such index is the compressed suffix array by Grossi and Vitter [7] that exploits the structure of the suffix array to achieve a succinct representation, while still providing efficient access to its elements. Another succinct index is the FM-index by Ferragina and Manzini [8] which has the benefit of inherently being a self-index, i.e. it allows efficient reconstruction of the text or parts of it.

Sadakane introduced a compressed suffix tree [9] representation that enhances a compressed suffix array using auxiliary data structures with a space complexity of $\Theta(n)$ to efficiently support the full functionality of the suffix tree. Although much smaller than classical representations, the space required by the compressed suffix tree still depends linearly on the input size. This linear term is especially bad if the text is very repetitive, so that the compressed text is much smaller than the uncompressed text. For example, this is the case when indexing different versions of a file in a version control system.

The $\Theta(n)$ space barrier was first broken by the fully-compressed suffix tree proposed by Russo et al. [10], which supports full suffix tree functionality using only $o(n)$ bits on top of the size of the compressed suffix array, although this is bought with a significant slowdown compared to the compressed suffix tree by Sadakane. The fully-compressed suffix tree has been implemented by the authors of the original paper [11], although their implementation differs from the theoretical proposal. For example, their tree representation is not succinct.

Using our implementation, different versions of a Wikipedia article[3] amounting to a size of 446 MB of plain text can be represented by a fully-compressed suffix tree with a space consumption of 70 MB, whereas a compressed suffix tree of the same text (using the same underlying compressed suffix array[4]) requires 474 MB.

## 1.2 Contributions

We make several contributions in this thesis: We provide a generic implementation of the fully-compressed suffix tree (FCST) proposed by Russo et al. [11]. The implementation is based on the Succinct Data Structure Library (SDSL) [12] and can be parametrized with SDSL data structures. We implement a recent variant of the FCST proposed by Navarro and Russo [13] which uses a different sampling strategy to achieve a lower space complexity. One drawback of the above FCST is that pattern matching is relatively slow in theory and practice. We try to solve this problem by introducing a binary FCST variant, called BFCST, that uses a blind search approach for pattern matching.

In an extensive empirical evaluation we explore different time-space trade-offs of each FCST variant and compare the three variants. We show that our FCST implementation outperforms the prototype implementation provided by Russo et al. in both space consumption as well as query and construction time.

[3] The file contains all versions of the article *Albert Einstein* in the English Wikipedia up until November 10, 2006. It can be found in the repetitive corpus of Pizza&Chili at `http://pizzachili.dcc.uchile.cl/repcorpus.html`.

[4] The compressed suffix array that is used is an instantiation of the FM-index from the Succinct Data Structure Library that is given by the type `csa_wt<wt_huff<rrr_vector<255> >, 64, 64, text_order_sa_sampling<>, text_order_isa_sampling_support<> >`.

## 1.3 Structure of this Document

Section 2 gives an overview of basic data structures required for the understanding and implementation of the fully-compressed suffix tree. Section 3 introduces the fully-compressed suffix tree, a variant with a different sampling proposed by Navarro and Russo and a third variant that provides improved pattern matching using a binary tree structure. Section 4 shows experimental results and comparisons of the implemented data structures. Section 5 gives a summary of the results as well as open problems in the context of the thesis.

# 2 Theoretical Foundation

In this section we lay the theoretical foundation of the data structures that play a role in the implementation of the fully-compressed suffix tree.

## 2.1 Notation

A *string* $S$ is a sequence of *characters* with *length* $|S| = n$. Each character is an element of a totally ordered finite set called the *alphabet* $\Sigma$ with size $|\Sigma| = \sigma$. We denote by $S[i]$ the character at position i (starting at position 0) and by $S[i..j]$ the *substring* $S[i], S[i+1] \dots S[j]$. A *prefix* is a substring of the form $S[0..j]$, while a *suffix* is a substring of the form $S[i..n-1]$. The *concatenation* of a string $S$ and a character $\alpha$ is denoted by $S.\alpha$ and obtained by appending $\alpha$ at the end of $S$. The concatenation $S.S'$ of two strings $S$ and $S'$ is obtained by subsequently appending each character of $S'$ to the end of $S$. The *empty string* is denoted by $\epsilon$ and has length $|\epsilon| = 0$.

When indexing, we will generally have a large string $T$ that is called the *text* and a (usually) smaller string $P$, the *pattern*. We always assume that the text has a unique terminal character \$ that is smaller than any other character in the alphabet and does not appear anywhere else in the text. In the context of string matching, two subproblems are particularly interesting: The first one is *count*, calculating the number of occurrences of the pattern $P$ in the text $T$. The second one is *locate*, finding all such occurrences, i.e. determining their position in the text.

## 2.2 Suffix Trees and Basic Definitions

We start by defining a simple data structure for storing a set of strings, the trie.

**Definition 1.** *A* trie *of a set of strings* $\mathcal{S} = \{S_1, S_2 \dots S_n\}$ *is a rooted labeled tree, where each node represents a prefix of a string* $S \in \mathcal{S}$*. A node* $v$ *representing string* $S$ *has a child* $u$ *representing* $S'$ *iff* $S' = S.\alpha$ *for a character* $\alpha \in \Sigma$*. The edge between* $u$ *and* $v$ *is labeled* $\alpha$*. The root represents the empty string* $\epsilon$*.*

Figure 1 shows an example of a trie.

We will call the string that is represented by a node $v$ of a labeled tree its *path label*, as it is the concatenation of all the edge labels on the path from the root to $v$. Note that a

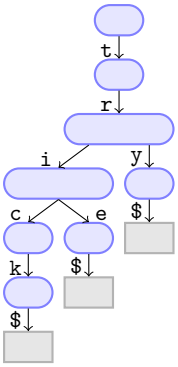


Figure 1: The trie of the strings 'trick$', 'trie$' and 'try$'. Inner nodes are represented by rectangles with rounded corners, while leaves are represented by rectangles without rounded corners.

node is uniquely determined by its path label. Therefore, we will refer to the node $v$ and its path label indifferently when the meaning is clear from context.

Assuming that every string in the trie is terminated by the sentinel character \$, which occurs nowhere else in the string, no string in the trie is a prefix of another and every string $S_i$ is represented by a leaf in the trie. To check whether a string $S$ is contained in the trie, we traverse the edges of the trie according to the characters in $S$ and check if we arrive at a leaf.

**Definition 2.** *The* suffix trie *of a text $T$ is the trie of all suffixes of $T$.*

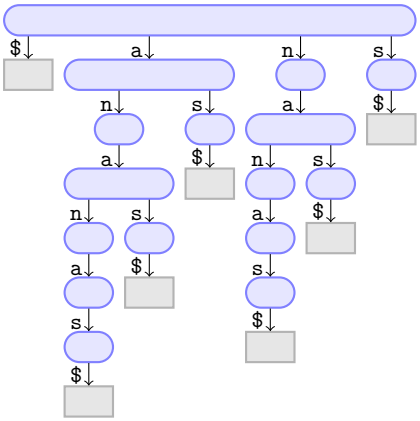


Figure 2: The suffix trie of the string 'ananas$'.

Figure 2 shows an example of a suffix trie.

The suffix trie can be used as an index of the text $T$ to find out whether a pattern $P$ of length $|P| = m$ is a substring of $T$. Since any substring in $T$ is a prefix of a suffix of $T$, $P$ has a representing node $v$ in the suffix trie iff $P$ is a substring of $T$.

By storing the children of each node in a lookup table, we can navigate to a child of a node in constant time. To find a node $v$ representing pattern $P$ in the trie, we traverse the tree according to the characters in $P$ until we arrive at the node $v$ after $m$ steps. To count the number of occurrences of $P$ in $T$, we count the number of leaves in the subtree rooted at node $v$, each of which represents such an occurrence.

The drawback of the suffix trie is that it has up to $\frac{n(n+1)}{2} + 1 = \mathcal{O}(n^2)$ nodes. Assuming each node only takes up as much space as its label, the suffix trie can be stored using $\mathcal{O}(n^2 \log \sigma)$ bits.

**Definition 3.** *The* compact trie *of a trie is obtained by removing each path without branching in the original trie and replacing it with a single edge labeled with the concatenation of the labels of the original edges.*

The compact trie is also known as the Patricia-Trie in literature.

**Definition 4.** *The* suffix tree *of a text $T$ is the compact trie of all suffixes of $T$.*

Figure 3 shows an example of a suffix tree.

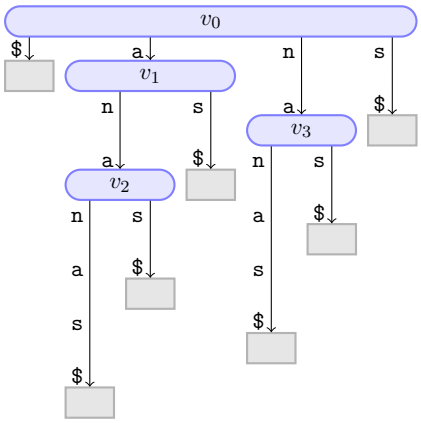


Figure 3: The suffix tree of the string 'ananas$'. This is the compact version of the trie shown in Figure 2.

In a suffix tree, the children of a node $v$ are distinguished by the first character of their edge labels, which we will call the *branching letter* of node $v$. For brevity, we will sometimes refer to the child $u$ of $v$ identified by the branching letter $\alpha \in \Sigma$ as $v.\alpha = u$, although the path label of $u$ may be longer.

Since there are $n$ leaves in the suffix tree and every inner node has at least two children, there are at most $n - 1$ inner nodes in the suffix tree. The total number of nodes of the suffix tree is thus $\mathcal{O}(n)$. Although the number of nodes is reduced when compared to the suffix trie, the number of characters in the labels remains the same. So instead of storing the edge labels explicitly, they are represented by a pointer to the text and the length of the label. In this representation, the suffix tree takes $\mathcal{O}(n \log n)$ bits of space.

Note that in contrast to the suffix trie, the suffix tree does not contain a node for every substring of $T$. For example, in the suffix tree in Figure 3 the string 'an' is not represented by a node but by a point on the edge between the nodes representing 'a' and 'ana'. Taking into consideration this difference, string matching is performed in a similar fashion to matching in the suffix trie.

In addition to string matching, there are a number of other operations that can be performed on suffix trees. One such operation is the suffix link.

**Definition 5.** *The* suffix link *of a node $v \neq root$, denoted by $slink(v)$, is the node $u$ so that $v = \alpha.u$ for a character $\alpha \in \Sigma$.*

The suffix link of a node $v$ is obtained by removing the first character from the path label of $v$. Notice that such a node always exists for any node in the suffix tree, except for the root. For example, in Figure 3, $slink(v_2) = v_3$, as the path label of $v_2$ is 'ana', while the path label of $v_3$ is 'na'.

The suffix link plays an important role in the construction of the suffix tree [14] and can be stored for each node without increasing the space complexity of the suffix tree.

Another interesting operation is the *lowest common ancestor* of two nodes $v$ and $w$, denoted by $LCA(v, w)$. This is equivalent to finding the longest common prefix of two strings represented by nodes in the suffix tree. The lowest common ancestor problem can be solved in constant time using $\mathcal{O}(n)$ bits [15].

## 2.3 Suffix Arrays

We will now introduce the suffix array, an index that provides part of the functionality of suffix trees and is particularly easy to implement.

**Definition 6.** *The* suffix array *of a text $T$ of size $n$ is an array $SA$ of size $n$ so that $T[SA[i]..n-1] < T[SA[i+1]..n-1]$ for $0 \leq i < n-1$ where $<$ is the lexicographic order on strings.*

$T[6..6] =$ `$`
$T[0..6] =$ `ananas$`
$T[2..6] =$ `anas$`
$T[4..6] =$ `as$`
$T[1..6] =$ `nanas$`
$T[3..6] =$ `nas$`
$T[5..6] =$ `s$`

| i | = | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|
| SA[i] | = | 6 | 0 | 2 | 4 | 1 | 3 | 5 |

Figure 4: The suffixes of $T =$ 'ananas$' sorted lexicographically (top) and the suffix array of the string 'ananas$' (bottom).

In other words, the suffix array contains the starting positions of the suffixes of $T$ sorted in lexicographic order. Figure 4 shows an example of a suffix array. The suffix array corresponds to the leaves of the suffix tree. Assuming we visit the children of each suffix tree node in the order of their branching letter, we obtain the suffix array by traversing the tree and writing down the starting positions of the suffixes represented by the leaves.

Since the suffixes are sorted, all suffixes with the same prefix are at consecutive positions in the suffix array. That means that suffix tree nodes, which themselves represent suffixes that share a common prefix, correspond to an interval in the suffix array. For example, node $v_1$ in the suffix tree in Figure 3 corresponds to the interval $[1, 3]$ in the suffix array in Figure 4. However, the tree structure is not represented in the suffix array, so that efficient navigation is not possible.

In combination with the text, the suffix array can be used to solve the string matching problem by performing a binary search. The time complexity for a *count* query is $\mathcal{O}(m \log n)$ for a pattern of length $m$. The suffix array can be stored in $n\lceil \log n \rceil = \mathcal{O}(n \log n)$ bits.

## 2.4 Bit Vectors

A basic building block of all succinct data structures we will present in this thesis are bit vectors (or binary sequences). In addition to efficient access to the value of a single element, we are interested in the following two operations:

For a bit vector $B$ of length $|B| = n$, $a \in \{0, 1\}$, $i \in [0, n]$ and $j \in [1, n]$,

- $rank_a(i, B)$ returns the number of times value $a$ occurs in $B[0..i-1]$.
- $select_a(j, B)$ returns the position of the $j$-th occurrence of value $a$ in $B$.

An equivalent definition of select is that $i = select_a(j, B)$ is the index in $B$ so that $rank_a(i, B) + 1 = j$ and $B[i] = a$.

Note that $rank_0(i, B)$ and $rank_1(i, B)$ can be reduced to one another by using the following relationship: $rank_0(i, B) = i - rank_1(i, B)$. Such a relationship does not exist for select [16].

### 2.4.1 Uncompressed Bit Vectors

We will now show how *rank* queries on a bit vector $B$ with size $|B| = n$ can be answered in constant time using only $o(n)$ bits of extra space. We will show the solution in two steps. The technique described here was first introduced in [17].

The first step divides the bit vector in blocks of size $t$. For each block $B[kt..kt + t - 1]$ we store the value of $rank_1(kt, B)$ explicitly. We choose $t = \frac{1}{2} \log n$, so that there are only $2^t = 2^{1/2 \log n} = \sqrt{n}$ possible blocks of size $t$ and we can store a lookup table that stores rank values for every position in every possible block relative to the beginning of the block. As each of these rank values can be stored in $\log(\frac{1}{2} \log n)$ bits, the lookup table takes up $\sqrt{n} \cdot \frac{1}{2} \log n \cdot \log(\frac{1}{2} \log n) = o(n)$ bits in total. The key used to look up the value in the table is the bit representation of the block itself. The lookup table is universal i.e. it does not depend on the data in the bit vector and can be reused for other instances of bit vectors. Figure 5 shows an example of this representation.

| i = | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $rank_1(3 \cdot \lfloor i/3 \rfloor, B) =$ | 0 | | | 2 | | | 3 | | | 5 | | |
| B[i] = | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 |

| i = | 1 | 2 | 3 |
|---|---|---|---|
| 000 | 0 | 0 | 0 |
| 001 | 0 | 0 | 1 |
| 010 | 0 | 1 | 1 |
| 011 | 0 | 1 | 2 |
| 100 | 1 | 1 | 1 |
| 101 | 1 | 1 | 2 |
| 110 | 1 | 2 | 2 |
| 111 | 1 | 2 | 3 |

Figure 5: A bit vector and its precalculated rank values (left) as well as the universal lookup table (right).

In this representation, the value $rank_1(i, B)$ can be calculated as the sum of $rank_1(t \cdot k, B)$ and $rank_1(i \bmod t, B[kt..kt + t - 1])$, where $k = \lfloor i/t \rfloor$. The first rank is precalculated for all blocks and the second rank is stored in the universal table.

However, to store the value of $rank_1(kt, B)$ for each block, we need $\frac{n}{t} \log n = \frac{n}{1/2 \log n} \log n = \Theta(n)$ bits. We will now show how to reduce this space requirement by adding another layer of blocks.

In the second step, we divide the bit vector into superblocks of size $s = \log^2 n$ so that every superblock consists of $\frac{s}{t} = 2 \log n$ blocks. We can store the rank value at the beginning of each superblock using $\frac{n}{\log^2 n} \log n = \frac{n}{\log n} = o(n)$ bits. The rank value at the beginning of each block is stored relative to the beginning of the superblock, which requires $\frac{n}{t} \log s = \frac{n}{1/2 \log n} \log \log^2 n = o(n)$ bits.

That way we can answer a rank query in constant time using $o(n)$ bits of space.

The *select* operation can also be solved in constant time using $o(n)$ bits [18]. The solution consists of dividing the bit vector into blocks, each of which contains $r$ one bits. Large blocks store the positions of the one bits explicitly while smaller blocks allow *select* to be solved by a scan over a constant number of bits. Note that although both *rank* and *select* are constant time operations, *rank* requires only 3 random memory accesses while *select* may require a scan over a larger portion of the bit vector. Therefore, *rank* is commonly faster than *select* in practice.

### 2.4.2 Sparse Bit Vectors

We will make use of bit vectors that are very sparsely populated, i.e. the number of ones is much lower than the number of zeros. Storing these bit vectors uncompressed is wasteful. In the following we will present a way of representing sparse bit vectors proposed by Okanohara and Sadakane [19] which is based on an encoding in [7] and allows efficient *rank* and *select* queries.

Given a bit vector $B$ of size $|B| = n$ with $m$ ones (and $n - m$ zeros), we define a vector $X$ of length $|X| = m$ so that $X[i] = select_1(i + 1, B)$. $X$ represents the positions of the set bits in $B$. For a parameter $t$, we represent $X$ using two data structures: $H$ represents the upper $h = \lceil \log t \rceil$ bits of the values in $X$ and $L$ the lower $l = \lceil \log n \rceil - \lceil \log t \rceil$ bits of the values in $X$.

We define two functions that map each value $x$ to their two parts. The function $high(x) = \lfloor x/2^l \rfloor$ gives the value of the upper part of $x$, while $low(x) = x \bmod 2^l$ gives the lower part.

We consider the vector $D$ with $D[0] = high(X[0])$ and $D[i] = high(X[i]) - high(X[i-1])$ for $0 < i < m$ that gives the differences between the upper part of two subsequent entries in $X$. H is the unary representation of the values in $D$, i.e. for each entry in $D$, $H$ contains $D[i]$ zeros terminated by a one. $L$ is stored explicitly using $m \cdot l$ bits. Figure 6 shows an example.

Since the values in $D$ sum up to $high(X[m-1])$, their sum has an upper bound of $2^h$, so there are at most $2^h = 2^{\lceil \log t \rceil}$ zeros and $m$ ones in $H$ and it requires at most $t + m$ bits, while $L$ stores $m$ entries of size $\log \frac{n}{t}$ each using $m \log \frac{n}{t}$ bits (neglecting rounding). In total, $H$ and $L$ require $t + m + m \log \frac{n}{t}$ bits, which is minimal for $t = m \log e \approx 1.44m$. We use the data structure from Section 2.4.1 to allow *rank* and *select* queries on $H$ and $L$, which is asymptotically smaller than the respective bit vector.

$$B = 000100000001010000100100110010000$$

| X[i] | high(X[i]) | low(X[i]) | D[i] |
|---|---|---|---|
| 3 | 0001 | 1 | 1 |
| 11 | 0101 | 1 | 4 |
| 13 | 0110 | 1 | 1 |
| 18 | 1001 | 0 | 3 |
| 21 | 1010 | 1 | 1 |
| 24 | 1100 | 0 | 2 |
| 25 | 1100 | 1 | 0 |
| 28 | 1110 | 0 | 2 |

$$H = 01\ 00001\ 01\ 0001\ 01\ 001\ 1\ 001$$
$$L = 11101010$$

Figure 6: Example of the representation of a sparse bit vector. The values in $X$ are separated into 4 high bits and 1 low bit.

To answer $rank_1(i, B)$, we first query $rank_1(select_0(high(i), H), H) = select_0(high(i), H) - high(i) + 1$ to find the number of ones in $B[0..high(i) \cdot 2^l - 1]$. The remaining number of ones in $B[high(i) \cdot 2^l..i - 1]$ is then determined by performing a binary search over the values in $X$ (represented by $H$ and $L$). In the worst case, there are up to $m$ ones in this interval, resulting in $\mathcal{O}(\log m)$ time for this step. On average, there are only $\frac{m}{t} = \mathcal{O}(1)$ ones in this interval, resulting in $\mathcal{O}(1)$ average runtime.

Selection of the $j$-th one is performed in the following way:

$$\begin{aligned} select_1(j, B) &= rank_0(select_1(j, H)) \cdot 2^l + L[j - 1] \\ &= (select_1(j, H) - j + 1) \cdot 2^l + L[j - 1] \end{aligned}$$

We recall that $rank_0$ queries can be reduced to $rank_1$ queries in the following way:

$$rank_0(i, B) = i - rank_1(i, B)$$

It remains to be shown how to solve $select_0$ queries. One way is finding the largest $j$ so that $select_1(j, B) - j + 1 < i$ using binary search. The result is $select_0(i, B) = i - 1 + j$. Since there are $m$ ones in $B$, this takes $\log m$ steps.

To summarize, we can represent a bit vector $B$ of size $|B| = n$ with $m$ ones using $\mathcal{O}(m \log \frac{n}{m})$ bits, allowing $rank$ queries in $\mathcal{O}(\log m)$ time in the worst case and constant time in the average case, $select_1$ queries in constant time and $select_0$ queries in $\mathcal{O}(\log m)$ time.

### 2.4.3 Entropy-Compressed Bit Vectors

The bit vector presented in Section 2.4.2 is well-suited to represent sparse bit vectors. Raman et al. propose a compressed bit vector representation that supports $rank$ and

*select* queries in constant time and requires space close to the information-theoretic lower bound for arbitrary binary sequences [20].

The basic idea is to divide the bit vector into blocks of length $t$ and group them into classes according to the number of ones they contain. The class $C_m$ contains all blocks that have $m$ ones. Each block in the bit vector is identified by a pair $(c, o)$ giving its class $c$ and the offset $o$ which is the position of the block inside its class. Note that the number of blocks in each class varies depending on $m$. The class $C_m$ contains $\binom{t}{m}$ different blocks, so that an offset in class $C_m$ can be stored using $\lceil \log \binom{t}{m} \rceil$ bits. Hence, blocks that are very dense or very sparse have a shorter representation, resulting in a compression for appropriately shaped bit vectors.

For a detailed description of the data structure and how to perform *rank* and *select* queries, we refer the reader to the original publication by Raman et al. [20].

## 2.5 Wavelet Trees

In Section 2.4, we have solved *rank* and *select* for bit vectors. We will now extend the *rank* and *select* operations to strings of characters over the alphabet $\Sigma$:

For a string $S$ of length $|S| = n$, $\alpha \in \Sigma$, $i \in [0, n]$ and $j \in [1, n]$,

- $rank_\alpha(i, S)$ returns the number of times character $\alpha$ occurs in $S[0, i-1]$.
- $select_\alpha(j, S)$ returns the position of the $j$-th occurrence of character $\alpha$ in $S$.

A naive solution is to store a separate bit vector for every character $\alpha \in \Sigma$, allowing constant time *rank* and *select* using $\mathcal{O}(n\sigma)$ space. Using the wavelet tree [21], we can trade off some runtime for storage space and perform *rank* and *select* in time $\mathcal{O}(\log \sigma)$ using $\mathcal{O}(n \log \sigma)$ space.

A *wavelet tree* is a balanced binary search tree in which every node $v$ represents a set of characters. The tree has $\sigma$ leaves, each representing a character $\alpha \in \Sigma$ and $\sigma - 1$ inner nodes, representing the union of their respective children. Let $S_v$ be the subsequence of $S$ that contains all occurrences of characters represented by the node $v$. Note that this sequence is only conceptual and will not be stored. Instead, for every inner node $v$ in the wavelet tree, we store a bit vector $B_v$ that indicates for each character in $S_v$ if that character is represented by the left (= 0) or the right (= 1) child of $v$. Figure 7 shows an example of a wavelet tree.

We solve $rank_\alpha(i, B)$ by traversing the wavelet tree from the root to the node representing $\alpha$. By performing *rank* on the bit vector associated with each node we visit, we get the position of the character in the subsequence associated with the next child. The last position, the position of the character in the subsequence associated with the leaf, is $rank_\alpha(i, B)$.

For example, we compute $rank_i(9, S)$ for $S =$ 'ipssm\$pissii' using the wavelet tree shown in Figure 7 in the following way:

$$rank_i(9, S) = \underbrace{rank_1(rank_0(\underbrace{rank_0(9, B_0)}_{6}, B_1), B_2)}_{3} = 2$$

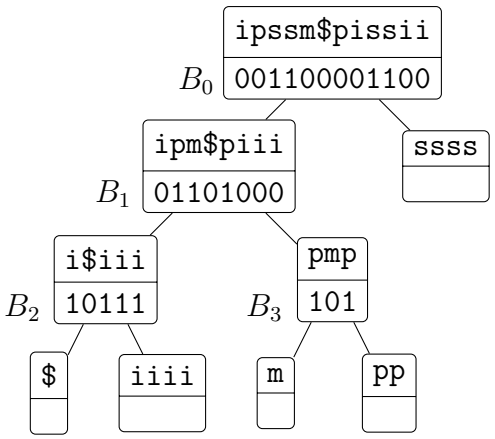


Figure 7: A wavelet tree for the string 'ipssm$pissii', which is the Burrows-Wheeler transform of 'mississippi$'. We will introduce the Burrows-Wheeler transform in Section 2.7. Each node is represented by its subsequence of S (upper half) and its bit vector (lower half).

We solve $select_\alpha(j, B)$ by traversing the tree bottom-up from the leaf that represents $\alpha$ to the root. At each step, we perform a select on the bit vector associated with the node, which yields the position of the character in the subsequence associated with the parent node. The last position is the position of the character in the sequence associated with the root, which is $S$ itself.

For example, we compute $select_p(2, S)$ for $S =$ 'ipssm$pissii' using the wavelet tree shown in Figure 7 in the following way:

$$select_p(2, S) = select_0(\underbrace{select_1(\underbrace{select_1(2, B_3)}_{2} + 1, B_1)}_{4} + 1, B_0) = 6$$

Since the tree is balanced, it has $\lceil \log \sigma \rceil$ levels. We solve *rank* and *select* on each level in constant time using the method described in Section 2.4.1, so that we need $\mathcal{O}(\log \sigma)$ time for *rank* and *select* using the wavelet tree. As the sets of characters represented by nodes in the same level are disjoint, each level contains at most $n$ bits and the total amount of space required for the wavelet tree is $\mathcal{O}(n \log \sigma)$ bits.

## 2.6 Succinct Trees

In classical suffix tree representations, the tree structure is represented by pointers connecting the nodes of the tree with each other. For a tree with $n$ nodes, a pointer needs to distinguish between all of these nodes and hence requires at least $\log n$ bits, adding up to $n \log n$ bits for the tree.

### 2.6.1 Balanced Parentheses Sequences

One way to achieve a succinct representation of a tree is storing it as a sequence of balanced parentheses so that every node corresponds to a pair of parentheses. Every node

is enclosed by the pair of parentheses representing its parent, except for the root, which is represented by the first (opening) and last (closing) parenthesis. The tree represented this way is an *ordinal tree*, i.e. the children of each node have an explicit order, which is represented by the order of their appearance in the balanced parentheses sequence (BPS in the following). Figure 8 shows an example of an ordinal tree and its BPS.

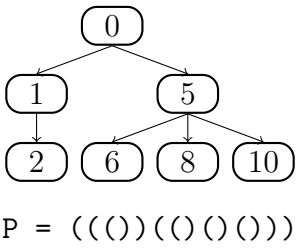


Figure 8: An ordinal tree and its balanced parentheses sequence. The nodes are labeled with the position of their opening parenthesis in the BPS.

The BPS of a tree can be constructed by performing a depth first search on the tree, writing an opening parenthesis when visiting a node for the first time and a closing parenthesis after all its children have been visited.

We are interested in a number of navigational operations on the tree:

- *root* returns the root of the tree.
- *is-leaf*$(v)$ returns whether the node $v$ is a leaf.
- *ancestor*$(v, v')$ returns whether $v$ is an ancestor of $v'$.
- *parent*$(v)$ returns the parent of a node $v$.
- *first-child*$(v)$/*last-child*$(v)$ return the first/last child of a node $v$.
- *next-sibling*$(v)$/*prev-sibling*$(v)$ return the next/previous sibling of a node $v$.
- $LCA(v, v')$ returns the lowest common ancestor of $v$ and $v'$.
- *pre-order*$(v)$ returns the number of nodes that are visited before $v$ in a pre-order traversal.
- *left-rank*$(v)$ returns the number of leaves that are visited before $v$ in a pre-order traversal.
- *right-rank*$(v)$ returns the number of leaves that are visited before the last leaf in the subtree of $v$ in a pre-order traversal.

Note that the ancestor relation is the transitive reflexive closure of the parent relation. In particular, this implies that any node $v$ is an ancestor of itself.

All the above operations can be implemented using the following operations on the BPS:

- *inspect*$(i)$ returns whether the parenthesis on position i is an opening parenthesis.
- $rank_{(}(i)$ returns the number of opening parentheses in the interval $[0, i-1]$.
- $rank_{()}(i)$ returns the number of occurrences of the pattern '()' in the interval $[0, i-1]$.

- $find\text{-}close(i)$ returns the position of the closing parenthesis matching the opening parenthesis on position $i$.
- $find\text{-}open(i)$ returns the position of the opening parenthesis matching the closing parenthesis on position $i$.
- $enclose(i)$ returns the position of the rightmost opening parenthesis that together with its matching parenthesis encloses the parenthesis on position $i$.
- $double\text{-}enclose(i, j)$ returns the position of the rightmost opening parenthesis that together with its matching parenthesis encloses both $i$ and $j$.

Nodes are identified by the position of their opening parenthesis in the BPS. In the following, we will refer to nodes and the position of their opening parenthesis in the BPS indifferently. The navigational operations are implemented using the operations on the balanced parentheses sequence as follows:

$$\begin{aligned}
root &= 0 \\
is\text{-}leaf(v) &= inspect(v+1) = \text{')'} \\
ancestor(v, v') &= v \leq v' \wedge find\text{-}close(v') \leq find\text{-}close(v) \\
parent(v) &= enclose(v) \\
first\text{-}child(v) &= v + 1 \\
last\text{-}child(v) &= find\text{-}open(find\text{-}close(v) - 1) \\
next\text{-}sibling(v) &= find\text{-}close(v) + 1 \\
prev\text{-}sibling(v) &= find\text{-}open(v - 1) \\
LCA(v, v') &= double\text{-}enclose(v, v') \\
pre\text{-}order(v) &= rank_{(}(v) \\
left\text{-}rank(v) &= rank_{()}(v) \\
right\text{-}rank(v) &= rank_{()}(find\text{-}close(v)) - 1
\end{aligned}$$

In the following, we will introduce a data structure proposed by Sadakane and Navarro [22] that solves the above operations efficiently and is small and fast in practice.

We denote by $P$ a BPS of length $|P| = n$. We will implement P as an uncompressed bit vector and represent opening and closing parentheses by 1 and 0 respectively. The *inspect* operation is just an access to $P$. We can solve $rank_{(}(i) = rank_1(i, P)$ in constant time using the data structure from Section 2.4.1. We can also solve $rank_{()}(i)$ using the same data structure by evaluating $rank_1(i, P')$ for a bit vector $P'$ of length $|P'| = n - 1$ which is defined as follows:

$$P'[i] := \begin{cases} 1 & \text{if } P[i] = 1 \wedge P[i+1] = 0 \\ 0 & \text{else} \end{cases}$$

$P'$ is not stored explicitly. Its values can be computed when necessary by accessing $P$.

We will now focus on the remaining operations and define the following auxiliary functions:

**Definition 7.** *[22] For a bit vector $P[0, n-1]$ and a function $g(\cdot) \in \{0,1\} \times \{-1,1\}$,*

$$\begin{aligned}
sum(P,g,i,j) &= \sum_{k=i}^{j} g(P[k]) \\
fwd\text{-}search(P,g,i,d) &= \min_{j \geq i}\{j \mid sum(P,g,i,j) = d\} \\
bwd\text{-}search(P,g,i,d) &= \max_{j \leq i}\{j \mid sum(P,g,j,i) = d\} \\
rmq(P,g,i,j) &= \min_{i \leq k \leq j}\{sum(P,g,i,j)\} \\
rmqi(P,g,i,j) &= \operatorname{argmin}_{i \leq k \leq j}\{sum(P,g,i,j)\} \\
RMQ(P,g,i,j) &= \max_{i \leq k \leq j}\{sum(P,g,i,j)\} \\
RMQi(P,g,i,j) &= \operatorname{argmax}_{i \leq k \leq j}\{sum(P,g,i,j)\}
\end{aligned}$$

**Definition 8.** *[22] Let $\pi$ be the function such that $\pi(1) = 1$ and $\pi(0) = -1$. Given $P[0, n-1]$, we define the* excess array *$E[0, n-1]$ of $P$ as an integer array such that $E[i] = sum(P, \pi, 0, i)$.*

$E$ stores for every position in $P$ the *excess*, i.e. the difference between the number of opening and closing parentheses to the left of that position.

The functions from Definition 7 can be used to implement the remaining operations:

$$\begin{aligned}
find\text{-}close(i) &= fwd\text{-}search(P,\pi,i,0) \\
find\text{-}open(i) &= bwd\text{-}search(P,\pi,i,0) \\
enclose(i) &= bwd\text{-}search(P,\pi,i,2) \\
double\text{-}enclose(i,j) &= \begin{cases} i & \text{if } ancestor(i,j) \\ j & \text{if } ancestor(j,i) \\ enclose(rmqi(P,\pi,i,j)+1) & \text{else} \end{cases}
\end{aligned}$$

### 2.6.2 The Range Min-Max Tree

The functions from Definition 7 can be solved efficiently using a range min-max tree.

**Definition 9.** *[22] A* range min-max tree *for a vector P[0, n-1] and a function $g(\cdot)$ is defined as follows: Let $[l_1..r_1], [l_2..r_2] \ldots [l_q, r_q]$ be a partition of $[0..n-1]$ where $l_1 = 0, r_i + 1 = l_{i+1}, r_q = n-1$. Then the i-th leftmost leaf of the tree stores the sub-vector $P[l_i, r_i]$, as well as $e[i] = sum(P, g, 0, r_i)$, $m[i] = e[i-1] + rmq(P, g, l_i, r_i)$ and $M[i] = e[i-1] + RMQ(P, g, l_i, r_i)$. Each internal node u stores in e[u]/m[u]/M[u] the last/minimum/maximum of the e/m/M values stored in its child nodes. Thus, the root node stores $e = sum(P, g, 0, n-1)$, $m = rmq(P, g, 0, n-1)$, and $M = RMQ(P, g, 0, n-1)$.*

We divide $P$ into blocks of size $s$, so that $B_i = P[is..is+s-1]$. Each block is a leaf of the range min-max tree. We will assume that the range min-max tree is $k$-ary for a constant value of $k$ and complete, so that the values of $e$, $m$ and $M$ can be stored in an integer array, like the nodes of a heap. An example of such a tree is shown in Figure 9.

Note that the values of the excess array are not stored explicitly. The value of $E[i]$ can be computed from the last excess value $e[j]$ of the preceding block $j = \lfloor i/s \rfloor$ and the values in $P$.

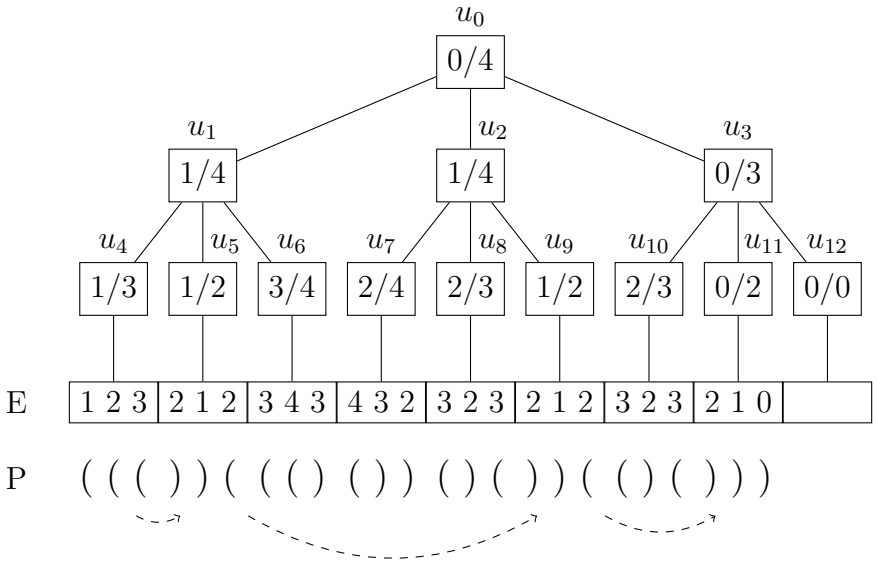


Figure 9: An example of a range min-max tree for the excess array and $k = s = 3$. Each node $u_i$ is represented by its minimum/maximum value $m[u_i]/M[u_i]$.

We will now show how $fwd\text{-}search(P, \pi, i, d)$ can be solved using a range min-max tree over the excess array (i.e. using the $\pi$ function). The solution for *bwd-search* is symmetric.

We choose $s = \frac{1}{2}\log n$, so that we can pre-compute $fwd\text{-}search(B, \pi, i, d)$ for $B \in \{0,1\}^s, i \in [0, s-1], d \in [-s, s]$ as a universal lookup table in space $2^s \cdot s \cdot (2s+1)\lceil \log s \rceil = \mathcal{O}(\sqrt{n}\log^2 n \log\log n) = o(n)$, similar to what we did in Section 2.4.1.

We are looking for the first value $j \geq i$ so that $sum(P, \pi, i, j) = d$. We first check whether $fwd\text{-}search(P, \pi, i, d)$ is in the same block as $i$ using the universal lookup table. If this is not the case, we start looking for $j \geq i$ so that $E[j] = d'$ with $d' = E[i-1] + d$. We check the next node $u$, which is either the right sibling of the current node or the sibling of its parent node if the current node has no right sibling. The range represented by the node $u$ contains $fwd\text{-}search(P, \pi, i, d)$ iff $m[u] \leq d' \leq M[u]$. If we find such a node, we start descending into its subtree until we arrive at the leaf that contains the result. We find the position of the parenthesis inside the block using the universal lookup table.

In the process, we need to check at most $2k$ nodes at each of the $\log_k n/s$ levels of the tree, amounting to $\mathcal{O}(\log n)$ time.

Consider we want to calculate the value of $find\text{-}close(5) = fwd\text{-}search(P, \pi, 5, 0)$ in the example depicted in Figure 9. We start by checking block $u_5$, which contains the index 5. We look up $fwd\text{-}search(P[3..5], \pi, 2, 0)$ in the universal lookup table and find that $u_5$ does not contain $fwd\text{-}search(P, \pi, 5, 0)$. We continue by searching for the index $j \geq 5$ so that $E[i] = d'$ with $d' = E[5-1] + 0 = 1$. Next, we check sibling $u_6$, which does not contain the resulting value since $3 = m[u_6] > d' = 1$. As $u_6$ has no right sibling, we check $u_2$, the sibling of its parent. Since $m[u_2] \leq d' \leq M[u_2]$, we descend into $u_2$ and check $u_7$, $u_8$ and $u_9$. The position of the final result is calculated as the sum of $fwd\text{-}search(P[15..17], \pi, 0, d' - E[14]) = 1$, which is retrieved using the universal lookup table, and the beginning offset of block $u_9$, 15. The final result is thus 16.

The $rmqi(P, \pi, i, j)$ and $RMQI(P, \pi, i, j)$ function are solved in a similar fashion, although

the process comprises two steps. In the first step, we examine up to $2k$ nodes that cover the interval $[i, j-1]$. In the second step, we find the leftmost of these nodes that contains the minimum excess value and descend into its subtree to find the position of said minimum.

The range min-max tree has $\frac{k\lceil n/s \rceil - 1}{k-1} = \mathcal{O}(\frac{n}{s})$ nodes, each of which requires $\log n$ bits to store each value of $e$, $m$ and $M$. We use $s = \frac{1}{2}\log n$, so that the universal lookup table requires $o(n)$ bits and $e$, $m$, and $M$ require $\mathcal{O}(n)$ bits. We can reduce the space consumption to $o(n)$ bits of extra space by introducing superblocks and storing values relative to the beginning of the superblock. Also, it is not necessary to store $e$ explicitly, since its values can be calculated by $E[i] = 2 \cdot rank(i, P) - i$.

Note that although the approach we outlined here performs well in practice, all operations can also be solved in constant time using $o(n)$ bits of extra space [23].

## 2.7 Compressed Suffix Arrays

The term *compressed suffix array* (CSA) was coined by Grossi and Vitter [7], who proposed the first compressed suffix array representation. We will use the term in a more general sense to refer to different succinct data structures that emulate the functionality of the suffix array. In particular, we will concentrate on the FM-index by Ferragina and Manzini [8], which is based on backward search using the Burrows-Wheeler transform [24]. In addition to the functionality that suffix arrays provide, the FM-index and other compressed suffix arrays provide a number of additional operations, which we will introduce in this section. Also note that in contrast to the suffix array the FM-index does not need the original text to perform any of its operations. Moreover, it can be used to efficiently reconstruct the text or parts of it.

### 2.7.1 Burrows-Wheeler Transform

We start by introducing the Burrows-Wheeler transform (BWT in the following) and showing how it can be used to index a text. The Burrows-Wheeler transform is commonly defined using the rotations of the text.

**Definition 10.** *A* rotation *of the string $T$ with $n = |T|$ is a concatenation of a suffix and a prefix of the form $T[i..n-1].T[0..i-1]$ for $0 \leq i < n$.*

We consider a matrix of size $n \times n$, which contains as its rows all rotations of the string $T$, sorted lexicographically. Figure 10 (left) shows an example of such a matrix. The last column (denoted by $L$) represents the Burrows-Wheeler transform. The first column is denoted by $F$. Note that the matrix is only conceptual and will not be stored.

As each rotation begins with a suffix of $T$ and each suffix has a terminal character \$ which does not occur anywhere else in the text, the order of the rotations is the same as the order of the suffixes in the suffix array. The BWT can thus be computed using the suffix array of the text $T$ by concatenating for each suffix array entry the character preceding it in the text:

$$L[i] = T[(SA[i] - 1) \bmod n]$$

| **F** | | | | | | | | | | | **L** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $ | m | i | s | s | i | s | s | i | p | p | **i** |
| i | $ | m | i | s | s | i | s | s | i | p | **p** |
| i | p | p | i | $ | m | i | s | s | i | s | **s** |
| i | s | s | i | p | p | i | $ | m | i | s | **s** |
| i | s | s | i | s | s | i | p | p | i | $ | **m** |
| m | i | s | s | i | s | s | i | p | p | i | **$** |
| p | i | $ | m | i | s | s | i | s | s | i | **p** |
| p | p | i | $ | m | i | s | s | i | s | s | **i** |
| s | i | p | p | i | $ | m | i | s | s | i | **s** |
| s | i | s | s | i | p | p | i | $ | m | i | **s** |
| s | s | i | p | p | i | $ | m | i | s | s | **i** |
| s | s | i | s | s | i | p | p | i | $ | m | **i** |

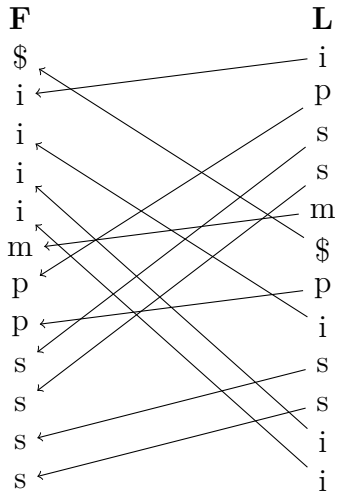


Figure 10: All rotations of the string 'mississippi$', sorted lexicographically (left) as well as an illustration of the LF mapping (right). The Burrows-Wheeler transform consists of the characters in the last column (labeled L) read from top to bottom.

We store $L$ using a wavelet tree (see Section 2.5), allowing efficient *rank* and *select* queries. We also need a representation for $F$. Since the rotations are sorted, $F$ can be split into $\sigma$ ranges each consisting only of multiple repetitions of a single character. Therefore, we represent $F$ as an array $C$ that contains for each character $\alpha \in \Sigma$ the position in $F$ at which the range of $\alpha$ begins:

$$C[\alpha] \coloneqq select_\alpha(1, F)$$

We determine the value of $F[i]$ using binary search over $C$.

### 2.7.2 LF and $\psi$ Function

When dealing with compressed suffix arrays, we have to distinguish between different orders of the rotations (or suffixes). The *suffix order* is their lexicographic order (the order in the suffix array). For instance, the rotation preceding 'ippi$mississ' in suffix order is 'i$mississipp'. The *text order* is the order of the rotations according to the position of their first character in the text. For example, the rotation preceding 'ippi$mississ' in text order is 'sippi$missis'.

The LF (last-to-front) function maps each rotation to its preceding rotation in text order.

**Definition 11.** *Given a suffix array $SA$, the* LF mapping *is a function $LF$ so that for $0 \leq i < n$, the following holds:*

$$SA[LF(i)] \equiv SA[i] - 1 \ (mod\ n)$$

If we consider all rotations that have the same character $\alpha$ in $L$, we observe that the rotations preceding them in text order are also in sorted order since each of them starts with the same character $\alpha$ and the subsequent string is a prefix of the original rotation, which is sorted. Therefore, the $i$-th occurrence of a character in $L$ has the same position in the text as the $i$-th occurrence of said character in $F$. This observation, which is

illustrated in Figure 10 (right), can be used to calculate the $LF$ function using only $C$ and $L$:

$$LF(i) = C[L[i]] + rank_{L[i]}(i, L)$$

Access to $L$ and $rank$ both take $\mathcal{O}(\log \sigma)$ time while $C$ can be accessed in constant time. Therefore, we can calculate $LF$ in $\mathcal{O}(\log \sigma)$ time. As $LF$ is a building block of the algorithms we present in this thesis, we will denote the time required for the evaluation of $LF$ with $t_{LF} = \mathcal{O}(\log \sigma)$.

The $LF$ function can be used to reconstruct the text by traversing the text in reverse direction. At each step we retrieve the character at the beginning of the rotation by looking at $C$.

The inverse of the $LF$ function is the $\psi$ function, which maps each rotation to its subsequent rotation in text order:

**Definition 12.** *Given a suffix array $SA$, $\psi$ is a function so that for $0 \leq i < n$, the following holds:*

$$SA[\psi(i)] \equiv SA[i] + 1 \ (mod\ n)$$

Note than $\psi$ is similar to $slink$, as $slink(v) = \psi(v)$ for a leaf $v$ of a suffix tree. The $\psi$ function is calculated using $C$ and $L$ analog to the $LF$-function:

$$\psi(i) = select_{F[i]}(i - C[F[i]] + 1, L)$$

We recall that $F$ is determined via binary search over $C$.

We denote the time required for the evaluation of $\psi$ by $t_\psi = \mathcal{O}(\log \sigma)$. Note that although $LF$ and $\psi$ have the same time complexity, in practice the evaluation of $LF$ is faster than $\psi$ for FM-indexes, because $rank$ is faster than $select$ (see 2.4.1).

### 2.7.3 Backward Search

We will now show how the Burrows-Wheeler transform can be used to solve the string matching problem. Since the rotations in the BWT, just like the suffixes in the SA, are sorted, we can represent all rotations that share a common prefix as an interval of rotations. In the following, we will refer indifferently to a string $v$ and the interval of rotations that are prefixed by $v$.

We use a concept similar to the $LF$ function to find a pattern in the Burrows-Wheeler transform. Suppose we want to find all occurrences of the pattern $P$ in the text. We do this by performing a stepwise backward search, starting with the empty pattern and subsequently finding all occurrences of a suffix of $P$ that is one character longer than the pattern matched in the previous step. Algorithm 1 shows how backward search is performed.

After at most $m$ steps, we obtain the interval that represents all rotations starting with the pattern $P$. The size of the interval tells us the number of appearances of pattern $P$. If the interval is empty, the pattern does not appear in the text. Using backward search we can solve $count$ in $2m \cdot t_{LF} = \mathcal{O}(m \log \sigma)$ time using the FM-index.

**Algorithm 1:** Backward search using the FM-index.

```
1  Algorithm backward-search(P[0, m − 1])
2      i ← m
3      s ← 0
4      e ← n
5      while i > 0 and s < e do
6          α ← P[i − 1]
7          s ← C[α] + rank_α(s, L)
8          e ← C[α] + rank_α(e, L)
9          i ← i − 1
10     return [s, e − 1]
```

Performing one step of backward search is the equivalent of adding a character $\alpha$ to the left of the string $X$. Due to its relationship to the $LF$ function, we will denote the operation as $LF(\alpha, X)$, where $X$ is a string represented by an interval over the CSA and $\alpha$ is the character to be appended to the string. It is calculated as follows:

$$LF(\alpha, [s, e]) = [C[\alpha] + rank_\alpha(s, L), C[\alpha] + rank_\alpha(e + 1, L) - 1]$$

Moreover, we extend $LF$ to strings, so that $LF(Y, X)$ yields the interval representing the string $Y.X$, which is obtained by successively applying $LF$ with the characters in $Y$.

In the context of suffix trees, $LF$ is also known as the *Weiner link*, which is the inverse of the suffix link. For any suffix tree node $v \neq root$:

$$LF(\alpha, slink(\alpha.v)) = \alpha.v$$

### 2.7.4 Locate Functionality

We now know how to navigate back- and forwards in the text using the $LF$ and $\psi$ functions. We also know how to apply string matching and count the number of occurrences of a pattern. But to actually find the positions of these occurrences in the text and solve *locate* queries, we need to access the values in the suffix array. As storing the SA completely requires too much space, we will store only a sampling of the SA and use $LF$ to navigate to a sampled position.

One way of sampling the SA is storing the value of $SA[i]$ if $i \equiv 0 \pmod{\theta}$ where $\theta$ is the sampling factor. This is called a *suffix order sampling*, as the position in the SA determines whether an entry is sampled. According to the definition of $LF$, $SA[i] = SA[LF(i)] + 1$. To get the value of $SA[i]$ if it is not sampled, we apply $LF$ until we find a value that is sampled. In the worst case, we need $n - \frac{n}{\theta}$ applications of $LF$.

Another way of sampling the SA is storing all values of $SA[i]$ for which $SA[i] \equiv 0 \pmod{\theta}$. This is called a *text order sampling*, as the position in the text determines whether an entry is sampled. On average, we need $\frac{\theta}{2}$ applications of $LF$ to calculate the value of $SA[i]$. In the worst case, we apply it $\theta$ times. However, since we do not know for which values of $i$ the value of $SA[i]$ is sampled, we have to store this information explicitly using an additional bit vector. Since this bit vector is sparsely populated, we can represent it

in $\mathcal{O}(\frac{n}{\theta}\log\theta)$ bits (see Section 2.4.2). In total, the sampling requires $\frac{n}{\theta}\lceil\log n\rceil + \mathcal{O}(\frac{n}{\theta}\log\theta)$ bits. Using $\theta = \Omega(\log n)$, the sampling requires $\mathcal{O}(n)$ bits.

We denote by $t_{SA}$ the time required to access a value of the suffix array. For the text order sampling, $t_{SA} = \theta \cdot t_{LF} = \mathcal{O}(\log n \log \sigma)$.

In addition to the values in the SA, we will also make use of the values of the inverted suffix array (ISA in the following), denoted by $ISA = SA^{-1}$. $ISA[j]$ gives the position in the suffix array of the suffix starting at position $j$. The ISA is sampled in a similar fashion as the suffix array. Using $\theta = \Omega(\log n)$, the ISA sampling requires $\mathcal{O}(n)$ bits and can be accessed in $t_{ISA} = \theta \cdot t_{LF} = \mathcal{O}(\log n \log \sigma)$ time.

There are newer techniques that improve upon sampling both the SA and the ISA separately by making use of the correspondence between the two [25].

Using the values in the SA and the ISA, we can implement $\psi^k$ in a way that is faster than evaluating the $\psi$ function $k$ times successively for large values of $k$. We recall that $SA[\psi(i)] \equiv SA[i] + 1 \pmod{n}$ according to the definition of $\psi$. Hence, by applying $ISA = SA^{-1}$ to both sides, we get $\psi(i) \equiv ISA[SA[i] + 1] \pmod{n}$. We calculate $\psi^k(i) = ISA[SA[i]+k]$ (applying reduction modulo $n$ where necessary) in $t_{SA}+t_{ISA}$ time for any value of $k$. This is especially useful to extract the $d$-th character in the suffix starting at position $SA[i]$, which is achieved by calculating $F[\psi^d(i)]$. We will refer to the time required to calculate $\psi^k$ as $t_{text} = t_{SA} + t_{ISA}$.

### 2.7.5 Summary

To summarize, we can represent an FM-index of a text $T$ of size $|T| = n$ over an alphabet $\Sigma$ of size $|\Sigma| = \sigma$ using $\mathcal{O}(n \log \sigma)$ bits. For a pattern $P$ of size $|P| = m$, the FM-index solves *count* in $\mathcal{O}(m \log \sigma)$ time and *locate* in $t_{SA} = \mathcal{O}(\log n)$ time. The $\psi$ and $LF$ functions are calculated in $t_\psi = \mathcal{O}(\log \sigma)$ and $t_{LF} = \mathcal{O}(\log \sigma)$ time respectively.

Note that using a Huffman-shaped wavelet tree, the space consumption can be reduced to $\mathcal{O}(nH_0) + o(n)$ bits[5] [26].

## 2.8 Compressed Suffix Trees

Compressed suffix arrays are missing some of the functionality that suffix trees provide. In particular, inner nodes are not represented, so that the suffix link for inner nodes as well as the lowest common ancestor of two nodes cannot be calculated efficiently using compressed suffix arrays.

Compressed suffix trees address this shortcoming by extending a compressed suffix array with additional data structures representing the structure of the suffix tree. We use the term compressed suffix tree to refer to an abstract data type that supports the operations described as follows.

A *compressed suffix tree* (CST) of a text $T$ is a data structure that supports the following operations:

[5] $H_0$ is the zeroth order empirical entropy of the text.

- *root* returns the root of the suffix tree.
- *is-leaf*$(v)$ returns whether the node $v$ is a leaf.
- *parent*$(v)$ returns the parent of node $v$.
- *child*$(v, \alpha)$ returns the child $v.\alpha$ of node $v$.
- $first$-*child*$(v)$ returns the first child of node $v$.
- *next-sibling*$(v)$ returns the next sibling of node $v$.
- *letter*$(v, d)$ returns the character $L[d]$ of the path-label $L$ of node $v$.
- *slink*$(v)$ returns the suffix link of node $v$.
- $LCA(v, v')$ returns the lowest common ancestor of $v$ and $v'$.
- *depth*$(v)$ returns string depth of $v$, i.e. the length of the path label of $v$.

Sadakane [9] proposed the first data structure that implements all of the above operations efficiently using $\mathcal{O}(n)$ bits on top of the size of the compressed suffix array. In the following, we will describe this data structure.

The compressed suffix tree consists of three parts: A compressed suffix array, a compressed longest common prefix (LCP) array and a succinct tree representation of the suffix tree topology.

**Definition 13.** *For two strings $T$ and $T'$, let*

$$lcp(T, T') \coloneqq \operatorname*{argmax}_k \{0 \leq k \leq min(|T|, |T'|) \wedge T[0..k-1] = T'[0..k-1]\}$$

*The* LCP array*, denoted by $LCP$, is an array of size $n$ as follows:*

$$LCP[i] = \begin{cases} lcp(T[SA[i]..n-1], T[SA[i+1]..n-1]) & \text{if } 0 \leq i < n-1 \\ 0 & else \end{cases}$$

The LCP array stores the length of the longest common prefix of each two suffixes at consecutive positions in the suffix array.

**Theorem 1.** *[9] Given $i$ and $SA[i]$, the value $LCP[i]$ can be computed in constant time using a data structure of size $2n + o(n)$ bits.*

The data structure makes use of the fact that $LCP[\psi(i)] \geq LCP[i] - 1$. Therefore, the LCP values can be reordered so that they form an ascending sequence. The sequence $s_i = LCP[\psi^i(p)] + i$ for $i \in [0, n-1]$, where $p = ISA[0]$ is the longest suffix, is ascending and $s_{n-1} = n - 1$. Hence, the difference between two consecutive values in the sequence can be stored using a unary encoding, resulting in $2n - 2$ bits in total.

The succinct tree representation from Section 2.6 already provides solutions to *root*, *is-leaf*$(v)$, *parent*$(v)$, $first$-*child*$(v)$, *next-sibling*$(v)$ and $LCA(v, w)$. It remains to be shown how to solve *child*$(v, c)$, *letter*$(v, d)$, *slink*$(v)$ and *depth*$(v)$.

### Extracting Path-Labels

To solve $letter(v, d)$, we note that the path label of node $v$ is a prefix of the path label of all leaves in the subtree rooted at $v$. Hence, we only need to extract the $d$-th character of any of the suffixes represented by the leaves. The function $left\text{-}rank(v)$ denotes the leftmost of these leaves. We determine the $d$-th character as follows:

$$letter(v, d) = F[\psi^d(left\text{-}rank(v))]$$

We recall that $\psi^d$ can be calculated either in $d \cdot t_\psi = \mathcal{O}(d \log \sigma)$ time or in $t_{text} = \mathcal{O}(\log n \log \sigma)$ time (see Section 2.7.4).

### Calculating String Depths

To calculate the string depth of a node $v$, we distinguish between leaves and inner nodes.

The string depth of a leaf is the length of the suffix it represents. We can calculate it using the suffix array:

$$depth(v) = n - SA[left\text{-}rank(v)]$$

For an inner node $v$, we recall that all its children differ in the branching letter, which is the character at position $d = depth(v)$. Hence the path label of $v$ is the longest common prefix of any pair of children of $v$. We calculate its length by looking up in the LCP array the value of the rightmost leaf in the subtree rooted at the first child of $v$:

$$depth(v) = LCP[right\text{-}rank(first\text{-}child(v))]$$

This gives the length of the longest common prefix between the rightmost leaf of the first child of $v$ and the leftmost leaf of the second child of $v$.

### Navigating to Child Nodes

One way of navigating to a child $v.\alpha$ of a node $v$ is to check the branching letter of each child of $v$. We iterate over the children of $v$ using $first\text{-}child$ and $next\text{-}sibling$. For each child $v'$, we check the first character of the edge label from $v$ to $v'$, which is given by $letter(v', depth(v)+1)$. We require up to $\sigma$ steps, each of which requires $t_{text} + \mathcal{O}(1)$ time, so in total, the time is $\mathcal{O}(t_{text} \cdot \sigma) = \mathcal{O}(\sigma \cdot \log n \log \sigma)$.

The process can be sped up by performing a binary search. We still need to check all children using $first\text{-}child$ and $next\text{-}sibling$ first, which takes $\mathcal{O}(\sigma)$ time, but we then perform a binary search on these children, which requires only $\mathcal{O}(\log \sigma)$ accesses to the text. The total time is $\mathcal{O}(\sigma + t_{text} \log \sigma) = \mathcal{O}(\sigma + \log n \log^2 \sigma)$.

### Calculating the Suffix Link

The compressed suffix array provides the $\psi$ function, which solves the suffix link operation for leaves. To solve the suffix link for inner nodes, we apply the $\psi$ function to the leftmost

and rightmost leaf rooted in the subtree of $v$ and calculate the lowest common ancestor:

$$slink(v) = LCA(\psi(left\text{-}rank(v)), \psi(right\text{-}rank(v)))$$

For a proof of correctness, see the proof to Lemma 2, which is given in Section 3.1.1.

Also note that we can calculate $slink^k$ in $\mathcal{O}(t_{text})$ time by making use of $\psi^k$ on the CSA.

# 3 Fully-Compressed Suffix Trees

Although compressed suffix trees are already very small, they still require $\mathcal{O}(n)$ bits on top of the space of the CSA. Fully-compressed suffix trees, as proposed by Russo et al. [10], break the linear space barrier by storing only a sampling of the nodes from the original suffix tree. The sampled nodes are chosen in such a way that the information about the non-sampled nodes can be restored. This process of restoring requires additional computational overhead compared to traditional CST representations. The overhead is typically polylogarithmic in the input size.

In this section, we present the original proposal by Russo et al. [11], a variant proposed by Navarro and Russo [13] that uses a sparser sampling of the string depth and a third variant that uses a binary tree to provide faster pattern matching capability.

## 3.1 Original Proposal

### 3.1.1 Basics

We start by giving the definition of a $\delta$-sampled tree, which is the primary component of the fully-compressed suffix tree representation.

**Definition 14.** *[11] A* $\delta$-sampled tree $S$ *of a suffix tree* $\mathcal{T}$ *with* $t$ *nodes is formed by choosing* $s = \mathcal{O}(t/\delta)$ *nodes of* $\mathcal{T}$ *so that, for each node* $v$ *of* $\mathcal{T}$*, there is an* $i < \delta$ *such that node* $slink^i(v)$ *is sampled.*

This means that for each node $v$, there is at least one sampled node in the sequence $v$, $slink(v)$, $slink(slink(v))$, ..., $slink^{\delta-1}(v)$. Since $slink(root)$ is not defined, the root of the tree is always sampled.

Note that because there are at most $t \leq 2n$ nodes in the suffix tree, the number of sampled nodes is also $\mathcal{O}(n/\delta)$.

We achieve a $\delta$-sampling by choosing the root and every node $v$ that satisfies the following two conditions[6]:

(i) $depth(v) \equiv 0 \pmod{\delta/2}$

(ii) There is a node $v' \in S$ such that $v = slink^{\delta/2}(v')$

[6] In the following, we assume that $\delta$ is a multiple of two. We note that the sampling works for any value of $\delta \geq 2$ by using $\lfloor \delta/2 \rfloor$ instead of $\delta/2$.

We now have a subset of nodes of $\mathcal{T}$, but no edges. To form a tree out of these nodes, we have to assign each node a parent in $S$. The obvious choice is to construct the tree in such a way that the parent of each node in $S$ (except the root) is its lowest ancestor in $\mathcal{T}$. This means that for two sampled nodes $v_1$ and $v_2$, $v_1$ is an ancestor of $v_2$ in the sampled tree iff $v_1$ is an ancestor of $v_2$ in $\mathcal{T}$. Figure 11 shows an example of a suffix tree and its $\delta$-sampled tree for $\delta = 4$.

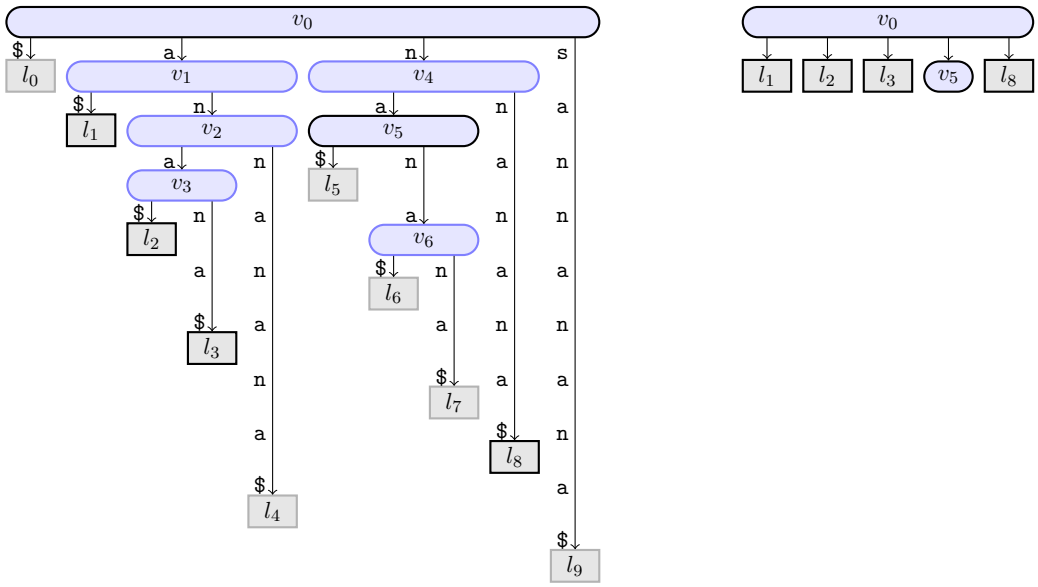


Figure 11: The suffix tree of the string 'sannanana$' (left) and its 4-sampled tree (right). The nodes of the suffix tree which are in the 4-sampled tree are highlighted (black border).

We reproduce the proof from [11] that the above construction rule results in a $\delta$-sampled tree.

*Proof.* For each node $v$ with $d = \mathit{depth}(v)$, we distinguish two cases. If $d < \delta$, then $\mathit{slink}^d = \mathit{root}$ is sampled. If $d \geq \delta$, then there are exactly two values of $i \in [0, \delta - 1]$ so that $\mathit{depth}(\mathit{slink}^i(v)) = \mathit{depth}(v) - i \equiv 0 \pmod{\delta/2}$. For the largest of those $i$ values, $\mathit{slink}^i(v)$ is sampled, since the second sampling condition holds as well. Also, for each node $v \neq \mathit{root}$ that is sampled, there are $\delta/2 - 1$ nodes $v_i$ so that $\mathit{slink}^i(v_i) = v$ for a value of $i \in [1, \delta/2 - 1]$. Those $v_i$ are not sampled because $\mathit{depth}(v_i) \not\equiv 0 \pmod{\delta/2}$. So there are $s \leq 1 + t/(\delta/2) = \mathcal{O}(t/\delta) = \mathcal{O}(n/\delta)$ sampled nodes. □

We represent the sampled tree using its balanced parentheses sequence $S$ (see Section 2.6). Nodes in the sampled tree are identified by the position of their opening parenthesis in the BPS of the sampled tree. Since not all nodes in the suffix tree are represented in the sampled tree, we will represent each node using its interval in the CSA. For example, the inner node $v_4$ in the suffix tree is represented by the interval $[5, 8]$, while leaf $l_3$ is represented by $[3, 3]$. Note that since the suffix tree is compact, each node is uniquely identified by an interval in the CSA. In the following, we will refer to a node and its interval in the CSA indifferently.

We will now show how to map the nodes of the sampled tree to their corresponding interval in the CSA.

**Definition 15.** *[11] Let $\mathcal{T}$ be a suffix tree of a text of length $n$ and let $S$ be the balanced parentheses representation of a $\delta$-sampled tree with $s$ nodes of $\mathcal{T}$. The* leaf mapping $B$ *is a bit vector of size $n + 2s$ containing $2s$ ones corresponding to the parentheses in $S$ and $n$ zeros corresponding to the leaves in the suffix tree. They are interleaved in such a way that iff a leaf $v$ is in the subtree rooted at a sampled node $u$, its corresponding zero in $B$ is located between the ones corresponding to the opening and closing parentheses of $u$.*

Since the leaf mapping contains only $2s$ ones, we can represent it as a sparse bit vector (see Section 2.4.2) using $\mathcal{O}(s \log \frac{n}{s}) = \mathcal{O}(\frac{n}{\delta} \log \delta)$ bits and support *rank* and *select* queries.

```
T: ((0)((1)(((2)(3))(4)))(((5)((6)(7)))(8))(9))

S: (    ( )  ( )( )       (           )( )    )
B: 1 0  101  101101  0    1 0   0  0  1101  0 1
```

Figure 12: Balanced parentheses representation of the suffix tree (T) as seen in Figure 11 and its sampled tree (S). The leaf mapping (B) of the sampled tree is also shown. The numbers in $T$ are just for illustration purposes and not part of the representation.

The leaf mapping can be used to determine the interval in the CSA that a sampled node represents. A sampled node $u \in [0, s-1]$ represents the interval $[v_l, v_r]$, where

$$\begin{aligned} v_l &= rank_0(select_1(u+1, B), B) \\ &= select_1(u+1, B) - u \end{aligned}$$

$$\begin{aligned} v_r &= rank_0(select_1(find\text{-}close(u)+1, B), B) - 1 \\ &= select_1(find\text{-}close(u)+1, B) - find\text{-}close(u) - 1 \end{aligned}$$

An operation that is essential for the implementation of all the following operations is the lowest sampled ancestor of a node $v$, denoted as $LSA$, that returns the lowest ancestor of $v$ that is in the sampled tree. Since the root is an ancestor of every node in $\mathcal{T}$ and sampled, such a node always exist.

For a leaf $v = [v, v]$, $LSA(v)$ can be computed using the leaf mapping. We define an auxiliary function $pred(v)$ which gives the position of the rightmost parenthesis in $S$ to the left of leaf $v$:

$$\begin{aligned} pred(v) &= rank_1(select_0(v+1, B), B) - 1 \\ &= select_0(v+1, B) - v - 1 \end{aligned}$$

We recall that an ancestor of a node $v$ is represented by a pair of parentheses enclosing $v$. The tightest such enclosing pair of parentheses is the lowest ancestor of node $v$. If the parenthesis in $S$ at position $pred(v)$ is an opening parenthesis, it encloses $v$ and certainly belongs to the tightest pair of enclosing parentheses. It thus represents the lowest sampled ancestor of $v$. If the parenthesis in $S$ at position $pred(v)$ is a closing parenthesis, it does not enclose $v$. Neither do any siblings of it, since they are either left or right of $v$. The pair of parentheses enclosing $pred(v)$ also encloses $v$, since its opening parenthesis is left of $pred(v)$ (it encloses it) and the closing parenthesis is right of $pred(v)$ and therefore also

$v$. It thus represents the lowest sampled ancestor of $v$. The following computation rule formalizes this:

$$LSA(v) = \begin{cases} pred(v) & \text{if } S[pred(v)] = \text{'('} \\ enclose_S(find\text{-}open_S(pred(v))) & \text{else} \end{cases}$$

For example, when evaluating $LSA(l_4)$, we find at position $pred(l_4) = 6$ in $S$ the closing parenthesis of $l_3$. Hence, neither $l_3$ nor any of its siblings is an ancestor of $l_4$. Therefore the lowest sampled ancestor of $l_4$ is the parent of $l_3$ in the sampled tree i.e. $LSA(v_4) = root$.

We will now show an essential property of the $LSA$ operation, namely the fact that calculating the lowest common ancestor in the sampled tree is essentially the same as calculating the lowest common ancestor in the original suffix tree with regard to $LSA$, or in other words: $LSA$ is a homomorphism from the structure of $LCA$ in $\mathcal{T}$ to the structure of $LCA$ in $S$. We formalize this in the following lemma:

**Lemma 1.** *[11] For a $\delta$-sampled tree $S$ of a suffix tree $\mathcal{T}$ and two nodes $v, v' \in \mathcal{T}$, the following holds:*

$$LSA(LCA(v, v')) = LCA_S(LSA(v), LSA(v'))$$

For a proof of this, see [11].

Using the $LSA$ operation for leaves and Lemma 1 we are now able to calculate $LSA$ for inner nodes and therefore any node in $\mathcal{T}$. It is easy to see that $LCA(v_l, v_r) = v$ for any node $v = [v_l, v_r]$ since the suffix tree is compact. Therefore, using Lemma 1:

$$LSA([v_l, v_r]) = LSA(LCA(v_l, v_r)) = LCA_S(LSA(v_l), LSA(v_r))$$

We introduce another Lemma that is essential for navigating the suffix tree using a $\delta$-sampled tree:

**Lemma 2.** *[11] For two nodes $v, v' \in \mathcal{T}$, such that $LCA(v, v') \neq root$, the following holds:*

$$slink(LCA(v, v')) = LCA(slink(v), slink(v'))$$

We reproduce the proof from [11]:

*Proof.* We recall that each node is uniquely determined by its path label. If $v = v'$, the proposition is trivially true. Otherwise, we can assume the path labels of $v$ and $v'$ are $\alpha.X.\beta.Y$ and $\alpha.X.\beta'.Y'$ respectively with $\alpha, \beta, \beta' \in \Sigma$ and $\beta \neq \beta'$. According to the definition of $LCA$ and $slink$, $LCA(v, v') = \alpha.X$ and $slink(LCA(v, v')) = X$. On the other hand, $slink(v) = X.\beta.Y$ and $slink(v') = X.\beta'.Y'$, so that $LCA(slink(v), slink(v')) = X$. Hence, both sides of the equation evaluate to the node with path label $X$. □

### 3.1.2 Navigation

We will now describe how to use the sampled tree to navigate the original suffix tree. We start with the calculation of the string depth. The intuition is that since for each node $v$ in the suffix tree there is a node $slink^i(v)$ that is sampled, we find such a node and deduce from its depth the depth of $v$ itself. Therefore, we will store the depth of each sampled node explicitly using a vector of $s = |S|$ integers indexed by the pre-order of each node $u \in S$. The string depth of any node is at most $n$. Since the string depth of sampled nodes is a multiple of $\delta/2$, it is sufficient to store $\frac{depth(u)}{\delta/2}$ and reconstruct $depth(u)$ by multiplying the stored value by $\delta/2$ . We can therefore store the depth of all sampled nodes using $s \log \frac{n}{\delta/2} = \mathcal{O}(\frac{n}{\delta} \log n)$ bits.

The following lemma is the main lemma for the navigation using the sampled tree.

**Lemma 3.** *[11] For two nodes $v, v' \in \mathcal{T}$, such that $slink^r(LCA(v, v')) = root$ and $m = min(\delta, r + 1)$, the following holds:*

$$depth(LCA(v, v')) = max_{0 \leq i < m}\{i + depth(LCA_S(LSA(slink^i(v)), LSA(slink^i(v'))))\}$$

*Proof.*

$$\begin{aligned}
depth(w) &\geq depth(LSA(w)) && (3.1)\\
depth(slink^i(LCA(v, v'))) &\geq depth(LSA(slink^i(LCA(v, v')))) && (3.2)\\
depth(LCA(v, v')) &\geq i + depth(LSA(slink^i(LCA(v, v')))) && (3.3)\\
&\geq i + depth(LSA(LCA(slink^i(v), slink^i(v')))) && (3.4)\\
&\geq i + depth(LCA_S(LSA(slink^i(v)), LSA(slink^i(v')))) && (3.5)
\end{aligned}$$

We start with Equation (3.1), which follows from the observation that for any node $w$, its depth is greater or equal than the depth of any of its ancestors. Note that the inequality becomes an equality if $w$ is sampled, since $w = LSA(w)$ iff $w$ is sampled. We get Equation (3.2) by substituting $slink^i(LCA(v, v'))$ for $w$. Equation (3.3) follows directly from the definition of $slink$. Finally, we apply Lemma 2 and 1 to get Equation (3.4) and (3.5) respectively. According to the definition of the $\delta$-sampled tree, there is a value of $i < \delta$ such that $w = slink^i(LCA(v, v'))$ is sampled, in which case the inequality becomes an equality, as mentioned in the beginning of the proof. Therefore, the depth is the maximum as defined in the proposition. □

#### String Depth

Lemma 3 allows us to compute the depth of any node $v = [v_l, v_r] = LCA(v_l, v_r)$. We successively compute the series of nodes $w_i = LCA_S(LSA(slink^i(v)), LSA(slink^i(v')))$ for all values of $0 \leq i < m$ or until $slink^i(LCA(v, v')) = LCA(slink^i(v), slink^i(v')) = root$. This can be checked efficiently by checking if $slink^i(v)$ and $slink^i(v')$ start with the same character.

Although we cannot yet evaluate $slink$ for inner nodes, we recall that $slink(v) = \psi(v)$ for leaves of the suffix tree. Since $v_l$ and $v_r$ are leaves, we can calculate $slink^i(v_l)$ and

$slink^i(v_r)$ using the $\psi$-function of the CSA. The $LSA$ operation is solved using the leaf mapping and $LCA_S$ is solved using the BPS of the sampled tree.

We can also calculate the string depth of the LCA of two inner nodes by using the following lemma:

**Lemma 4.** *[11] For two nodes $v, v' \in \mathcal{T}$ with $v = [v_l, v_r]$ and $v' = [v'_l, v'_r]$, the following holds:*

$$LCA(v, v') = LCA(\min(v_l, v'_l), \max(v_r, v'_r))$$

For a proof of this, see [11].

Using the fully-compressed suffix tree, we can solve *depth* in $\mathcal{O}(\delta \cdot t_\psi) = \mathcal{O}(\delta \log \sigma)$ time.

### LCA

As a by-product of the calculation of $depth(LCA(v, v'))$ using Lemma 3, we get one or two values of $i$ so that $slink^i(LCA(v, v'))$ is sampled. These values can easily be identified, as these are the values for which $i + depth(w_i)$ is maximal. The sampled nodes and their corresponding values of $i$ are at least as important as the resulting string depth itself, as they can be used to solve the $LCA$ operation.

We know from the definition of $LCA$ that $v[0..d-1] = v'[0..d-1] = LCA(v, v')$ for $d = depth(LCA(v, v'))$. Therefore, we can calculate $LCA(v, v')$ by using backward search:

**Lemma 5.** *[11] For two nodes $v, v' \in \mathcal{T}$ and $i \leq depth(LCA(v, v'))$, the following holds:*

$$LCA(v, v') = LF(v[0..i-1], slink^i(LCA(v, v')))$$

The total time to solve $LCA$ is $\mathcal{O}(\delta \cdot (t_\psi + t_{LF})) = \mathcal{O}(\delta \log \sigma)$.

### Suffix Link

The suffix link of a node $v = [v_l, v_r]$ is calculated using Lemma 2. Again, we solve *slink* for leaves using the $\psi$ function of the CSA. $LCA$ is then calculated using Lemma 5. The time required to solve *slink* is $\mathcal{O}(\delta \cdot (t_\psi + t_{LF})) = \mathcal{O}(\delta \log \sigma)$.

### Parent

Using Lemma 5 we can also compute the parent of a node. Note that since the suffix tree is compact, every inner node has at least two children. Hence, the parent of a node $v$ covers at least $v$ and at least either one of the leftmost leaf right of $v$, or the rightmost leaf left of $v$ (if either of them exists). To find the parent of a node $v = [v_l, v_r]$, we calculate both $LCA(v, [v_l + 1, v_l + 1])$ and $LCA(v, [v_r - 1, v_r - 1])$, which are both ancestors of $v$. In case these are different nodes, the lower one is the parent of $v$.

The time to calculate *parent* is $\mathcal{O}(\delta \cdot (t_\psi + t_{LF})) = \mathcal{O}(\delta \log \sigma)$.

#### Child

The last remaining operation is the *child* operation. It can be implemented using binary search over the CSA. The child $v.\alpha$ (which we recall is short for $v.\alpha.X$) of $v$ is the subinterval $v.\alpha = [v.\alpha_l, v.\alpha_r]$ of $v = [v_l, v_r]$ so that the branching letter of $v$ is $\alpha$. To find it, we use the forward search approach i.e. we compute the position $d = \mathit{depth}(v)$ of the branching letter and determine the left and right boundary of $v.\alpha$ using binary search.

We need $\mathcal{O}(\delta \cdot (t_\psi + t_{LF}))$ time to find the string depth and $t_{text} \cdot \lceil \log n \rceil$ to perform the binary search. The total required time to solve *child* is $\mathcal{O}(\delta \cdot (t_\psi + t_{LF}) + t_{text} \cdot \log n) = \mathcal{O}((\delta + \log^2 n) \log \sigma)$. The operations *first-child* and *next-sibling* are also solved using the *child* operation.

### 3.1.3 Construction

Using a compressed suffix tree of a text $T$, we can efficiently construct the fully-compressed suffix tree in two tree traversals. In the first tree traversal we determine the set of nodes for the $\delta$-sampled tree. In the second traversal we create the required data structures to represent the fully-compressed suffix tree.

During the construction the set of sampled nodes is represented by a bit vector $X$. The bit $X[\mathit{pre\text{-}order}(v)]$ will determine whether $v$ will be sampled. We traverse the CST and for every node $v \neq \mathit{root}$ with $\mathit{depth}(v) \equiv 0 \pmod{\delta/2}$, we determine $v' = \mathit{slink}^{\delta/2}(v)$ and set the bit $X[\mathit{pre\text{-}order}(v')]$ to one.

In the second traversal, we create the BPS of the sampled tree by copying the parenthesis of each node $v$ with $X[v] = 1$. The leaf mapping and the depth sampling are constructed accordingly.

It is sufficient to sample only inner nodes that meet the sampling conditions. Lemma 3 still holds if no leaves are sampled, since $LCA(v, v')$ is a leaf only if $v = v'$ and both $v$ and $v'$ are leaves. In this case, we can calculate $LCA(v, v') = v = v'$ and $\mathit{depth}(LCA(v, v')) = n - SA[v]$ without making use of the sampled tree.

### 3.1.4 Summary

We review the components of the fully-compressed suffix tree as well as their space requirements. The $\delta$-sampled tree is stored in $\mathcal{O}(\frac{n}{\delta})$ bits, the leaf mapping in $\mathcal{O}(\frac{n}{\delta} \log \delta)$ bits and the depth sampling in $\mathcal{O}(\frac{n}{\delta} \log \frac{n}{\delta})$ bits. In total, the fully-compressed suffix tree requires $\mathcal{O}(\frac{n}{\delta} \log \frac{n}{\delta})$ bits on top of the space of the CSA.

The fully-compressed suffix tree solves the operations $LCA$, *slink* and *parent* in $\mathcal{O}(\delta(t_\psi + t_{LF}))$ time, *depth* in $\mathcal{O}(\delta \cdot t_\psi)$ and *child* in $\mathcal{O}(\delta \cdot t_\psi + t_{text} \cdot \log n)$ time.

By choosing $\delta = \Omega(\log n \log \log n)$, the fully-compressed suffix tree can be stored in $\mathcal{O}(\frac{n}{\log \log n}) = o(n)$ bits on top of the CSA. The time for the operations $LCA$, *slink*, *parent* and *depth* is $\mathcal{O}(\log n \log \log n \log \sigma)$. The *child* operation is solved in $\mathcal{O}(\log^2 n \log \sigma)$ time[7].

[7] We assume $t_{text} = \mathcal{O}(\log n \log \sigma)$, as provided by the CSA described in Section 2.7.

## 3.2 Fully-Compressed Suffix Trees with Sparse Depth Sampling

The asymptotic space complexity of the fully-compressed suffix tree is bound by the size of the depth sampling, which requires $\mathcal{O}(\frac{n}{\delta} \log \frac{n}{\delta})$ bits. Navarro and Russo [13] proposed a way of reducing the size of the depth sampling by storing the string depth only for a subset of the sampled nodes, reducing the asymptotic complexity of the fully-compressed suffix tree to $\mathcal{O}(\frac{n}{\delta} \log \delta)$. This allows using a smaller value of $\delta$ to improve the performance of operations that do not require access to the depth sampling, while retaining the same space requirements.

### 3.2.1 Lowest Common Ancestor without Depth Sampling

In Section 3.1, we used Lemma 3 to find a sampled node $slink^i(v)$ for a node $v$ using the depth sampling. Navarro and Russo show that this can also be achieved without using the depth sampling.

**Lemma 6.** *[13] Let $v, v'$ be nodes such that $slink^r(LCA(v, v')) = root$. Then there is an $i$ with $0 \leq i < d = min(\delta, r + 1)$ such that*

$$LF(v[0..i-1], LCA(LSA(slink^i(v)), LSA(slink^i(v')))) = LCA(v, v')$$

We reproduce the proof from [13].

*Proof.* We consider the sequence of nodes $u_i = slink^i(LCA(v, v'))$, which is not computed. According to Lemma 2, $u_i$ can also be expressed as $u_i = LCA(slink^i(v), slink^i(v'))$. Comparing it with the sequence of nodes $w_i = LCA(LSA(slink^i(v)), LSA(slink^i(v'))) = LSA(LCA(slink^i(v), slink^i(v')))$, which is used in the lemma, shows that $w_i$ is an ancestor of $u_i$ for any $i$ in the interval. According to Lemma 3, there is an $i$ in the interval such that $depth(LCA(v, v')) = i + depth(w_i) = i + depth(u_i)$, which means that $w_i = u_i$ for an $i$ in the interval. Therefore, $LF(v[0..i-1], w_i) = LF(v[0..1-1], u_i) = LCA(v, v')$ according to the definition of $LF$. □

We use Lemma 6 to calculate $LCA(v, v')$. We compute the values of $w_i$ for all values of $i$ in the interval. Then, we calculate another sequence of nodes $w'_i$ with $w'_{d-1} = w_{d-1}$ and for $0 \leq j < d - 1$, $w'_i$ is $w_i$ or $LF(v[i], w'_{i+1})$, either of which is lower. Since $w_i$ is either an ancestor or a descendant of $LF(v[i], w'_{i+1})$, determining the lower one of them can be done only by looking at their intervals over the CSA, either of which is a subinterval of the other. According to the proof of Lemma 6, at least one node $w_i$ is equal to $u_i$. Since $w_i$ is an ancestor of $u_i$, we will choose $w'_i = LF(v[i], w'_{i+1}) = LF(v[i], u_{i+1}) = u_i$ in all following steps. Hence, $w'_0 = u_0 = LCA(v, v')$ is the resulting node. It is also important to note that we can identify the node $u_i$ that is sampled (so that $u_i = w_i$), as it has the smallest value of $i$ so that $LF(v[i], w'_{i+1})$ is an ancestor of $w_i$.

### 3.2.2 Sparse Depth Sampling

Now that we can calculate $LCA$ without using the depth sampling, we can use a sparser depth sampling while keeping the structure of the $\delta$-sampled tree unchanged.

We conceptually partition the set of nodes into bands according to their string depth. Each band contains all nodes with the same logarithm of their string depth, i.e. node $v$ belongs to the band of index $l_v = \lfloor 1 + \log depth(v) \rfloor$ (starting with index 1). The new sampling rule is to sample the depth of a node $v$ iff $depth(v)$ is a multiple of $\frac{\delta}{2} l_v$ and there is another node $v'$ so that $l_v = l_{v'}$ and $v = slink^{(\delta/2)l_v}(v')$.

The sampling rule guarantees that in each band $l$ at most one out of $\frac{\delta}{2} l$ nodes is sampled. Since we can represent the string depth of a node $v$ in band $l$ using $\mathcal{O}(\log depth(v)) = \mathcal{O}(l)$ bits, the sampling requires only $\mathcal{O}(\frac{1}{\delta})$ bits per node in the suffix tree. The total space for the depth sampling is $\mathcal{O}(\frac{n}{\delta})$.

We also need a bit vector indicating for each node $v \in S$ whether $v$ is sampled or not. We can implement this vector as a sparse bit vector (see Section 2.4.2) using $\mathcal{O}(n/\delta)$ bits.

To find the string depth of a node $v$ we need to follow *slink* at most $\mathcal{O}(\delta \log depth(v))$ times [13], resulting in a runtime of $\mathcal{O}(\delta \log depth(v) \cdot t_\psi)$.

The *slink* and *parent* and operations are implemented based on the $LCA$ operation just as in Section 3.1.

### 3.2.3 Summary

The sparse depth sampling can be stored using $\mathcal{O}(\frac{n}{\delta})$ bits. The $\delta$-sampled tree still requires $\mathcal{O}(\frac{n}{\delta})$ bits, the leaf mapping $\mathcal{O}(\frac{n}{\delta} \log \delta)$ bits. In total, the fully-compressed suffix tree with sparse depth sampling requires $\mathcal{O}(\frac{n}{\delta} \log \delta)$ bits on top of the space of the CSA.

The fully-compressed suffix tree with sparse depth sampling solves the operations $LCA$, *slink* and *parent* in $\mathcal{O}(\delta(t_\psi + t_{LF}))$ time, $depth(v)$ in $\mathcal{O}(\delta(t_\psi \log depth(v) + t_{LF}))$ time and *child* in $\mathcal{O}(\delta(t_\psi \log depth(v) + t_{LF}) + t_{text} \cdot \log n)$ time.

By choosing $\delta = \Omega(\log \log n)$, the fully-compressed suffix tree with sparse depth sampling can be stored in $\mathcal{O}(\frac{n \log \log \log n}{\log \log n}) = o(n)$ bits on top of the CSA. The time for the operations $LCA$, *slink* and *parent* is $\mathcal{O}(\log \log n \log \sigma)$. Assuming $depth(v) = \mathcal{O}(\log n)$, *depth* is solved in $\mathcal{O}((\log \log n)^2 \log \sigma)$ time. The *child* operation is solved in $\mathcal{O}(\log^2 n \log \sigma)$ time[8].

## 3.3 Binary Fully-Compressed Suffix Trees

One advantage of suffix trees over suffix arrays is faster pattern matching. As each substring of the text is represented by a node in the suffix tree (or a point on an edge between two nodes), pattern matching is simply descending into the tree. With the original form of the fully-compressed suffix tree, pattern matching is no more efficient than using the CSA alone. In Section 3.1.2, we described how to implement the *child* operation using the compressed suffix array, making no use of the sampled tree. In this section, we will present a variant of the fully-compressed suffix tree that allows the computation of the *child* operation using the sampled tree, resulting in a better runtime both in theory and practice.

[8] Again, we assume $t_{text} = \mathcal{O}(\log n \log \sigma)$, as provided by the CSA described in Section 2.7.

### 3.3.1 Child Sampling

Our approach is to extend the $\delta$-sampling of the suffix tree (as described in Section 3.1) to include the nodes of the children of sampled nodes. However, as each node has up to $\sigma$ children, we end up with $\mathcal{O}(\sigma \cdot n/\delta)$ sampled nodes if we sample all the children of the originally sampled nodes. To preserve the upper bound of $\mathcal{O}(n/\delta)$ sampled nodes we include only some of these children. The following definition describes which nodes have to be sampled to allow faster pattern matching.

**Definition 16.** *A $\delta$-sampled tree with child sampling $S'$ of a suffix tree $\mathcal{T}$ with $t$ nodes is formed by choosing $s = \mathcal{O}(t/\delta)$ nodes of $\mathcal{T}$ so that, for each node $v$ of $\mathcal{T}$, there is an $i < \delta$ such that node $slink^i(v)$ is sampled and if $depth(v) \geq \delta$, then for every child $v.\alpha$ of $v$, $slink^i(v).\alpha$ is sampled.*

The requirement for $depth(v) \geq \delta$ is necessary so that the number of sampled nodes is guaranteed to be $\mathcal{O}(n/\delta)$. We recall that in the original $\delta$-sampled tree, the root is always sampled. In the child sampling, this corresponds to all the children of the root being sampled. While sampling one additional node (the root) is feasible, we cannot afford sampling all of its $\sigma$ children. In practice, the children of the root can be calculated very efficiently using backward search.

Note that since the subtree rooted at $slink^i(v')$ is at least as large as the subtree rooted at $v'$, $slink^i(v').\alpha$ is an ancestor of $slink^i(v'.\alpha)$. See Figure 13 for an illustration of this.

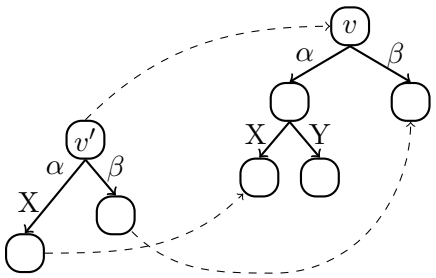


Figure 13: Both the nodes $v'$ and $v = slink^{\delta/2}(v')$ have a child with branching letter $\alpha$. $v.\alpha$ is an ancestor of $slink^{\delta/2}(v'.\alpha)$ and both may be different nodes, as this illustration shows. The dashed lines represent $slink^{\delta/2}$.

Every $\delta$-sampled tree with child sampling is also a $\delta$-sampled tree. Therefore, we sample the tree by choosing the nodes according to the construction rule described in Section 3.1.1 and another $\mathcal{O}(n/\delta)$ nodes for the child sampling.

For the child sampling we sample each child $v.\alpha$ of a sampled node $v \in S$ if there is a node $v'$ so that $slink^{\delta/2}(v') = v$ and $v'$ has a child $v'.\alpha$.

We prove that the above construction rule results in a $\delta$-sampled tree with child sampling.

*Proof.* For each node $v$ with $depth(v) \geq \delta/2$, there are two values of $i \in [0, \delta - 1]$ for which $depth(slink^i(v)) \equiv 0 \pmod{\delta/2}$. We will call the smaller one $i_0$. The larger one is $i_0 + \delta/2$. If $v$ has a child $v.\alpha$, then $slink^{i_0}(v)$ is sampled and has a child $slink^{i_0}(v).\alpha$. Thus $slink^{i_0+\delta/2}(v).\alpha$ is in the child sampling according to the construction rule.

We know that the number of nodes in the original sampling of the $\delta$-sampled tree is $\mathcal{O}(n/\delta)$. It remains to be shown that the number of nodes in the additional child sampling

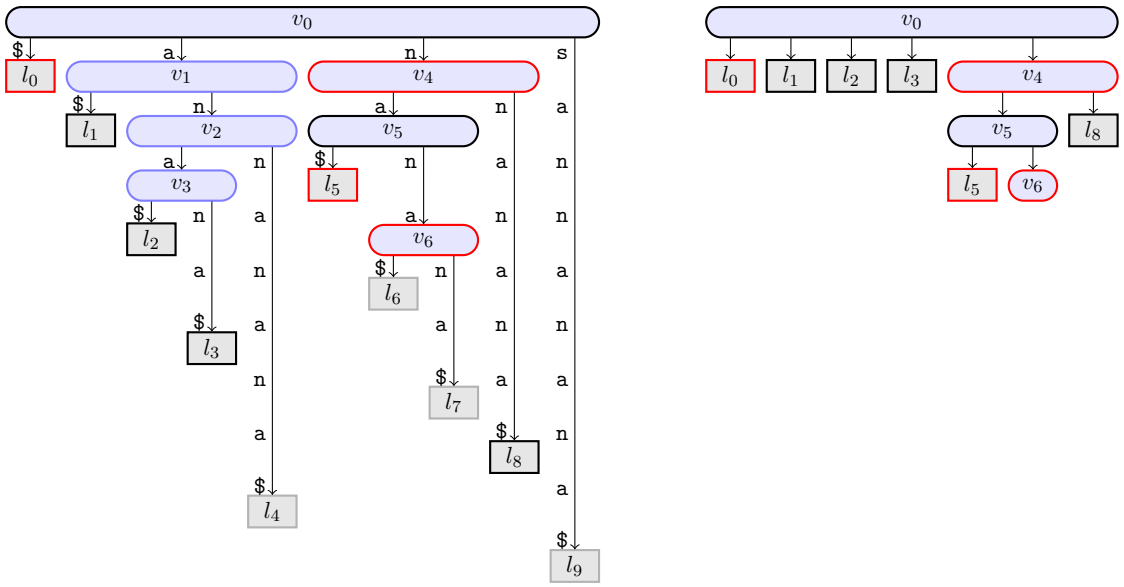


Figure 14: The suffix tree of the string 'sannanana$' (left) and its 4-sampled tree with child sampling (right). Highlighted are the nodes of the suffix tree which are in the 4-sampled tree (black border) and the nodes which are in the child sampling (red border).

is also $\mathcal{O}(n/\delta)$. We consider a node $v$ that is in the original sampling and a child $v.\alpha$ of $v$ that is in the child sampling. According to the sampling condition there is a node $v'$ so that $slink^{\delta/2}(v') = v$ and that has a child $v'.\alpha$. We consider the sequence of nodes $v_i = slink^i(v')$. Because $v'$ has a child $v'.\alpha$, every such $v_i$ also has a child $v_i.\alpha$. For any value of $i \in [1, \delta/2 - 1]$, the node $v_i$ is not in the original sampling because $depth(v_i) \not\equiv 0 \pmod{\delta/2}$. Therefore, $v_i.\alpha$ is not in the child sampling. So for each node in the child sampling there are $\delta/2 - 1$ nodes which are not in the child sampling and the number of nodes in the child sampling is $s' \leq 1 + t/(\delta/2) = \mathcal{O}(t/\delta) = \mathcal{O}(n/\delta)$. The total number of sampled nodes is $\mathcal{O}(n/\delta)$. □

All the nodes in the $\delta$-sampling and the child sampling are stored in a single sampled tree. Figure 14 shows an example of a $\delta$-sampled tree with child sampling. In the example, the nodes in the original and the child sampling are disjoint. Note that in general, there can be nodes that meet both sampling conditions.

A $\delta$-sampled tree with child sampling can be constructed from a compressed suffix tree. We traverse the suffix tree and find nodes $v'$ so that $depth(v') \equiv 0 \pmod{\delta/2}$. We then find the node $v = slink^{\delta/2}(v')$ which is in the $\delta$-sampling. To find the nodes in the child sampling we determine for each child $v'.\alpha$ of $v'$ the child $v.\alpha$, which is in the child sampling. $v.\alpha$ is computed using the *child* operation of the CST. Hence, we need to extract the character $\alpha$ from the text, which is rather inefficient.

We can eliminate this text access using an observation we made earlier in this section: If $v'.\alpha$ exists, $v.\alpha$ is an ancestor of $slink^{\delta/2}(v'.\alpha)$. Therefore, $u := LF(v'[0..\delta/2 - 1], v.\alpha)$ is an ancestor of $LF(v'[0..\delta/2 - 1], slink^{\delta/2}(v'.\alpha)) = v'.\alpha$. Since the string depth of $u$ is higher than that of $v'$ (there is an additional character $\alpha$ at the end), $u$ is a descendant of $v'$. Since $u$ is an ancestor of $v'.\alpha$ but a descendant of $v'$ and $u \neq v'$, we conclude that $u = v'.\alpha$. Note that $u = LF(v'[0..\delta/2 - 1], v.\alpha)$ may refer to an empty interval. This is

the case if $v'[0..\delta/2-1].v.\alpha = v'.\alpha$ is not a substring of the text and therefore not a suffix tree node.

This means that we can find each child $v.\alpha$ of $v$ and sample it if $u = LF(v'[0..\delta/2-1], v.\alpha)$ exists (i.e. does not result in an empty interval over the CSA) without the need to extract character $\alpha$.

Using the $\delta$-sampled tree with child sampling we can implement the *child* operation as follows. If $depth(v) < \delta$, we calculate $child(v, \alpha)$ using backward search. If $depth(v) \geq \delta$, there is a value of $i$ such that $v' = slink^i(v)$ and $slink^i(v).\alpha$ are sampled, which we find using Lemma 3. To find $v'.\alpha$, we iterate over all the children of $v'$ and check whether the corresponding branching letter is $\alpha$. Finally, we use backward search to find $child(v, \alpha) = LF(v[0..i-1], slink^i(v).\alpha)$.

The time to compute *child* using the $\delta$-sampled tree with child sampling is $2\delta(t_\psi + t_{LF}) + \sigma \cdot t_{text}$. This is poor for large alphabets, since iterating all the children is very expensive. Also, accessing the text by itself is very expensive, even for small alphabets.

### 3.3.2 Binary Suffix Trees

In order to avoid checking the branching letter for each child of a node, we propose using a variation of the suffix tree which contains the binary representation of the suffixes. In such a tree, every inner node has exactly two children with edge labels 0 and 1 respectively. Therefore, we can perform a blind search, i.e. find a path with a given label without the need to extract the text to determine the edge labels. This blind search is inspired by the blind trie[9] proposed in [27].

We denote by $bin(s)$ the binary representation of a string $s$. Note that we assume a compact alphabet i.e. the alphabet consists only of characters that occur in the text. For instance, to encode the string $s =$ 'nasa\$', we use the alphabet $\Sigma = \{\$, a, n, s\}$, so that $bin(s) =$ `10 01 11 01 00`. See Table 1 for the encoding of each character in the alphabet.

| Character | Binary encoding |
|---|---|
| \$ | 00 |
| a | 01 |
| n | 10 |
| s | 11 |

Table 1: The binary encoding of each character in the alphabet $\Sigma = \{\$, a, n, s\}$.

**Definition 17.** *The* binary suffix tree $\mathcal{B}$ *of a text* $T$ *is the compact trie of the binary representation of all the suffixes in* $T$*.*

Note that in general, the binary suffix tree of $T$ is different from the suffix tree of $bin(T)$, as the former contains only a subset of all suffixes of $bin(T)$. Figure 15 shows an example of a binary suffix tree.

The suffix tree $\mathcal{T}$ of a text $T$ and the binary suffix tree $\mathcal{B}$ of the same text are similar to each other. $\mathcal{B}$ has the same number of leaves and at least as many inner nodes as $\mathcal{T}$, but in contrary to $\mathcal{T}$, $\mathcal{B}$ is always a binary tree.

[9] The blind trie is called Patricia Trie in the original paper (see [27]) but is commonly referenced as blind trie.

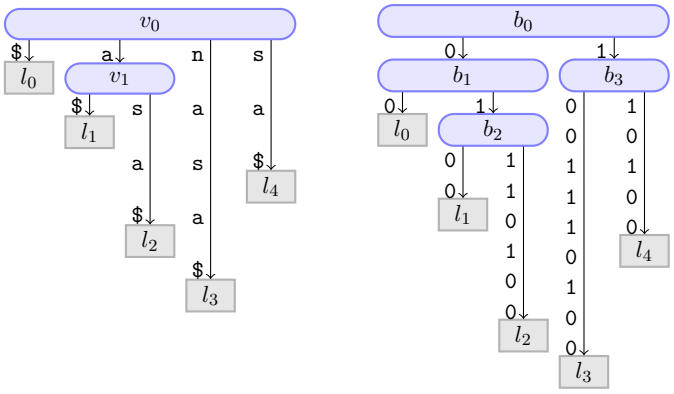


Figure 15: The suffix tree of the string 'nasa$' (left) and the corresponding binary suffix tree (right).

For each inner node $v$ in $\mathcal{T}$ with degree $m$, there are $m-1$ nodes in $\mathcal{B}$. They are identified by their edge labels, which are prefixed by $bin(v)$, but not by $bin(v')$ for any descendant $v' \neq v$ of $v$. For example, in Figure 15 the nodes corresponding to $v_0$ in $\mathcal{T}$ are $b_0$, $b_1$ and $b_3$ in $\mathcal{B}$, while $v_1$ corresponds only to $b_2$. We will now formalize this node correspondence.

For each node $v \in \mathcal{T}$ we denote by $C(v)$ all *corresponding nodes* $b \in \mathcal{B}$ and by $c(v)$ the *representative node* for $v$, which is the corresponding node of $v$ that has the smallest string depth.

$$C(v) := \{b \in \mathcal{B} \mid bin(v) \text{ is a prefix of } b \wedge \lfloor depth_{\mathcal{B}}(b)/\lceil \log \sigma \rceil \rfloor = depth_{\mathcal{T}}(v)\}$$
$$c(v) := \operatorname*{argmin}_{b \in C(v)} depth_{\mathcal{B}}(b)$$

A node $v$ and its representative $c(v)$ share common properties. For example, the leaves rooted in the subtree of $v$ are the same as in the subtree of $c(v)$. Therefore, they can be represented by the same interval over the CSA. In the following, we will refer to $v$ and $b(v)$ indifferently. On the other hand, we will refer to the nodes in $\mathcal{B}$ that are not a representative node of any node in $\mathcal{T}$ as *pseudo nodes*.

We will now explain how we can navigate from a node $v$ to a child $v.\alpha$ in the binary suffix tree using blind search. We start at node $b = c(v)$. In each step, we choose one of the children of $b$ according to the bit at position $depth_{\mathcal{B}}(b) \bmod \lceil \log \sigma \rceil$ of $bin(\alpha)$. We continue this process with the new node until we find a node that is not a pseudo node. The resulting node is $v.\alpha$, if it exists.

For example in Figure 15, assume we want to navigate from $v_0 = b_0$ to $v_0.a = v_1 = b_2$ using blind search. We recall that $bin(a) = 01$. Because $depth(b_0) \equiv 0 \pmod{\lceil \log \sigma \rceil}$, we look at the first bit of $bin(a)$, which is 0. Hence, we choose the left child of $b_0$, which is $b_1$. Since $b_1$ is a pseudo node, we continue the process with $b_1$. Since $depth(b_0) \equiv 1 \pmod{\lceil \log \sigma \rceil}$, we look at the second bit of $bin(a)$, which is 1. Hence, we choose the right child of $b_1$, which is $b_2$. Since $b_2$ is not a pseudo node, the result of the blind search is $b_2$.

$\mathcal{B}$ can be represented by a balanced parentheses sequence (see Section 2.6) using $2(2n - 1) = \mathcal{O}(n)$ bits. We can extend $\mathcal{B}$ to a fully-functional compressed suffix tree by adding a CSA and a compressed LCP array (see Section 2.8). However, we cannot afford spending $\mathcal{O}(n)$ bits of extra space for the fully-compressed suffix tree.

### 3.3.3 Sampled Binary Trees

To reduce the space requirements, we propose a sampled binary tree. The basic idea is to use the $\delta$-sampled tree with child sampling from Section 3.3.1 and add just enough nodes from the binary suffix tree so that each node in the resulting tree has at most two children and we can perform blind search in the sampled tree.

Note that the binary suffix tree may contain nodes with only one child. For example, consider an arbitrary sampling in which $b_0$ and $b_2$ are sampled in the binary suffix tree in Figure 15. In the resulting sampled tree, $b_0$ has only one child $b_2$.

There are multiple ways to add nodes from the binary suffix tree so that no node in the sampled binary tree has more than two children. For example, consider we want to sample $b_0$, the root, and the three leaves $l_1$, $l_2$ and $l_3$ in the binary suffix tree in Figure 15. Sampling only these nodes results in $b_0$ having three children in the resulting tree. By additionally sampling either $b_1$ or $b_2$, the resulting tree becomes a binary tree. If we choose to sample $b_1$, both $l_1$ and $l_2$ are in the right branch of $b_1$, which has the edge label 1. This means that both nodes have the same path label in the resulting sampled tree, which is a problem if we want to perform blind search. So instead, we choose to sample $b_2$, the lowest common ancestor of $l_1$ and $l_2$, which ensures that both nodes can be distinguished by the branching letter of the corresponding parent node.

**Definition 18.** *The* $\delta$-sampled binary tree $\mathcal{S}_{bin}$ *of a* $\delta$*-sampled tree with child sampling* $\mathcal{S}'$ *and a binary suffix tree* $\mathcal{B}$ *of the same text is formed by choosing the corresponding node* $c(v)$ *for every node* $v \in \mathcal{S}'$ *and* $\mathcal{O}(n/\delta)$ *additional nodes from* $\mathcal{B}$ *so that all nodes in* $\mathcal{S}_{bin}$ *have degree two or less and if the degree of any node* $v$ *is two, both children of* $v$ *in* $\mathcal{S}_{bin}$ *are in subtrees rooted at different children of* $v$ *in* $\mathcal{B}$*.*

Figure 16 shows an example of a $\delta$-sampled binary tree.

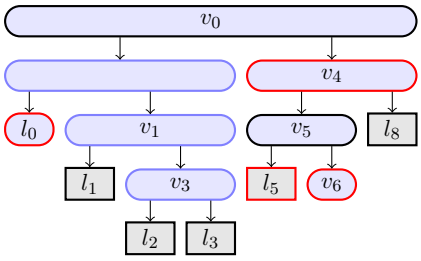


Figure 16: The 4-sampled binary tree of the string 'sannanana$'. The corresponding suffix tree is shown in Figure 14. The nodes of the suffix tree which are in the 4-sampled tree are shown with a black border, while the nodes which are in the child sampling are shown with a red border. Nodes that are added to compose the binary tree are shown with a blue border.

Algorithm 2 shows how to construct a $\delta$-sampled binary tree from a binary suffix tree and a $\delta$-sampled tree with child sampling. Note that this separation into different trees is only conceptual. In practice, only a single binary suffix tree and a bit vector, which marks the nodes that comprise the $\delta$-sampled tree with child sampling, are used. The binary suffix tree can be implemented in such a way that both the binary tree operations as well as the original suffix tree operations are supported.

**Algorithm 2:** Construction of a $\delta$-sampled binary tree.

```
Input: A binary suffix tree 𝓑 and a δ-sampled tree with child sampling 𝓢′ of the same
       text.
Output: A δ-sampled binary tree S_bin
1  Algorithm construct()
2      global stack: Stack
3      global S_bin ← {}
4      stack.push(0)
5      visit (root)
6      stack.pop()
7      return S_bin
8  Procedure visit(v)
9      stack.push(0)
10     foreach child w of v do
11         visit (w)
12     if v ∈ S′ ∨ stack.top = 2 then
13         S_bin ← S_bin ∪ {v}
14         stack.pop()
15         stack.top ← stack.top + 1
16     else if stack.top = 1 then
17         stack.pop()
18         stack.top ← stack.top + 1
19     else
20         stack.pop()
```

Note that since each node in $\mathcal{B}$ has only two children and each child increases the value in the stack by at most one, all the values in the stack in Algorithm 2 are either 0 or 1, except for the top of the stack, which can be 2. Therefore, we can implement the stack using only one bit per entry and an additional bit for the top value.

Before we prove the correctness of Algorithm 2, we introduce some additional terminology.

**Definition 19.** *Let $v'$ be a sampled node and $v \in \mathcal{B}$. We call $v'$ a* bare node *of $v$ if $v'$ is a descendant of $v$ and there is no sampled node $v'' \neq v'$ so that $v''$ is a descendant of $v$ and an ancestor of $v'$.*

In particular, if a node $v$ is sampled it is a bare node of itself and it is the only bare node of itself.

For example, in Figure 17, before the sampling step, node $b_0$ has three bare nodes: $b_2$, $b_5$ and $b_6$. After the step, $b_4$ is sampled and the bare nodes in $b_0$ are $b_2$ and $b_4$. Also note that the stack stores the current number of bare nodes in each node $v \in \mathcal{B}$ for which visitation has started and not yet finished.

**Lemma 7.** *After any node $v \in \mathcal{B}$ has been visited, $v$ has at most one bare node.*

*Proof.* Proof by structural induction. If $v$ is a leaf, the proposition is trivially true. If $v$ is an inner node, both its children have been visited, since we perform a post-order

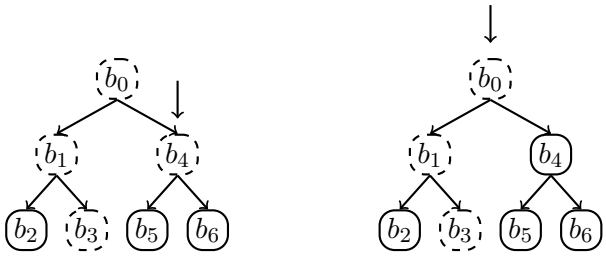


Figure 17: Part of the binary tree before (left) and after (right) a step in an exemplary execution of Algorithm 2. The nodes that are in the resulting sampled tree are have a solid border, the remaining nodes have a dashed border.

traversal of the tree. Therefore, both children have at most one bare node according to the premise. If they both have a bare node, $v$ is added to the sampled tree (Line 13) and $v$ itself is now the only bare node of $v$. □

**Lemma 8.** *Algorithm 2 returns a $\delta$-sampled binary tree.*

*Proof.* The nodes are visited in post-order, i.e. after every child of a node $v$ has been visited, the choice whether $v$ is added to $S_{bin}$ is made.

Since each node $v \in \mathcal{B}$ has at most two children and according to Lemma 7 both of them have at most one bare node after the algorithm finishes and all nodes have been visited at that point, the degree of $v$ in the sampled tree is at most two. Also, if the degree is two, both children are descendants of different children of $v$ in $\mathcal{B}$.

To prove that the number of additional nodes is $\mathcal{O}(n/\delta)$, we take a look at the state of the tree $S' \cup S_{bin}$ during the execution of the algorithm. When the algorithm finishes, this tree is equivalent to the final $\delta$-sampled binary tree. We call a node $v$ an *additional node* if $v \in S_{bin} \wedge v \notin S'$.

Let $s = |S'|$ be the number of nodes in the $\delta$-sampled tree with child sampling. Since each node (apart from the root) is the child of exactly one node, the sum of the degrees of all nodes in $S'$ is $s - 1$. Sampling a node $v \in S'$ does not modify the tree $S' \cup S_{bin}$, so we can ignore it. If there are two bare nodes in a node $v$ that is not in $S'$, $v$ is sampled as an additional node (Line 12) and $S' \cup S_{bin}$ is modified. After the modification, the parent $v'$ of $v$ in $S' \cup S_{bin}$ has two fewer children (which are now children of $v$) and $v$ becomes a child of $v'$. If $v'$ had degree $m$ before the modification, it has degree $m - 1$ afterwards. The new node $v$ has degree 2. The degree of $v$ is not changed in any later step, since all its children have been visited. The sum of the degree of all nodes which can still change decreases by one with each additional node. Hence, at most $s - 1$ additional nodes are added to $S' \cup S_{bin}$. □

We represent the $\delta$-sampled binary tree using a CSA and the following data structures:

- $S_{bin}$ The balanced parentheses representation (Section 2.6) of the $\delta$-sampled binary tree.
- $B_{bin}$ The leaf mapping for $S_{bin}$ represented by a sparse bit vector (Section 2.4.2).
- $D_{bin}$ The string depth of each node in $S_{bin} \pmod{\lceil \log \sigma \rceil}$.

- $M_{\mathcal{T}}$ A bit vector that stores for each node $v$ in $S_{bin}$ whether $v$ is a node in the suffix tree (1) or a pseudo node (0).
- $S$ The balanced parentheses representation (Section 2.6) of the original $\delta$-sampled tree.
- $M_S$ A bit vector storing for each parenthesis in $S_{bin}$ whether the parenthesis is also in S (1) or not (0).
- The string depth for each node in $S$.

An example of the representation is shown in Figure 18.

Storing the string depth for all nodes in $S_{bin}$ is not beneficial for the runtime of the navigation operations, since we still have to follow $\delta$ suffix links to be sure to find a sampled node. Hence, it is sufficient (and more space efficient) to store the string depth only for nodes in $S$. The bit vector $M_S$ is used to map positions in $S_{bin}$ to positions in $S$ and vice versa.

```
T_bin: (((0)((1)(((2)(3))(4))))((((5)((6)(7)))(8))(9)))

D_bin: 010  00   00  0          000  0        0
S_bin: ((( )(( ) (( )( ))    )) ((( )(      ))( ))    )
B_bin: 111011101 11011011 0  11 111011 0  0 111011 0  1
M_T:   101  11   11  1          111  1        1
S:     (     ( )  ( )( )         (           )( )     )
M_S:   100 001 1 01 11 10    00 010 00      011 10    1
```

Figure 18: The succinct representation of the sampled binary tree (bottom) seen in Figure 16 and its corresponding binary suffix tree (top). The numbers in $T_{bin}$ are just for illustration purposes and not part of the representation.

### 3.3.4 Navigation

#### Lowest Sampled Ancestor

To solve basic navigation operations using Lemma 3, we need to compute the lowest sampled ancestor in $S$ of any node $v$, since the string depth is only stored for nodes in $S$.

The calculation of $LSA_S$ for leaves is analog to the calculation in the original proposal (see Section 3.1) with a small difference: Since $B_{bin}$ maps the leaves in the suffix tree to positions in $S_{bin}$, we need an additional step to calculate $LSA_S$.

Again, we start by defining an auxiliary function $pred_{bin}(v)$, which gives the position of the last parenthesis in $S_{bin}$ preceding leaf $v$:

$$\begin{aligned} pred_{bin}(v) &= rank_1(select_0(v+1, B_{bin}), B_{bin}) - 1 \\ &= select_0(v+1, B_{bin}) - v - 1 \end{aligned}$$

In the next step, we calculate the position of the last parenthesis in $S$ preceding leaf $v$ by using $M_S$ to map the position of $pred_{bin}(v)$ to the corresponding position in $S$. If

$M_S[v] = 1$, the node represented by the parenthesis at position $pred_{bin}(v)$ in $S_{bin}$ is also in $S$. If $M_S[v] = 0$, we use the parenthesis in $S$ that is preceding the parenthesis at position $pred_{bin}(v)$ in $S_{bin}$:

$$pred_S(v) = \begin{cases} rank_1(pred_{bin}(v), M_S) & \text{if } M_S[pred_{bin}(v)] = 1\text{'('} \\ rank_1(pred_{bin}(v), M_S) - 1 & \text{else} \end{cases}$$

$pred_S(v)$ can also be formulated without the case distinction:

$$pred_S(v) = rank_1(pred_{bin}(v), M_S) + M_S[pred_{bin}(v)] - 1$$

Using $pred_S$, we can calculate $LSA_S$ in the same way as in the original proposal:

$$LSA_S(v) = \begin{cases} pred_S(v) & \text{if } S[pred_S(v)] = \text{'('} \\ enclose_S(find\text{-}open(pred_S(v))) & \text{else} \end{cases}$$

$LSA$ for inner nodes is solved using Lemma 1 as in the original proposal.

### Child Operation

To determine the child $v.\alpha$ of a node $v$, we need to find a node $v' = slink^i(v)$ so that $v'.\alpha$ is sampled. If $depth(v) \geq \delta$, there is at least one such node with $i < \delta$ according to the definition of the sampled binary tree and the node we are looking for is $v.\alpha = LF(v[0..i-1], v'.\alpha)$. If $depth(v) < \delta$, we calculate $v.\alpha$ via backward search.

We find the sampled node $v' = slink^i(v)$ using Lemma 3. To make use of this lemma, we require the depth of $slink^i(v)$. Therefore, it is necessary for the node to be sampled not only in $S_{bin}$ but also in $S$. There are at least one and at most two such nodes for $i < \delta$. Assume $slink^i(v)$ is sampled for $i_0$ and $i_0 + \delta/2$ and node $v$ has a child $v.\alpha$. Then $slink^{i_0}(v)$ has a child $slink^{i_0}(v).\alpha$ and $slink^{i_0+\delta/2}(v).\alpha$ is sampled according to the definition of the sampled binary tree. The same is valid for $slink^i(v).\alpha$ in case there is only one value of $i < \delta$ so that $slink^i(v)$ is sampled.

We find $v'.\alpha$ using blind search by traversing the subtree rooted at $v'$, at each step choosing the left or right node according to the bit in $\alpha$ at the position indicated by $D_{bin}$. We are done when we arrive at a node $v''$ that is not a pseudo node, which is determined by looking up $M_T[pre\text{-}order(v'')]$.

Finally, we need to check whether $v.\alpha[d] = \alpha$, where $d = depth(v)$. If this is not the case, $v.\alpha$ does not exist.

In the worst case, we need $\delta - 1$ applications of $\psi$ to find the sampled node $v'$ and its string depth, $\log \sigma$ steps to perform blind search to find $v'.\alpha$, $\delta - 1$ applications of $LF$ for the backward search and one text access for the final check. In total, the time complexity for $child$ is $\mathcal{O}(\delta(t_\psi + t_{LF}) + \log \sigma + t_{text})$.

### 3.3.5 Summary

The $\delta$-sampled tree is stored in $\mathcal{O}(\frac{n}{\delta})$ bits, the leaf mapping in $\mathcal{O}(\frac{n}{\delta}\log\delta)$ bits and the depth sampling in $\mathcal{O}(\frac{n}{\delta}\log\frac{n}{\delta})$ bits. We also store the binary depth ($D_{bin}$) using $\mathcal{O}(\frac{n}{\delta}\log\log\sigma) = \mathcal{O}(\frac{n}{\delta}\log\log n)$ bits[10]. The remaining data structures ($M_{\mathcal{T}}$, $S$, $M_S$) require $\mathcal{O}(\frac{n}{\delta})$ space. In total, the binary fully-compressed suffix tree requires $\mathcal{O}(\frac{n}{\delta}\log\frac{n}{\delta})$ bits on top of the space of the CSA.

The binary fully-compressed suffix tree solves the operations $LCA$, *slink* and *parent* in $\mathcal{O}(\delta(t_\psi + t_{LF}))$ time, *depth* in $\mathcal{O}(\delta \cdot t_\psi)$ and *child* in $\mathcal{O}(\delta(t_\psi + t_{LF}) + \log\sigma + t_{text})$ time.

By choosing $\delta = \Omega(\log n \log\log n)$, the fully-compressed suffix tree can be stored in $\mathcal{O}(\frac{n}{\log\log n}) = o(n)$ bits on top of the CSA. The time for the operations $LCA$, *slink*, *parent*, *depth* and *child* is $\mathcal{O}(\log n \log\log n \log\sigma)$.

Assuming $\sigma = \mathcal{O}(polylog(n))$, we can use the FM-index of Ferragina et al. [28] as our underlying CSA and $\delta = \Omega(\log n \log\log n)$. The FCST requires $\mathcal{O}(\frac{n}{\log\log n}) = o(n)$ bits on top of the CSA and solves $LCA$, *slink*, *parent*, *depth* and *child* in $\mathcal{O}(\log n \log\log n)$ time. This improves the theoretical result of Russo et al. [11].

# 4 Experimental Evaluation

In this section, we will conduct an experimental evaluation of the data structures we implemented. We start off by describing our implementation and the experimental setup. We then examine the size of the indexes our implementation produces, the performance of individual queries and finally the time required for index construction.

## 4.1 Implementation

Our implementation is based on the Succinct Data Structure Library (SDSL) [12]. The SDSL is a C++ library that provides a variety of data structures, including compressed suffix trees, compressed suffix arrays, wavelet trees, bit vectors and integer vectors. In particular, the succinct and compact data structures we introduced in Section 2 are implemented in the SDSL. All SDSL data structures are implemented as template classes, allowing easy composition and configuration.

Our implementation of the fully-compressed suffix tree can be parametrized with different $\delta$s, compressed suffix arrays, balanced parentheses sequences, bit vectors for the leaf mapping and integer vectors for the depth sampling. Input text over byte as well as integer sequences can be processed by our implementation.

[10] By using a compact alphabet (an alphabet that contains only characters that occur in the text), we can ensure that $\sigma \leq n$.

## 4.2 Experimental Setup

We will compare implementations of the following three variants of fully-compressed suffix trees:

- `cst_fully` The original fully-compressed suffix tree (see Section 3.1).
- `cst_fully_sds` The fully-compressed suffix tree with sparse depth sampling (see Section 3.2).
- `cst_fully_blind` The binary fully-compressed suffix tree that uses blind search (see Section 3.3).

For reference, we will also show results for the following data structures:

- `cst_russo` The fully-compressed suffix tree implementation by Russo et al.[11].
- `cst_sada` The compressed suffix tree by Sadakane (see Section 2.8).
- `csa_wt` The compressed suffix array based on the FM-index (see Section 2.7). The exact type of the data structure in the SDSL is `csa_wt<wt_huff<rrr_vector<63> >, 32, 32, text_order_sa_sampling<>, text_order_isa_sampling_support<> >`. Both the fully-compressed suffix tree and the compressed suffix tree use this data structure as their underlying CSA.

The evaluation was performed on an Intel Xeon E5-4640 CPU @ 2.40 GHz. The code was compiled using g++ 4.8.1 and compiler flags `-O3 -ffast-math -funroll-loops -msse4.2`.

To evaluate the data structures, we use the text collection from the Pizza&Chili corpus[11]. If not stated otherwise, we use a prefix of length 100MB for each text. If the file is smaller than 100MB we used the complete text.

The text collection contains the following files[12]:

- *Sources (source program code)* This file is formed by C/Java source code obtained by concatenating all the .c, .h, .C and .java files of the linux-2.6.11.6 and gcc-4.0.0 distributions.
- *Pitches (MIDI pitch values)* This file is a sequence of pitch values (bytes in 0-127, plus a few extra special values) obtained from a myriad of MIDI files freely available on Internet. The MIDI files were processed using semex 1.29 tool by Kjell Lemstrom, so as to convert them to IRP format. This is a human-readable tuple format, where the 5th column is the pitch value. Then the pitch values were coded in one byte each and concatenated.
- *Proteins (protein sequences)* This file is a sequence of newline-separated protein sequences (without descriptions, just the bare proteins) obtained from the Swissprot database. Each of the 20 amino acids is coded as one uppercase letter.
- *DNA (gene DNA sequences)* This file is a sequence of newline-separated gene DNA sequences (without descriptions, just the bare DNA code) obtained from files 01hgp10 to 21hgp10, plus 0xhgp10 and 0yhgp10, from Gutenberg Project. Each of the 4 bases

[11] `http://pizzachili.dcc.uchile.cl/texts.html`
[12] The description is taken from `http://pizzachili.dcc.uchile.cl/texts.html`

is coded as an uppercase letter A,G,C,T, and there are a few occurrences of other special characters.

- *English (English texts)* This file is the concatenation of English text files selected from etext02 to etext05 collections of Gutenberg Project. We deleted the headers related to the project so as to leave just the real text.
- *XML (structured text)* This file is an XML that provides bibliographic information on major computer science journals and proceedings and it is obtained from dblp.uni-trier.de.

We also use the file *Einstein* from the repetitive corpus from Pizza&Chili[13]. It consists of all versions of the article of Albert Einstein in the German Wikipedia up to January 12, 2010.

The last file we use is *WikiInt*, which contains an excerpt of the English Wikipedia. In this text, the alphabet consists of words instead of letters resulting in a larger alphabet than the texts from the Pizza&Chili corpus.

## 4.3 Index Size

We constructed the fully-compressed suffix trees for the texts described earlier with the sampling parameter $\delta = \lceil \log n \rceil \lceil \log \log n \rceil$. Table 2 shows the space requirements of the data structures and related information for each text.

| | DNA | English | Pitches | Proteins | Sources | XML | Einstein | WikiInt |
|---|---|---|---|---|---|---|---|---|
| $\sigma$ | 17 | 216 | 134 | 26 | 228 | 97 | 118 | 281577 |
| $n/2^{20}$ | 100.0 | 100.0 | 53.2 | 100.0 | 100.0 | 100.0 | 88.5 | 12.2 |
| $\lvert T\rvert/2^{20}$ | 167.4 | 161.7 | 87.5 | 155.6 | 162.7 | 147.4 | 175.2 | 14.8 |
| $\lvert S\rvert$ | 13,683 | 312,720 | 164,779 | 249,798 | 123,014 | 73,980 | 1,350,024 | 84 |
| $\delta$ | 135 | 135 | 130 | 135 | 135 | 135 | 135 | 120 |
| $\lvert T\rvert/\lvert S\rvert$ | 12,830 | 542 | 557 | 653 | 1,387 | 2,089 | 136 | 185,077 |
| cst_fully [MB] | 37.34 | 42.41 | 31.96 | 63.69 | 41.04 | 30.58 | 24.32 | 18.52 |
| cst_fully_sds [MB] | 37.33 | 42.07 | 31.84 | 63.50 | 40.95 | 30.55 | 22,47 | 18.52 |
| cst_fully_blind [MB] | 37.55 | 45.51 | 33.65 | 66.25 | 42.51 | 32.18 | 35,29 | 18.53 |
| cst_russo [MB] | 56.57 | 70.84 | 43.98 | 75.79 | 65.40 | 54.42 | 58.92 | n/a |
| cst_sada [MB] | 124.08 | 126.05 | 77.01 | 145.68 | 125.77 | 110.45 | 104.92 | 27.23 |
| csa_wt [MB] | 37.28 | 41.16 | 31.37 | 62.79 | 40.55 | 30.29 | 19.22 | 18.52 |
| csa_wt/cst_fully | 0.998 | 0.971 | 0.981 | 0.986 | 0.988 | 0.990 | 0.790 | 1.000 |

Table 2: Space requirements of fully-compressed suffix trees with $\delta = \lceil \log n \rceil \lceil \log \log n \rceil$. Statistics such as the number of nodes in the suffix tree $|T|$ and the sampled tree $|S|$ are also shown for each test case. Note that since the implementation of Russo et al. does not support integer alphabets, *WikiInt* was not tested with their implementation.

The total space consumption of `cst_fully` is smaller than the space consumption of the prototype by Russo et al. for every test case. One reason for this is the smaller size of the underlying CSA. Another reason is the succinct representation of the tree in `cst_fully`, as opposed to `cst_russo`, which uses a pointer-based implementation. The extra space required by `cst_fully` is less than 3 percent the size of the CSA for the texts from the

[13] `http://pizzachili.dcc.uchile.cl/repcorpus.html`

Pizza&Chili corpus, whereas `cst_russo` requires up to 13 percent the size of the CSA (see [11]) for these texts.

The line $|T|/|S|$ shows the ratio of the number of nodes in the suffix tree to the number of nodes in the sampled tree. While this ratio cannot be lower than $\delta/2$ (or even $\delta$, since leaves are not sampled in our implementation) due to the property of the sampling, we observe that the values are much larger in practice (except for *Einstein*). This is especially the case if the nodes in the suffix tree are very close to the root and few nodes have a string depth large enough to satisfy the sampling condition. Note that for the highly repetitive text *Einstein*, the value of $|T|/|S|$ is very close to $\delta$.

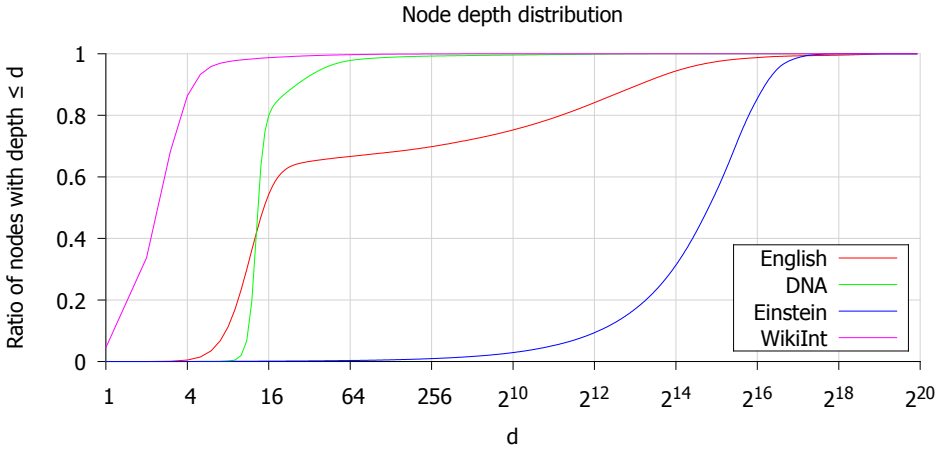


Figure 19: Distribution of the string depth of all inner nodes of the suffix tree for *English*, *DNA*, *Einstein* and *WikiInt*. The y-axis shows the ratio of inner nodes having a smaller depth than $d$ to the total number of inner nodes in the suffix tree of the corresponding text.

The distribution of depth in the suffix tree highly depends on the type of text. While *DNA*, which is an excerpt of the human genome, bears resemblance with random data and longer repetitions are exceedingly unlikely, *English* contains a lot of redundancy and repetitions of words, phrases or even whole texts. *Einstein* is highly repetitive and contains a larger portion of suffix tree nodes with high string depths. Figure 19 shows the distribution of the string depth of the inner nodes in the suffix tree for *English*, *DNA*, *Einstein* and *WikiInt*.

We will now take a more in-depth look at the space requirements of the data structures for different values of the sampling parameter $\delta$. We start by reviewing the theoretical space complexities of the data structures (see Table 3).

| | cst_fully | cst_fully_sds | cst_fully_blind |
|---|---|---|---|
| Balanced parentheses | $n/\delta$ | $n/\delta$ | $n/\delta$ |
| Leaf mapping | $n/\delta \log \delta$ | $n/\delta \log \delta$ | $n/\delta \log \delta$ |
| Depth sampling | $n/\delta \log(n/\delta)$ | $n/\delta$ | $n/\delta \log(n/\delta)$ |
| Binary depth | 0 | 0 | $n/\delta \log \log \sigma$ |
| Total | $n/\delta \log(n/\delta)$ | $n/\delta \log \delta$ | $n/\delta \log(n/\delta)$ |

Table 3: Space complexities of the components of fully-compressed suffix trees (excluding the CSA). All entries are $\mathcal{O}(\cdot)$.

The complexity of the extra space (on top of the CSA) of the original version of the fully-compressed suffix tree (`cst_fully`) is dominated by the depth sampling (assuming $\delta \leq \sqrt{n}$). The fully-compressed suffix tree with sparse depth sampling (`cst_fully_sds`) requires only $\mathcal{O}(n/\delta)$ bits for the depth sampling, hence its extra space is dominated by the leaf mapping. The blind fully-compressed suffix tree (`cst_fully_blind`) additionally stores the binary depth of each sampled node. Its extra space complexity is the same as that of `cst_fully`.

What is not represented in the asymptotic complexity is the constant factor. Assuming there are $s$ sampled nodes in `cst_fully`, in `cst_fully_blind` there are about $s$ additional nodes in the child sampling. In the process of making the sampled tree a binary tree, further nodes are added to the sampled tree, so that in total about $4s$ nodes are in the binary sampled tree. Therefore, the balanced parentheses representation and the leaf sampling of `cst_fully_blind` require about 4 times as much space as their counterparts in `cst_fully`.

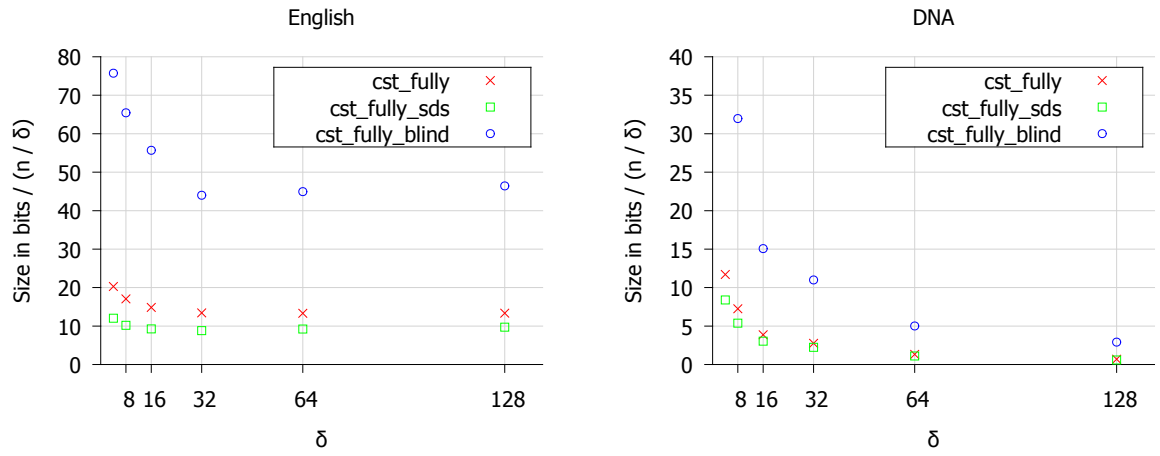


Figure 20: Space requirements of fully-compressed suffix trees for different values of $\delta$ and *English* and *DNA*. The space required for the CSA is not included.

We built the three variants of the fully-compressed suffix tree for *English* and *DNA* for different values of $\delta$. Figure 20 shows their space requirements. As the value on the y-axis shows the space divided by $n/\delta$, theory suggests we find a logarithmic curve. Instead, the space requirement per $n/\delta$ decreases with increasing $\delta$. This is because the number of nodes satisfying the sampling condition decreases.

The extra space consumption of `cst_fully_blind` is about 4 times as high as that of `cst_fully`, as expected. The extra space of `cst_fully_sds` is only marginally smaller than `cst_fully`, especially for larger values of $\delta$. To see why this is the case, we take a look at the space consumption of the individual components of the FCST.

Figure 21 shows the space consumption of each individual component of `cst_fully` for different values of $\delta$. For the leaf mapping, we expect a value of $\log \delta$ bits per sampled node, which is between 2 and 7 bits for the 100 MB of *English*. In practice, we see values between 10 and 20 bits per sampled node. One reason for this is that the actual number of sampled nodes is much less than $\frac{n}{\delta/2}$ (see Table 2), so that the distance between two one bits in the leaf sampling is higher and it requires more space per sampled node (but less overall space). For the depth sampling, we expect $\log \frac{n}{\delta}$ bits per sampled node, which

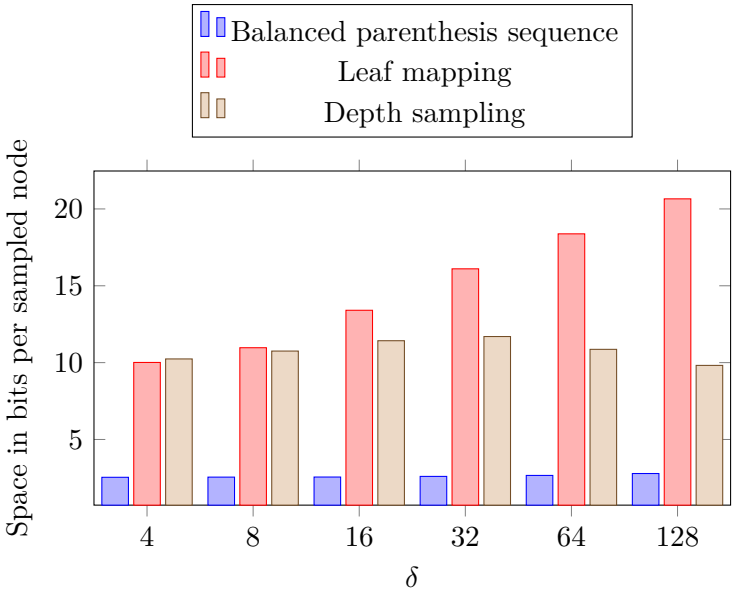


Figure 21: Space consumption per sampled node for the individual components of `cst_fully` for *English* and different values of $\delta$.

is between 20 and 25 bits for the 100 MB of *English*. Additional overhead is introduced due to the way the depth sampling is stored[14]. Our results show that the depth sampling requires only about 10 bits per sampled node. This is because $n$ is a very pessimistic upper bound for the string depth of a node and most nodes have a very small string depth in practice, as we have seen in Figure 19.

The small space consumption of the depth sampling is the reason why the space saved by using the sparser depth sampling of `cst_fully_sds` is very small in practice.

## 4.4 Query Time

We will now evaluate the query times of our implementation. We start by reviewing the theoretical time complexities of the data structures, which are shown in Table 4.

| Operation | cst_fully | cst_fully_sds | cst_fully_blind |
|---|---|---|---|
| $root$ | 1 | 1 | 1 |
| $is\text{-}leaf(v)$ | 1 | 1 | 1 |
| $parent(v)$ | $\delta(t_\psi + t_{LF})$ | $\delta(t_\psi + t_{LF})$ | $\delta(t_\psi + t_{LF})$ |
| $child(v, c)$ | $\delta \cdot t_\psi + t_{text} \log n$ | $\delta(t_\psi \log depth(v) + t_{LF}) + t_{text} \log n$ | $\delta \cdot t_\psi + \log \sigma + t_{text}$ |
| $first\text{-}child(v)$ | $\delta \cdot t_\psi + t_{text} \log n$ | $\delta(t_\psi \log depth(v) + t_{LF}) + t_{text} \log n$ | $\delta \cdot t_\psi + \log \sigma + t_{text}$ |
| $next\text{-}sibling(v)$ | $\delta \cdot t_\psi + t_{text} \log n$ | $\delta(t_\psi \log depth(v) + t_{LF}) + t_{text} \log n$ | $\delta \cdot t_\psi + \log \sigma + t_{text}$ |
| $letter(v, d)$ | $t_{text}$ | $t_{text}$ | $t_{text}$ |
| $slink(v)$ | $\delta(t_\psi + t_{LF})$ | $\delta(t_\psi + t_{LF})$ | $\delta(t_\psi + t_{LF})$ |
| $LCA(v, w)$ | $\delta(t_\psi + t_{LF})$ | $\delta(t_\psi + t_{LF})$ | $\delta(t_\psi + t_{LF})$ |
| $depth(v)$ | $\delta \cdot t_\psi$ | $\delta(t_\psi \log depth(v) + t_{LF})$ | $\delta \cdot t_\psi$ |

Table 4: Time complexities of the different variants of the fully-compressed suffix tree. All entries are $\mathcal{O}(\cdot)$.

[14]In our evaluation, the SDSL type `dac_vector<>` is used to store the depth sampling.

*root* and *is-leaf* are simple operations on the fully-compressed suffix tree, which are computed in constant time. *first-child* and *next-sibling* are not very interesting, as they are computationally equivalent to *child* (with an additional call to *letter*). *letter* is implemented only by CSA operations and is therefore independent of the sampled tree. We still include it for reference purposes. We will concentrate on the following six operations: *parent*, *child*, *letter*, *slink*, $LCA$ and *depth*.

We evaluate each operation by computing it 10,000 times with different arguments chosen at random, measuring the average execution time.

Each operation takes a node as an argument, while some require additional arguments. The node is chosen in the following way: We choose a random leaf $u = [u, u]$ by choosing $u$ uniformly from the interval $[0, n-2]$ and calculate $v = LCA([u, u], [u+1, u+1])$, which yields any inner node $v$ with a probability $\frac{deg(v)-1}{n-1}$, where $deg(v)$ is the degree of node $v$.

For the evaluation of the *child*-operation, we need a character $c$ for each evaluation of $child(v, c)$. We choose a leaf $u$ uniformly from all the leaves rooted in $v$ and choose $c$ as the first character on the label from $v$ to $u$.

For the $LCA$ operation, we need a pair of nodes. Choosing two nodes $v$ and $v'$ independently causes $LCA(v, v')$ to be *root* most of the time. Hence, we choose a random inner node $v$ using the method described above. We then choose two leaves $v$, $v'$ uniformly from the leaves in the subtree rooted at $v$. This ensures that $LCA(v, v')$ is a descendant of $v$.

Evaluating $parent(v)$ for a uniformly chosen node $v$ often yields nodes that are near the root of the tree, which are particularly easy to compute given the structure of the fully-compressed suffix tree. So instead, we choose a random child $v'$ of a uniformly chosen node $v$, so that we know that $v = parent(v')$ is uniformly distributed.

For the *letter* operation and an inner node $v \neq root$, we choose $d = depth(v)$, the last letter of the path-label from *root* to $v$.

Table 5 shows the query times of our implementation in comparison with both the implementation of Russo et al. and the compressed suffix tree by Sadakane. Our implementation is faster for all operations (except *letter*) by a factor of 3 to 9. Note that the arguments chosen to evaluate the operations in the implementation of Russo et al. are different in some cases. For instance, their benchmark for the *letter* operation retrieves the first letter of the path label of a node $v$, which is particularly easy to compute, while our implementation retrieves the last letter, which requires applying $\psi^i$.

Figures 22, 23, 24, 25 and 26 show the query times for *English* (100 MB), *DNA* (100 MB), *WikiInt*, *English* (2.2 GB) and *Einstein* respectively.

The graph showing the execution time for the $LCA$ operation is not linear, as theory suggests. Instead, the graph becomes more flat with increasing $\delta$. This is because many nodes in the suffix tree are close to the root. The time required to calculate the string depth for a node $v$ with $depth(v) = d$ is the same for any $\delta > 2d$, since in this case $slink^d(v) = root$ is the only sampled node in the suffix link sequence of $v$. Also note that $LCA$ is slightly slower for `cst_fully_sds`, since Lemma 6 is used instead of Lemma 3, which potentially uses more applications of $LF$. The *parent* operation takes about twice the time to compute as the $LCA$ operation.

The *depth*-operation is much slower for `cst_fully_sds` than for the other variants. The

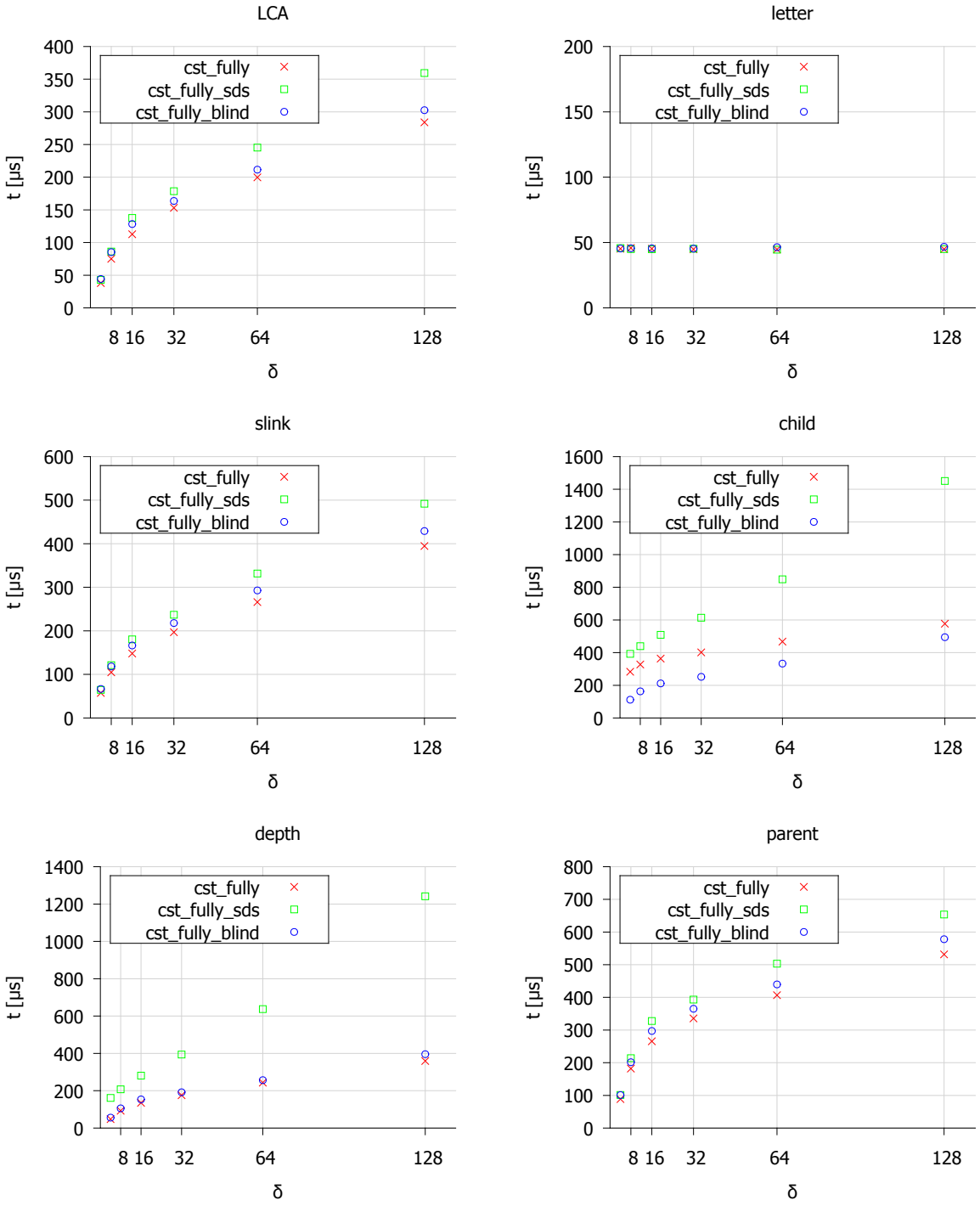


Figure 22: Query times for *English* and different values of $\delta$.

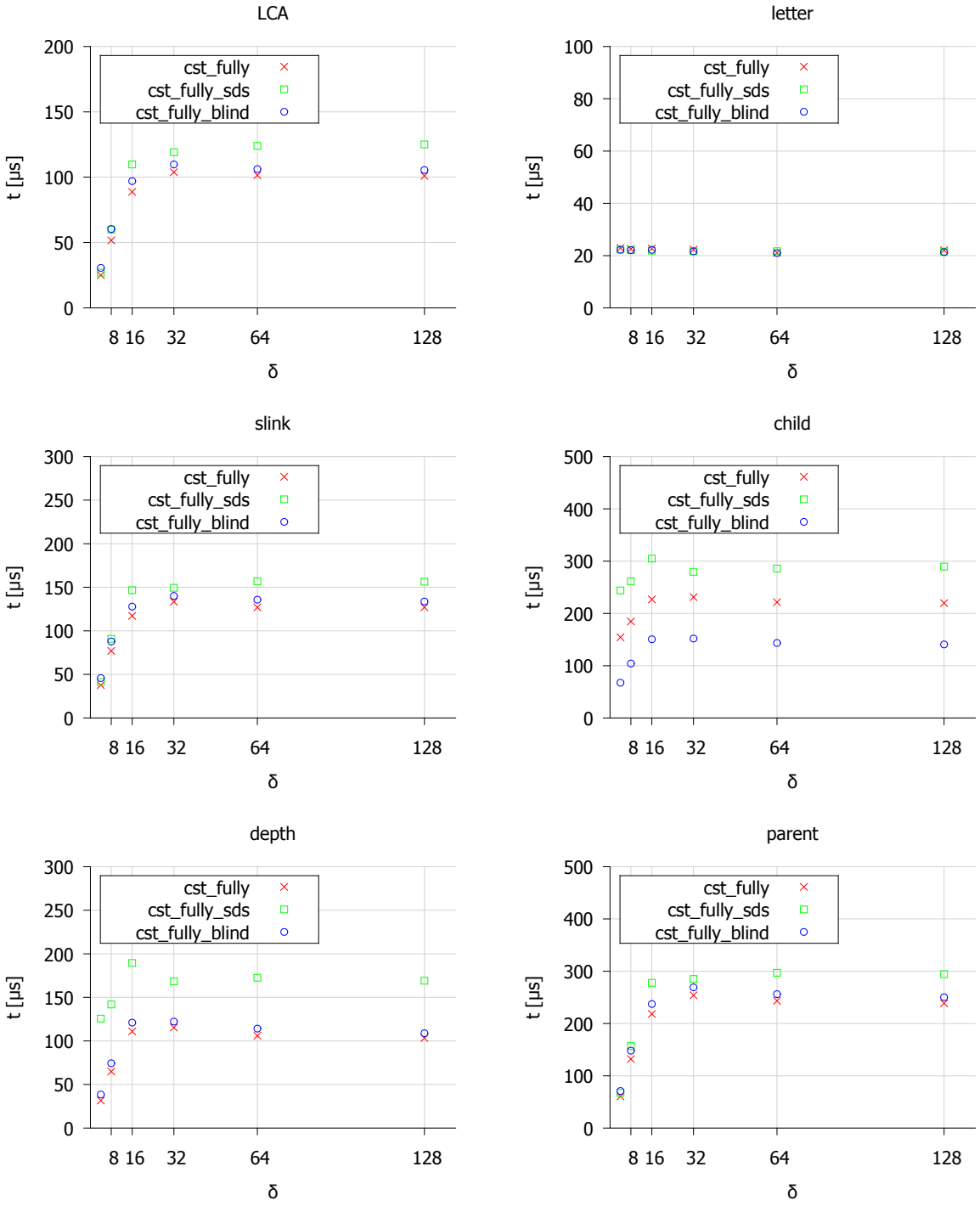


Figure 23: Query times for *DNA* and different values of $\delta$.

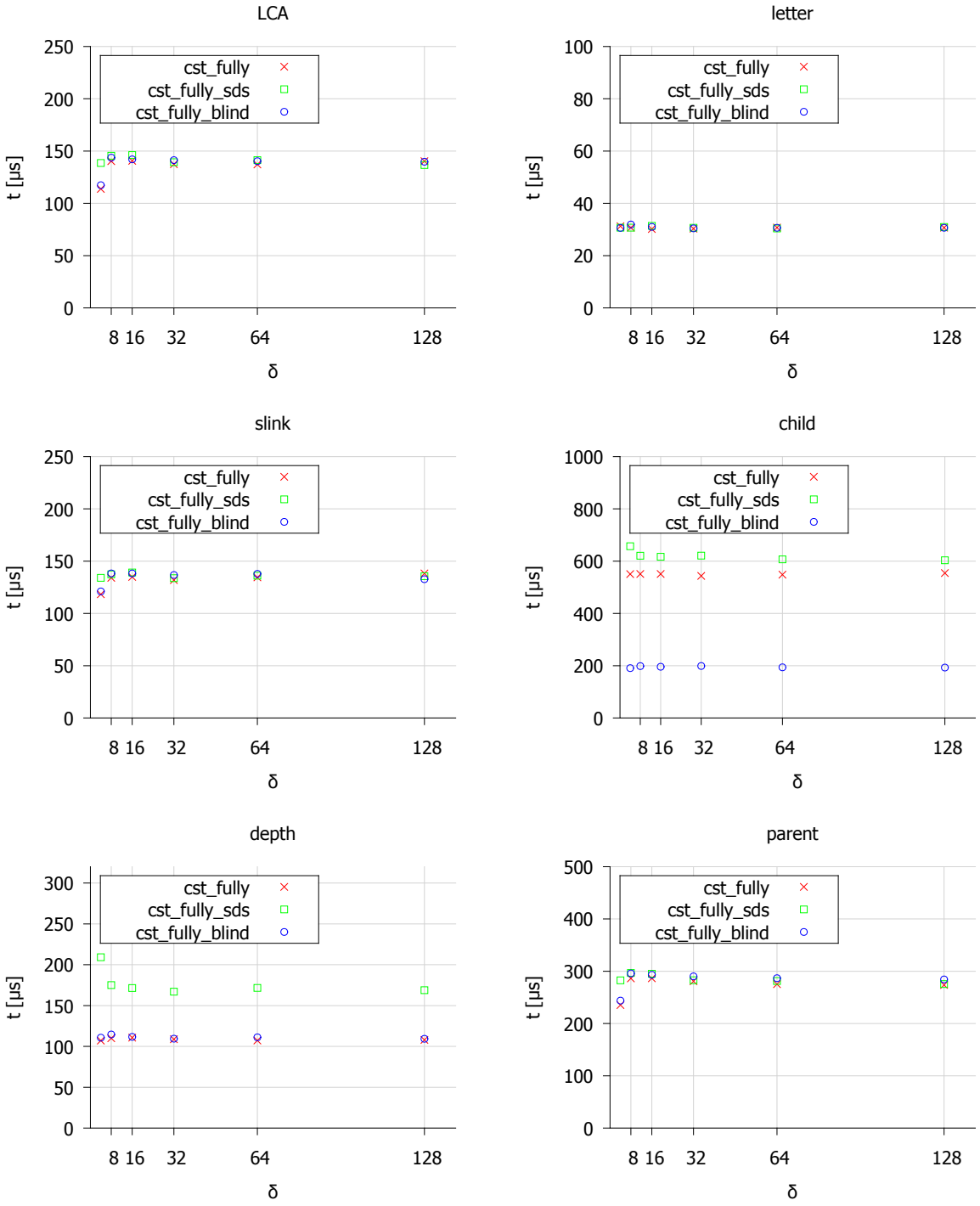


Figure 24: Query times for *WikiInt* and different values of $\delta$.

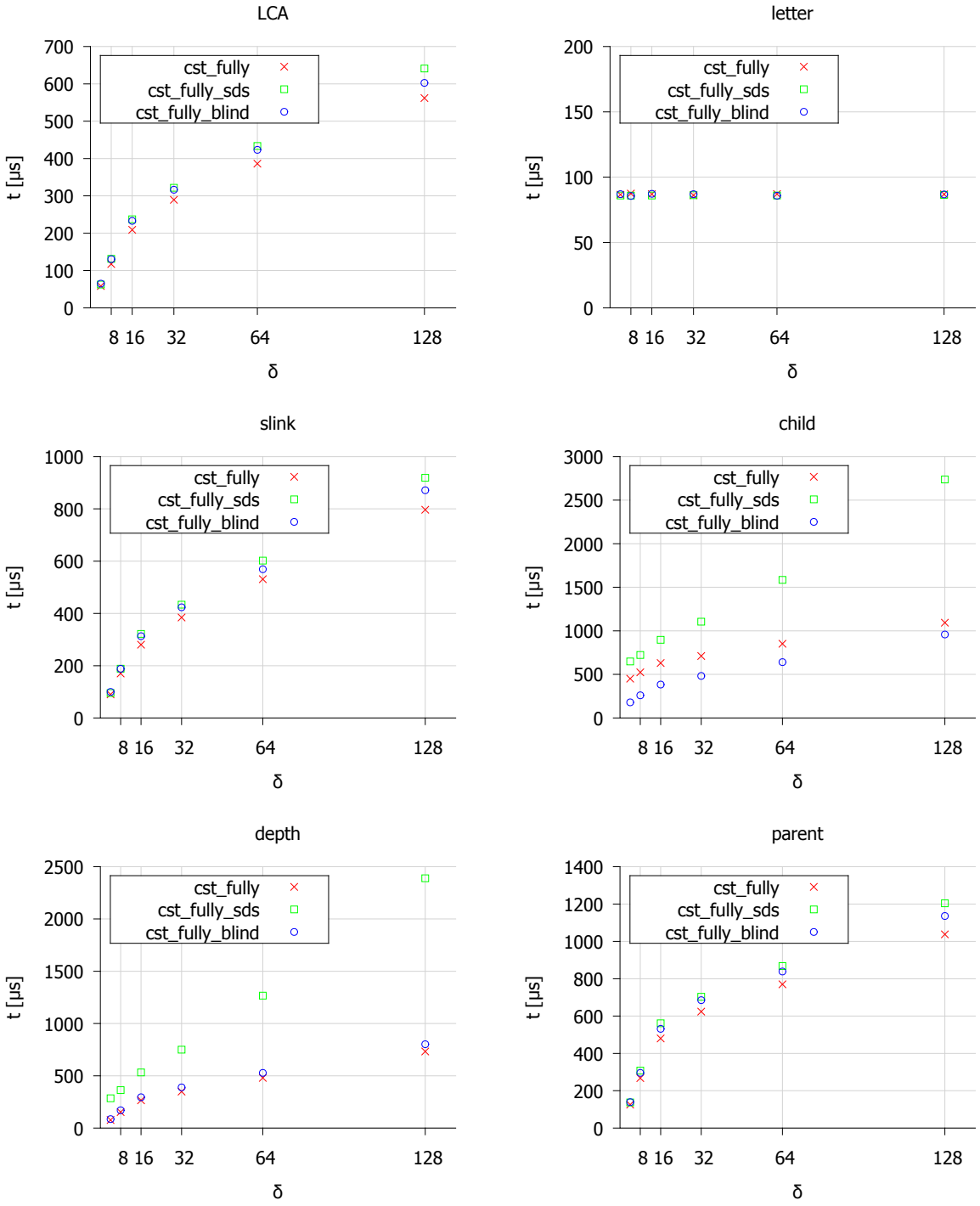


Figure 25: Query times for *English* (2.2 GB) and different values of $\delta$.

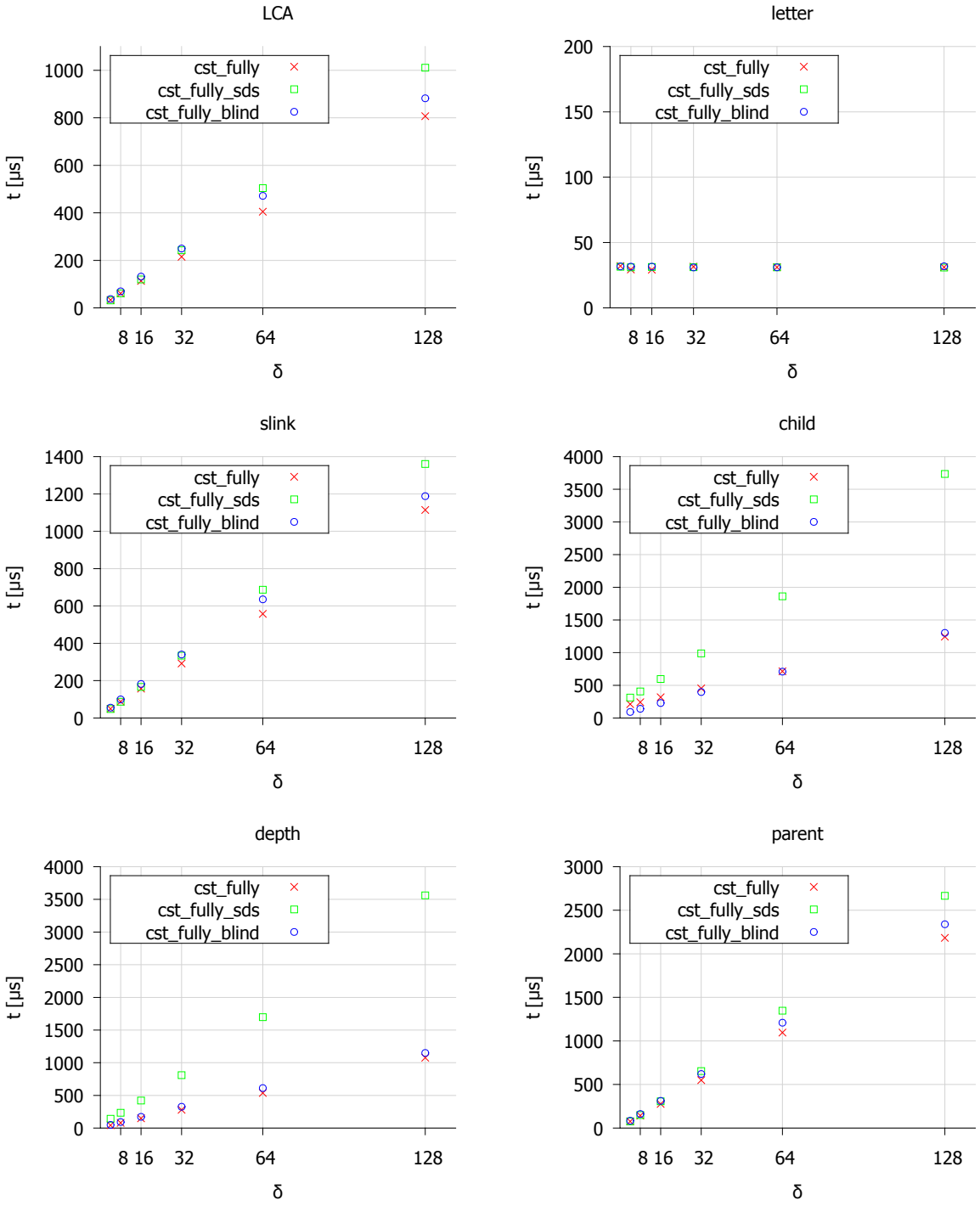


Figure 26: Query times for *Einstein* and different values of $\delta$.

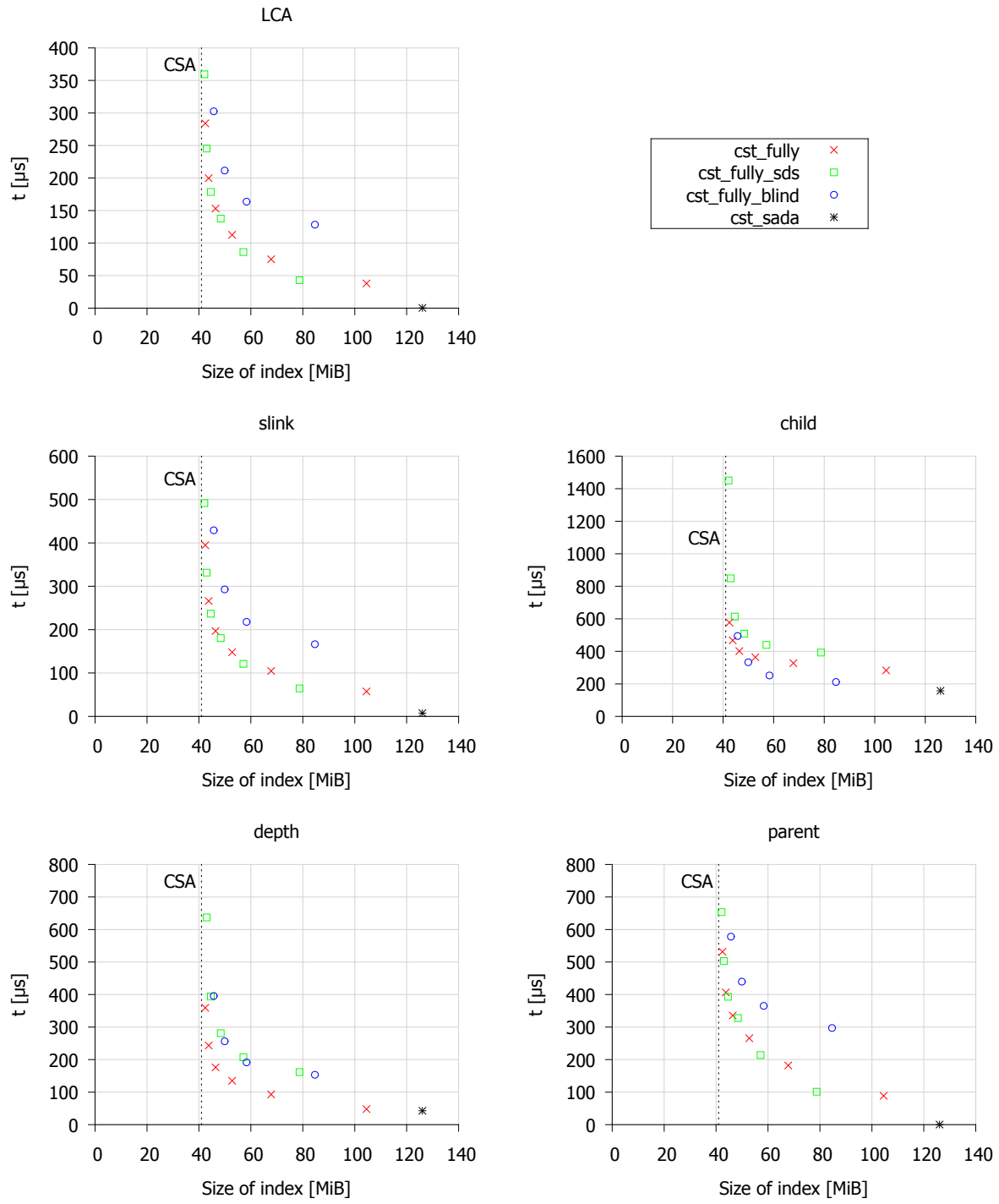


Figure 27: Query times and space consumption for *English* and $\delta = 4$, 8, 16, 32, 64 and 128. The query time and size of `cst_sada` is included for reference. The line labeled CSA shows the size of the compressed suffix array, which is a lower bound of the size of the FCST. Results with very large index size or query time are omitted (they are outside the range shown here).

| Operation | | DNA | English | Pitches | Proteins | Sources | XML | Einstein |
|---|---|---|---|---|---|---|---|---|
| *LCA* | `cst_fully` | 96.38 | 296.51 | 356.83 | 319.56 | 298.33 | 347.12 | 859.62 |
| | `cst_russo` | 726.12 | 2093.07 | 2613.58 | 1610.44 | 2344.94 | 3234.85 | 7043.08 |
| | `cst_sada` | 0.19 | 0.27 | 0.35 | 0.28 | 0.30 | 0.26 | 0.20 |
| *letter* | `cst_fully` | 20.96 | 45.41 | 52.95 | 42.88 | 56.85 | 51.62 | 32.28 |
| | `cst_russo` | 2.82 | 3.96 | 4.97 | 3.69 | 4.98 | 4.97 | 4.30 |
| | `cst_sada` | 22.12 | 47.20 | 53.28 | 44.66 | 57.72 | 51.49 | 31.32 |
| *slink* | `cst_fully` | 123.35 | 404.97 | 473.77 | 380.55 | 368.35 | 422.50 | 1173.24 |
| | `cst_russo` | 704.08 | 2085.43 | 2620.89 | 1590.20 | 2307.60 | 3207.01 | 7079.65 |
| | `cst_sada` | 4.25 | 7.58 | 9.94 | 8.50 | 8.89 | 6.38 | 5.25 |
| *child* | `cst_fully` | 214.69 | 587.36 | 669.53 | 561.15 | 617.45 | 685.70 | 1316.67 |
| | `cst_russo` | 586.01 | 1791.79 | 2173.83 | 1363.05 | 1857.30 | 2473.21 | 6193.61 |
| | `cst_sada` | 74.42 | 155.55 | 176.91 | 161.77 | 181.10 | 182.97 | 57.06 |
| *depth* | `cst_fully` | 102.20 | 366.15 | 398.49 | 342.39 | 329.13 | 380.07 | 1134.73 |
| | `cst_russo` | 583.44 | 1805.27 | 2148.23 | 1361.62 | 1880.76 | 2472.39 | 6166.50 |
| | `cst_sada` | 31.44 | 43.13 | 58.11 | 60.37 | 42.54 | 26.18 | 15.49 |
| *parent* | `cst_fully` | 233.29 | 550.28 | 672.33 | 601.28 | 604.06 | 804.09 | 2309.25 |
| | `cst_russo` | 1191.47 | 1425.04 | 2700.27 | 1829.66 | 2804.99 | 5057.32 | 13574.66 |
| | `cst_sada` | 0.14 | 0.20 | 0.20 | 0.20 | 0.19 | 0.18 | 0.10 |

Table 5: Query times of our fully-compressed suffix tree implementation (`cst_fully`) with $\delta = \lceil \log n \rceil \lceil \log \log n \rceil$, the implementation of Russo et al. (`cst_russo`) using the same value of $\delta$ and the compressed suffix tree by Sadakane (`cst_sada`). All values are in microseconds.

average string depth of inner nodes in test case English is about 5000, so we would expect a slowdown of factor 12 caused by the final step in which the node which is in the depth sampling is determined. We can see in Figure 22 that the slowdown factor is only about 3, independent of the value of $\delta$. This discrepancy is again due to the fact that most of the nodes are close to the root.

The *child* operation is fastest for `cst_fully_blind` for every text and every value of $\delta$. The time improvement is especially large for small values of $\delta$. To see why this is the case, we recall that for `cst_fully` and `cst_fully_sds` *child* is implemented by a binary search and a call to *depth* to determine the branching letter. While the former is potentially time consuming, it does not depend on the sampled tree and is therefore independent of $\delta$. The *child* operation of `cst_fully_blind` benefits from smaller values of $\delta$ to a greater extent. This also explains why `cst_fully_sds`, which has a slow *depth* operation, is also very slow for *child*.

The graphs for *DNA* (Figure 23) look similar to those for *English* (Figure 22) for small values of $\delta$. For larger values, there is no noticeable increase in time, as the suffix tree for this text has very few nodes with high string depths (see Figure 19). For *WikiInt* (Figure 24), this is even more extreme and the time is almost the same for any value of $\delta \geq 8$.

The execution times for *English* (2.2 GB, Figure 25) are by a factor of about 1.9 slower than the operations on *English* (100 MB, Figure 22). This is mainly due to caching effects, since the index is larger and a smaller proportion of it can be stored in the cache.

The execution times of the operations $LCA$, *parent* and *slink* for *Einstein* are almost perfectly linear with $\delta$. Since the text is highly repetitive, most of the nodes have a high string depth and we have to perform the full $\delta$ steps to calculate $LCA$.

Figure 27 shows time-space trade-offs for *English* (100 MB) for different values of $\delta$.

`cst_fully` provides a very good overall performance. `cst_fully_sds` is marginally better for $LCA$, *parent* and *slink* but significantly worse for *depth* and *child*. `cst_fully_blind` is slightly better for *child* but worse for the remaining operations. This shows that values of $\delta < \lceil \log n \rceil \lceil \log \log n \rceil$ provide very interesting trade-offs in practice.

## 4.5 Construction Time

We will now evaluate the construction time of our implementation.

| Index | DNA | English | Pitches | Proteins | Sources | XML | Einstein |
|---|---|---|---|---|---|---|---|
| `cst_fully` | 157 | 209 | 121 | 226 | 183 | 178 | 225 |
| `cst_russo` | 774 | 4246 | 579 | 1416 | 1731 | 926 | 60098 |

Table 6: Construction time of our fully-compressed suffix tree implementation (`cst_fully`) with $\delta = \lceil \log n \rceil \lceil \log \log n \rceil$ and the implementation of Russo et al. (`cst_russo`) using the same value of $\delta$. All values are in seconds.

Table 6 shows the construction time of our implementation (`cst_fully`) and the implementation of Russo et al. (`cst_russo`). Our implementation is at least 4 times faster for all texts. It is striking that the construction of `cst_russo` is especially slow if the number of sampled nodes is high. An extreme example of this is the highly repetitive text *Einstein*, which took more than 16 hours of construction time for `cst_russo`. This is because the overhead for adding nodes to the sampled tree is particularly high in the pointer-based representation that is used in `cst_russo`.

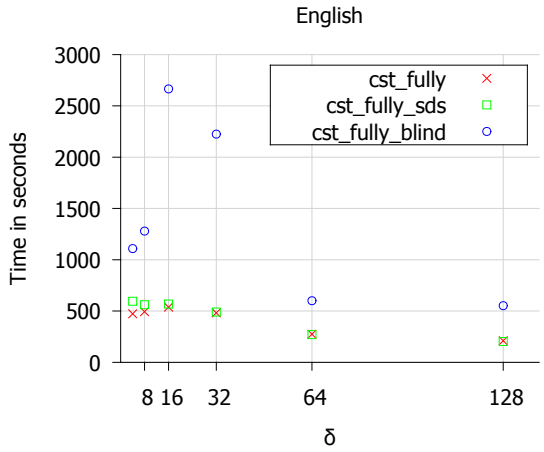


Figure 28: Construction time for *English* and different values of $\delta$.

Figure 28 shows the construction time for the different variants of the FCST and different values of $\delta$. The construction of `cst_fully_blind` is very slow, since the sampling conditions for the child sampling have to be checked for every child $v.\alpha$ of each sampled node $v$.

# 5 Conclusion

In this thesis, we implemented three variants of the fully-compressed suffix tree. The implementations are generic and can be customized with the data structures provided by the Succinct Data Structure Library. In an experimental study we explored different time-space trade-offs by varying the node sampling parameter ($\delta$) on different real-world text inputs. The study showed that we improve on Russo et al.'s prototype implementation regarding index space, query and construction time. While the improvements compared to Russo et al. are significant, the navigational operations are still much slower compared to a regular compressed suffix trees.

Another outcome of our experimental study is that the fully-compressed suffix tree with sparse depth sampling slightly outperforms the other variants on navigational operations $LCA$, *slink*, and *parent*. However, operations *depth* and *child* are not competitive with the original proposal.

Our new binary fully-compressed suffix tree improves the time complexity for *child* queries from $\mathcal{O}(\log n(\log\log n)^2)$ (see [11]) to $\mathcal{O}(\log n \log\log n)$ using the same asymptotic space complexity as the fully-compressed suffix tree by Russo et al. This improvement is also seen in practice, albeit the remaining operations are a bit slower.

Regarding the space consumption of the data structures, there is a significant gap between theory and practice. As we have shown in the empirical evaluation, the upper bound of $n$ for the string depth of suffix tree nodes is a rather pessimistic one. Szpankowski [29] shows a more realistic upper bound of $c \cdot \log n$ for a constant $c$ (depending on the text type) using a probabilistic model.

While we have made significant practical improvements there are still open challenges: The construction process currently depends on the uncompressed input size and not only on the size of the final output. The speed of the construction is also still slow compared to the construction process of the regular compressed suffix tree. Moreover, there is no efficient way to calculate an ID for a node, like the pre-order index. Therefore it is not easily possible to attach satellite data to a node.

# References


[1] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Communications of the ACM*, vol. 20, no. 10, pp. 762–772, 1977.

[2] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt, "Fast pattern matching in strings," *SIAM journal on computing*, vol. 6, no. 2, pp. 323–350, 1977.

[3] D. Gusfield, *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge university press, 1997.

[4] P. Weiner, "Linear pattern matching algorithms," in *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*. IEEE, 1973, pp. 1–11.

[5] S. Kurtz, "Reducing the space requirement of suffix trees," *Software-Practice and Experience*, vol. 29, no. 13, pp. 1149–71, 1999.

[6] M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch, "Replacing suffix trees with enhanced suffix arrays," *Journal of Discrete Algorithms*, vol. 2, no. 1, pp. 53–86, 2004.

[7] R. Grossi and J. S. Vitter, "Compressed suffix arrays and suffix trees with applications to text indexing and string matching," *SIAM Journal on Computing*, vol. 35, no. 2, pp. 378–407, 2005.

[8] P. Ferragina and G. Manzini, "Opportunistic data structures with applications," in *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*. IEEE, 2000, pp. 390–398.

[9] K. Sadakane, "Compressed suffix trees with full functionality," *Theory of Computing Systems*, vol. 41, no. 4, pp. 589–607, 2007.

[10] L. M. Russo, G. Navarro, and A. L. Oliveira, "Fully-compressed suffix trees," in *LATIN 2008: Theoretical Informatics*. Springer, 2008, pp. 362–373.

[11] L. Russo, G. Navarro, and A. L. Oliveira, "Fully compressed suffix trees," *ACM Transactions on Algorithms (TALG)*, vol. 7, no. 4, p. 53, 2011.

[12] S. Gog, T. Beller, A. Moffat, and M. Petri, "From theory to practice: Plug and play with succinct data structures," in *Experimental Algorithms*. Springer, 2014, pp. 326–337.

[13] G. Navarro and L. Russo, "Fast fully-compressed suffix trees," in *Data Compression Conference (DCC), 2014*. IEEE, 2014, pp. 283–291.

[14] E. Ukkonen, "On-line construction of suffix trees," *Algorithmica*, vol. 14, no. 3, pp. 249–260, 1995.

[15] D. Harel and R. E. Tarjan, "Fast algorithms for finding nearest common ancestors," *siam Journal on Computing*, vol. 13, no. 2, pp. 338–355, 1984.

[16] G. Navarro and V. Mäkinen, "Compressed full-text indexes," *ACM Computing Surveys (CSUR)*, vol. 39, no. 1, p. 2, 2007.

[17] G. J. Jacobson, "Succinct static data structures," 1988.

[18] D. Clark, "Compact pat trees," Ph.D. dissertation, PhD thesis, University of Waterloo, 1998.

[19] D. Okanohara and K. Sadakane, "Practical entropy-compressed rank/select dictionary." in *ALENEX*. SIAM, 2007.

[20] R. Raman, V. Raman, and S. S. Rao, "Succinct indexable dictionaries with applications to encoding k-ary trees and multisets," in *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2002, pp. 233–242.

[21] R. Grossi, A. Gupta, and J. S. Vitter, "High-order entropy-compressed text indexes," in *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 2003, pp. 841–850.

[22] K. Sadakane and G. Navarro, "Fully-functional succinct trees," in *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2010, pp. 134–149.

[23] J. I. Munro and V. Raman, "Succinct representation of balanced parentheses and static trees," *SIAM Journal on Computing*, vol. 31, no. 3, pp. 762–776, 2001.

[24] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," 1994.

[25] S. Gog, G. Navarro, and M. Petri, "Improved and extended locating functionality on compressed suffix arrays," *Journal of Discrete Algorithms*, 2015.

[26] V. Mäkinen and G. Navarro, "Succinct suffix arrays based on run-length encoding," in *Combinatorial Pattern Matching*. Springer, 2005, pp. 45–56.

[27] P. Ferragina and R. Grossi, "The string b-tree: a new data structure for string search in external memory and its applications," *Journal of the ACM (JACM)*, vol. 46, no. 2, pp. 236–280, 1999.

[28] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro, "Compressed representations of sequences and full-text indexes," *ACM Transactions on Algorithms (TALG)*, vol. 3, no. 2, p. 20, 2007.

[29] W. Szpankowski, "A generalized suffix tree and its (un) expected asymptotic behaviors," *SIAM Journal on Computing*, vol. 22, no. 6, pp. 1176–1198, 1993.