Verbatim Implementation of a Fast and Space-Efficient Indexed Pattern Matching Algorithm

Tuukka Norri

MSc Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, Tuesday 23rd August, 2016
Approximate string matching considers finding a given text pattern in another text while allowing some number of differences. In the offline version of the problem, the text is known beforehand and is processed to generate an indexing data structure. While the problem has received a lot of attention and it has many practical uses in bioinformatics, the common tools often do not make use of the algorithms the time and space complexities of which are the best ones known. Hence it is interesting to compare the performance of an efficient algorithm to tools that make use of heuristics.

In this work, a pattern matching algorithm by T.-W. Lam, W.-K. Sung and S.-S. Wong is described. An implementation of the algorithm is provided and tested against two other tools, namely Erne 2 and readaligner 2012. The algorithm by Lam, Sung and Wong searches the text for the pattern while allowing one mismatch or difference, that is, also allowing character insertion and deletion. It makes use of certain types of compressed suffix array and compressed suffix tree that provide fast operations. Additionally, to restrict the search to relevant parts of the suffix tree, a sample is taken from the suffix array and the sampled indices are stored into a data structure that provides double logarithmic worst case range queries.

To find the pattern in the text while allowing \( k \) errors, the algorithm is combined with a dynamic programming algorithm. The latter is used to find partial matches with \( k-1 \) errors. The candidate occurrences are located from the suffix tree and these branches are used with Lam, Sung and Wong’s algorithm. For a text of length \( n \) and a pattern of length \( m \) drawn from an alphabet of size \( \sigma \), the time complexity of the algorithm is \( O(\sigma^k n^k (k+\log \log n) + \text{occ}) \) using an \( O(n\sqrt{\log n} \log \sigma) \)-bit indexing data structure, where \( \text{occ} \) is the number of occurrences in the text, given that \( \sigma \) is \( O(2^{\sqrt{\log n}}) \). For short patterns, this is the best known time complexity with an indexing data structure of the given size.

The results indicate that in practice relying on heuristics yields better results in terms of time and memory use. While such an outcome is not remarkable, some important data structures were implemented in the process. An implementation of S. S. Rao’s compressed suffix array already existed but it was rewritten to allow using different supporting data structures for e.g. rank and select support. The inverse suffix array described by Lam, Sung and Wong was implemented. Also, while implementations of X and Y-fast were available, to the author’s knowledge a publicly available implementation of these combined with perfect hashing had not been produced before. Moreover, no implementation of Lam, Sung and Wong’s algorithm was known to exist before.

ACM Computing Classification System (CCS):
10010405.10010444.10010087.10010086
[Applied computing ~ Molecular sequence analysis]
10003752.10003809.10010031.10010032
[Theory of computation ~ Pattern matching]
10002944.10011123.10011674
[General and reference ~ Performance]
Acknowledgement

I would like to express my gratitude to my supervisor Prof. Veli Mäkinen and my instructor Dr. Travis Gagie for their invaluable support and guidance during my work on this master’s thesis. I would also like to thank the Department of Computer Science at University of Helsinki for funding the research project that led to this thesis.

Finally, I wish to thank my friends and family for all of the support that I have received throughout my studies.
A SDSL Benchmark Results

A.1 Suffix Array Benchmarks ............... A1
A.1.1 Count Benchmark ................... A1
A.1.2 Locate Benchmark .................. A1
A.1.3 Extract Benchmark .................. A1
A.2 Suffix Tree Benchmarks ............... A6
1 Introduction

1.1 The Approximate String Matching Problem

Given a text and a pattern, both drawn from the same alphabet, the intent in the approximate string matching problem is to find all occurrences of the pattern in the text with a given number of errors allowed. There are two variations of the problem with slightly different definitions of error. In the $k$-difference problem, the occurrences of the pattern may have edit distance of at most $k$ from the original text. Edit distance is the minimum number of character insertions, deletions and substitutions required to convert a text string to another. In the $k$-mismatch problem, only character substitutions are allowed.

Especially the $k$-difference problem is important in computational biology, since DNA and protein sequence alignment are common tasks and require inexact matching. The sequences involved tend to be very long; for instance the size of the human genome is over 3,200 million basepairs. Some other vertebrates and many plants have many times larger genomes. Scanning the whole sequence for inexact matches would be extremely time-consuming and any practical applications would not be feasible.

To speed up the matching process, one or several of the sequences involved may be preprocessed to create a data structure with which some useful operations may be done reasonably quickly. While many such indexing data structures have been designed, the nature of the biological sequences places constraints on their applicability.

The alphabet for DNA requires only four characters if only the four nucleobases, adenine, guanine, thymine and cytosine, are considered, or 17 characters if various ambiguities are allowed in the represented sequence. The alphabet for the 20 proteinogenic amino acids with the addition of pyrrolysine, selenocysteine and various ambiguities requires 28 characters. While these are not notable requirements, the sequences do not have some properties of many natural languages, which have been utilized in some indexing data structures. The lack of word boundaries, for instance, limits the usefulness of some efficient data structures such as inverted files, which store the text positions where each word occurs. Other data structures, such as suffix arrays and suffix trees, do not make use of this property of the indexed text and thus are a good candidate for use in a pattern matching algorithm.

1.2 Existing Work

The online version of the problem, that is, where both the text and the pattern are not known in advance, may be solved with a dynamic programming algorithm by Sellers in $O(nm)$ time and space where $n$ is the text length and $m$ is the pattern length [Sel80]. This was improved by Landau and
Table 1: Comparison of various results for the one-mismatch and one-difference problems. $n$ is the text length, $m$ is the pattern length, $\sigma$ is the alphabet size, $\text{occ}$ is the number of occurrences and $\varepsilon$ is a positive constant.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Space requirement</th>
<th>Query time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cobbs(^1)</td>
<td>$O(\sigma q)$ words</td>
<td>$O(m^2 + \text{occ})$</td>
</tr>
<tr>
<td>[Cob95]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Buchsbaum et al.(^2)</td>
<td>$O(n \log n)$ words</td>
<td>$O(m \log \log n + \text{occ})$</td>
</tr>
<tr>
<td>[BGW00]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cole et al.(^3)</td>
<td>$O(n \log n)$ words</td>
<td>$O(m + \log n \log \log n + \text{occ})$</td>
</tr>
<tr>
<td>[CGL04]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hyunh et al.(^4)</td>
<td>$O(n \log n)$ bits</td>
<td>$O(\sigma m \log n + \text{occ})$</td>
</tr>
<tr>
<td>[HHLS06]</td>
<td>$O(n)$ bits</td>
<td>$O(\sigma m \log^2 n + \text{occ} \cdot \log n)$</td>
</tr>
<tr>
<td>Lam et al.(^5)</td>
<td>$O(n \sqrt{\log n \log \sigma})$ bits</td>
<td>$O(\sigma m \log \log n + \text{occ})$</td>
</tr>
<tr>
<td>[LSW08]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chan et al.(^6)</td>
<td>$O(n)$ bits</td>
<td>$O(m + \text{occ} + \log^2 n \log \log n)$</td>
</tr>
<tr>
<td>[Cha+06](^4,5)</td>
<td></td>
<td>$O(m + (\text{occ} + \log^4 n \log \log n) \log^\varepsilon n)$</td>
</tr>
</tbody>
</table>

Table 2: Comparison of various results for the $k$-mismatch and $k$-difference problems. $n$ is the text length, $m$ is the pattern length, $\sigma$ is the alphabet size, $\text{occ}$ is the number of occurrences and $c$, $c'$ and $\varepsilon$ are positive constants.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Space requirement</th>
<th>Query time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cobbs(^1)</td>
<td>$O(\sigma q)$ words</td>
<td>$O(\sigma mq + \text{occ})$</td>
</tr>
<tr>
<td>[Cob95]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cole et al.(^3)</td>
<td>$O(\frac{n}{k!} (\log n)^k)$ w.</td>
<td>$O(m + \text{occ} \cdot 3^k + \frac{1}{k!} (c' \log n)^k \log \log n)$</td>
</tr>
<tr>
<td>[CGL04]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hyunh et al.(^6)</td>
<td>$O(n \log n)$ bits</td>
<td>$O(\sigma^k m^k \cdot \max{k, \log n} + \text{occ})$</td>
</tr>
<tr>
<td>[HHLS06]</td>
<td>$O(n)$ bits</td>
<td>$O(\sigma^k m^k \cdot \max{k, \log^2 n} + \text{occ} \cdot \log n)$</td>
</tr>
<tr>
<td>Lam et al.(^6)</td>
<td>$O(n \sqrt{\log n \log \sigma})$ bits</td>
<td>$O(\sigma^k m^k (k + \log \log n) + \text{occ})$</td>
</tr>
<tr>
<td>[LSW08]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chan et al.(^4,5)</td>
<td>$O(n)$ words</td>
<td>$O(m + \text{occ} + (\log n)^k (k+1) \log \log n)$</td>
</tr>
<tr>
<td>[Cha+06](^4)</td>
<td>$O(n)$ words</td>
<td>$O(m + \text{occ} \cdot 3^k k^3 + (\log n)^k (k+1) \log \log n)$</td>
</tr>
<tr>
<td>ibid.(^4,5)</td>
<td>$O(n)$ bits</td>
<td>$O(m + (\text{occ} + (\log n)^k (k+1) \log \log n) \log^\varepsilon n)$</td>
</tr>
</tbody>
</table>

1 Given $q \leq n$.
2 Space requirement is $O(n \sqrt{\log n})$ words if no exact matches of the pattern occur in the text.
3 Given $\sigma = O(2\sqrt{\log n})$.
4 The effect of $\sigma$ is not shown.
5 Hamming distance only.
6 The first case applies when $\sigma = O(2\sqrt{\log n})$. 
Vishkin [LV89] to $O(nk)$ time with $O(m)$ space. A further improvement to the $k$-mismatch variant was made by Amir et al. [ALP04] who gave an $O(n\sqrt{k\log k})$ time solution.

In the offline version of the problem the text is processed to generate an indexing data structure. The first ones to study this variant were Jokinen and Ukkonen [JU91]. Furthermore, Ukkonen was the first who proposed a solution in which the query time is independent of the text length [Ukk93]. Since then, both the query time and space complexity of the indexing data structure have been improved.

The fastest known algorithm with an $O\left(\frac{n}{\sqrt{k}}(c \log n)^k\right) = O(n \log^k n)$ word indexing data structure by Cole et al. has a time complexity of $O(m + occ \cdot 3^k + \frac{1}{\sqrt{k}}(c' \log n)^k \log \log n)$ where $occ$ is the number of occurrences and $c$ and $c'$ are positive constants [CGL04]. Reducing the space complexity to linear, the fastest known algorithm with an $O(n)$ word indexing data structure by Chan et al. [Cha+06] has a time complexity of $O(m + occ \cdot 3^k k^3 + (c \log n)^{k(k+1)} \log \log n)$ where $c$ is a positive constant. Other recent results are summarized in Table 1 for the one-mismatch and one-difference problems and in Table 2 for the $k$-mismatch and $k$-difference problems.

Since the indexed texts, such as DNA and protein sequences, tend to be very long, they need to be compressed. While efficient queries are desired in practice, reducing the size of the data structure is also an important research objective.

1.3 My Contribution

This work describes the fast and space-efficient pattern matching algorithm by Lam et al. [LSW08]. Implementations for solving the $k$-difference problem with $O(\sigma m^k (k + \log \log n) + occ)$ time complexity and the auxiliary data structures required by the algorithm are also provided. These have been listed in Table 3 and Figure 1. The main purpose of the implementation is to test its speed in practice against other algorithms that apply heuristics.

To the author’s knowledge the algorithm has not been implemented earlier. Another consideration in favour of choosing this particular algorithm was perceived feasibility of implementing it. The algorithm is also required as a part of the algorithm by Chan et al. [Cha+06] and implementing it later was deemed possible.

1.4 Lam et al.’s algorithm for string matching with $k$ differences

Lam et al. [LSW08] describe an algorithm for string matching with one difference or mismatch. The algorithm is based on using a particular type of suffix tree that provides certain operations in constant time. The pattern to be matched with the text is modified with every possible type of error at
each text position and the suffix tree is queried with the resulting altered patterns. The query is limited to certain parts of the suffix tree by storing a sample of suitable suffix array positions into data structures that also provide fast queries. When at least one match is found and its suffix array index has been retrieved, the other matches may be located by checking the substrings at adjacent suffix array indices, since the array has all substrings of the original text in lexicographic order.

The algorithm is extended to the case of $k$ differences with a simple dynamic programming algorithm. This algorithm compares the unmodified pattern to each substring of the text found in the suffix tree. As the tree is traversed, the algorithm reports partial matches, that is, matches with $k - 1$ differences up to some pattern and text position. The one difference algorithm is then used to handle the final difference by using the found suffix tree branch, the text position and the suffix of the pattern as inputs.

1.5 The Described Algorithms and Data Structures

As the algorithm by Lam et al. has numerous requirements, a decision had to be made how detailed descriptions of these would be provided. The option chosen was to consider the prominence of each algorithm or data structure. The better-known algorithms and the ones the implementations of which were provided by others would be described only briefly. The more obscure ones and the ones that were implemented specifically for this work would have more comprehensive descriptions. Moreover, some proofs of the complexities of the algorithms have been slightly extended. We assume the RAM model of computation throughout the work, such that bitwise and arithmetic operations on words of length $\Theta(\log n)$ bits take constant time, where $n$ is the input size.

Chapter 2 contains an overview of the of the data structures and algorithms on which Lam, Sung and Wong’s algorithm is based and descriptions of some basic operations and data structures, as well as a table of notational conventions. The compressed suffix array and the compressed suffix tree upon which the algorithm is based are described in Chapter 3. Other required data structures are described in Chapter 4. Lam, Sung and Wong’s algorithm for pattern matching with one difference or mismatch is described in Chapter 5 and its extension to multiple differences or mismatches is shown in Chapter 6. Experiments and their results are discussed in Chapter 7. Final conclusions are presented in Chapter 8.
### Table 3: Algorithms and data structures referred to in this thesis.

<table>
<thead>
<tr>
<th>Data structure or algorithm</th>
<th>Implemented in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer and bit vectors</td>
<td>SDSL</td>
</tr>
<tr>
<td>Rank and select support</td>
<td>SDSL</td>
</tr>
<tr>
<td>Balanced parentheses support</td>
<td>SDSL</td>
</tr>
<tr>
<td>Sparse balanced parentheses support</td>
<td>This work</td>
</tr>
<tr>
<td>Range minimum query support</td>
<td>SDSL</td>
</tr>
<tr>
<td>Elias inventory</td>
<td>This work</td>
</tr>
<tr>
<td>Compressed suffix array</td>
<td>This work</td>
</tr>
<tr>
<td>Inverse suffix array</td>
<td>This work</td>
</tr>
<tr>
<td>Compressed suffix tree</td>
<td>SDSL</td>
</tr>
<tr>
<td>Perfect hashing</td>
<td>William Ahern’s PHF library</td>
</tr>
<tr>
<td>Binary search tree with the Eytzinger method</td>
<td>This work</td>
</tr>
<tr>
<td>X-fast trie</td>
<td>This work</td>
</tr>
<tr>
<td>Y-fast trie</td>
<td>This work</td>
</tr>
<tr>
<td>Algorithm for string matching with one difference</td>
<td>This work</td>
</tr>
<tr>
<td>Dynamic programming algorithm for string matching with $k - 1$ differences</td>
<td>This work</td>
</tr>
<tr>
<td>Lam et al.’s algorithm for string matching with $k$ differences</td>
<td>This work</td>
</tr>
</tbody>
</table>

2 Preliminaries

2.1 Required Data Structures and Algorithms

The pattern matching algorithm requires data structures for many different purposes. In the implementation, some existing library code was used when available and fit for use. In particular, many important data structures were provided by the SDSL library [GBMP14]. While an implementation of the required compressed suffix array was available from the Pizza & Chili Corpus\(^1\), its integration into SDSL was considered difficult and thus it was re-written.

The data structures and algorithms used in this thesis are summarized in Table 3. Their dependencies are shown in Figure 1.

\(^1\)http://pizzachili.dcc.uchile.cl/
Figure 1: Dependencies of the different algorithms and data structures. Integer and bit vectors have been omitted to increase clarity as they are used in most of the other data structures.
2.2 Notation

Below is a summary of the notation used in equations.

Table 4: Notation used in this thesis.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\min{a, b}$</td>
<td>The minimum of $a$ and $b$.</td>
</tr>
<tr>
<td>$\max{a, b}$</td>
<td>The maximum of $a$ and $b$.</td>
</tr>
<tr>
<td>$a + b$</td>
<td>The sum of $a$ and $b$.</td>
</tr>
<tr>
<td>$ab$ or $a \cdot b$</td>
<td>The product of $a$ and $b$ or, in case of strings, the concatenation of $a$ and $b$.</td>
</tr>
<tr>
<td>$f(\cdot)$</td>
<td>$f$ being a function, $\cdot$ denotes an unnamed variable.</td>
</tr>
<tr>
<td>$a \in B$</td>
<td>$a$ is an element of $B$.</td>
</tr>
<tr>
<td>$A \subseteq B$</td>
<td>$A$ is a subset or equal to $B$.</td>
</tr>
<tr>
<td>$A \setminus B$</td>
<td>The relative complement of $B$ in $A$, i.e. the set of elements in $A$ but not in $B$.</td>
</tr>
<tr>
<td>$</td>
<td>A</td>
</tr>
<tr>
<td>$</td>
<td>a</td>
</tr>
<tr>
<td>$</td>
<td>S</td>
</tr>
<tr>
<td>$A[i]$</td>
<td>$i$-th element in the totally ordered set or list $A$, or $i$-th character in the string $A$. Indices start from one.</td>
</tr>
<tr>
<td>$T[a..b]$</td>
<td>$T$ being a string, its substring between indices $a$ and $b$ inclusive. Indices start from one.</td>
</tr>
<tr>
<td>$E[i,j]$</td>
<td>$E$ being a matrix, the entry on the $i$-th row in the $j$-th column.</td>
</tr>
<tr>
<td>$\lfloor a \rfloor$</td>
<td>The integral part of $a$.</td>
</tr>
<tr>
<td>$\lceil a \rceil$</td>
<td>The smallest integer equal to or greater than $a$.</td>
</tr>
<tr>
<td>$a \equiv b \pmod{c}$</td>
<td>Integers $a$ and $b$ are congruent modulo $c$, i.e. $c$ divides $a - b$.</td>
</tr>
<tr>
<td>$f(n) = O(g(n))$</td>
<td>$f$ is bounded above by $g$, i.e. there are constants $k &gt; 0$ and $n_0$ such that for all $n &gt; n_0$, $</td>
</tr>
<tr>
<td>$f(n) = o(g(n))$</td>
<td>$f$ is dominated by $g$ asymptotically, i.e. for all constants $k &gt; 0$ there is $n_0$ such that for all $n &gt; n_0$, $</td>
</tr>
</tbody>
</table>

2.3 Rank and Select

Given a bit vector $V$ of length $n$, there are two classes of required operations: $\text{rank}_x(i)$ retrieves the number of instances of the bit pattern $x$ up to and including position $i$; its inverse operation, $\text{select}_x(i)$, retrieves the starting
position of the $i$-th occurrence of the bit pattern $x$.

The queries may be answered by solving the indexable dictionary problem, in which a set of integers $S \subseteq \{0, ..., m - 1\}$ is queried for the number of elements lesser than a given element or the $i$-th greatest element for some $i$. Suppose $n - 1$ is the maximum of the indexed integers and $S$ contains all the indices from which a certain bit pattern begins. The following result has been achieved.

**Lemma 1.** [RRR02] There is a data structure that implements the operations $\text{rank}_x$ and $\text{select}_x$ in constant time while requiring extra $o(n)$ bits of space.

### 2.4 Balanced Parentheses Support

Strings of balanced parentheses, also known as the Dyck language, are strings that have the characters “(” and “)” properly nested. Here, proper nesting refers to the property that, if $\text{imbalance}$ is the difference of the counts of opening and closing parentheses, then imbalance is non-negative for all the prefixes of the string and zero for the whole string. Such strings have various uses, such as representing a tree topology. For this and other purposes, some operations are required.

- **open($i$):** Finds the matching opening parenthesis for the closing parenthesis at position $i$ and reports its index.
- **close($i$):** Finds the matching closing parenthesis for the opening parenthesis at position $i$ and reports its index.
- **enclose($i$):** Finds the opening parenthesis of the pair that most tightly encloses $i$.

**Lemma 2.** [GRRR06] There is a data structure that provides the operations for balanced parentheses in constant time with a space requirement of additional $o(n)$ bits.

A typical representation of the balanced parentheses strings is to use ones for opening parentheses and zeros for closing parentheses. As a result of this, $\text{rank}$ and $\text{select}$ may also be used on the strings.

### 2.5 Range Minimum Queries

Given an integer array $A$ of length $n$, the range minimum query for indices $i$ and $j$ retrieves the index of the minimum array value between indices $i$ and $j$ inclusive. The problem has received attention to the extent of finding a solution that has constant query time with $O(nH_k) + o(n)$ space requirement in bits on top of the space requirement of $A$ [FH11]. Here, $H_k$ denotes the
k-th order empirical entropy of the input array. Hence, the following result is attained.

Lemma 3. [FH11] There is a data structure that retrieves the solution to range minimum queries in constant time using additional $O(nH_k) + o(n)$ bits of space. In case the input array is only available at construction time, the space requirement is $2n + o(n)$ bits.

2.6 Eytzinger Method

A binary search tree may be stored into a flat array containing only the values with a simple numbering system known as the Eytzinger method or Ahnentafel [Ait90]. The same method is utilized in a related data structure, the binary heap. In particular, pointers to child nodes are not needed. Suppose the root node is located at array index 1. Given an array index $i$ of some binary tree node, its left descendant will be located at index $2i$ and its right descendant will be located at index $2i + 1$.

This allows the parent and child nodes to be accessed in constant time, given the array index of any node. If the binary tree is balanced and its every level except for the last one is completely filled, space is required for at most $2n − 1 = O(n)$ nodes, the worst case being that given $l$ levels, every node except the rightmost one on the $l − 1$-th level is a leaf node.

3 Suffix Trees and Arrays

3.1 Suffix Tree

The suffix tree [Wei73] is a trie-like data structure that represents all suffixes of a text. Given a text $T$ of length $n$ drawn from alphabet $\Sigma$, the suffix tree of $T$ consists of $n$ leaves. Any path from the root to a leaf spells out a suffix of $T\$, i.e. $T$ concatenated with the special character $\$ that does not appear in $\Sigma$. Each internal node of the tree has at least two children. Each concatenation of edges that leads to a node $\bar{v}$ is labelled with a nonempty substring of $T$. This is called the path label of $\bar{v}$. The labels of any two out-edges of a node may not begin with the same character.

Since every internal node of a suffix tree is branching, there can be at most $n − 1$ internal nodes in a suffix tree. Therefore the total count of nodes is at most $n + n − 1 = 2n − 1$.

In addition to edges, nodes of the suffix tree may be connected with suffix links. If the path from root to node $\bar{u}$ spells out the string $x\alpha$ where $\alpha$ is a string and $x$ is a single character, there is a suffix link from $\bar{u}$ to a node the path label of which is the string $\alpha$. Every internal node in the suffix tree that has a path label longer than one has an outgoing suffix link.

The nodes of the suffix tree are numbered by traversing the tree in pre-order and assigning consecutive positive integers starting from zero.
An example suffix tree is shown in Figure 2 and some of its properties are listed in Table 5. The concepts are described in this chapter and the following chapters.

### 3.1.1 Required Operations

For the purposes of the approximate pattern matching algorithm by Lam et al., the suffix tree needs to support the following operations. The variables $t_{SA}$ and $t_\Psi$ depend on the design of the suffix tree and $\sigma$ is the size of the alphabet.

- $label(\bar{u}, \bar{v})$: returns the label on the edge that joins nodes $\bar{u}$ and $\bar{v}$ in $O(lt_{SA})$ time where $l$ is the length of the edge label.
- $plen(\bar{v})$: returns the string depth of $\bar{v}$, i.e. the length of the path label $label(\bar{r}, \bar{v})$ in $O(t_{SA})$ time where $\bar{r}$ is the root node.
- $lb(\bar{v})$: returns the left bound or the smallest index of the suffix array range of the subtree rooted at node $\bar{v}$ in $O(1)$ time.
- $rb(\bar{v})$: returns the right bound or the greatest index of the suffix array range of the subtree rooted at node $\bar{v}$ in $O(1)$ time.
- $slink(\bar{v})$: returns the target node of the suffix link that starts from node $\bar{v}$ in $O(t_\Psi)$ time.
- $child(\bar{v}, c)$: returns the child node of $\bar{v}$ the edge label of which starts with the character $c$ in $O(\sigma t_{SA})$ time.

### 3.1.2 Representing the Suffix Tree with a Suffix Array and Auxiliary Data Structures

While the space requirement of suffix trees is linear, there was a large constant factor involved that used to limit the practicality of the data structure. For a text of length $n$, the suffix tree could require over $15n$ bytes [Kur99].

<table>
<thead>
<tr>
<th>Node number</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Label</td>
<td>a</td>
<td>$</td>
<td>ca</td>
<td>$</td>
<td>ttc$a</td>
<td>ttc$a</td>
<td>$</td>
<td>ttc$a</td>
</tr>
<tr>
<td>Parentheses</td>
<td>(</td>
<td>)</td>
<td>(</td>
<td>)</td>
<td>( )</td>
<td>(</td>
<td>)</td>
<td>( )</td>
</tr>
</tbody>
</table>

Table 5: Balanced parentheses representation and node numbers of the subtree of the suffix tree in Figure 2 that begins with $a$. The second table row indicates the label of the edge that points to the node in question, and the third row indicates the opening parenthesis associated with the node. (See Section 3.8.) The root node, which is not shown, has number 1.
Figure 2: The suffix tree for the string \textit{gattacattaca}. Suffix links are shown dashed and the numbers on the leaf nodes indicate suffix starting positions in the text. One possible heavy path decomposition is also shown with the thicker lines denoting core edges. (See Section 5.1.)
However, with some additional data structures, suffix tree traversals may be simulated with suffix arrays with the same time and space complexity. In this case they have been referred to as enhanced suffix arrays. In terms of memory use, this has been an efficient approach [AKO04].

3.2 Suffix Array, Inverse Suffix Array and $\Psi^k$ function

Like suffix tree, the suffix array [MM93] represents all suffixes of a text but without the tree structure. Given a text $T$ of length $n$, the suffix array $SA$ of $T$ is a permutation of indices from 1 to $n$. The indices correspond to the starting positions of the suffixes of $T$ sorted in lexicographical order.

The inverse suffix array $SA^{-1}$ of $T$ is similarly a permutation of integers. $SA^{-1}[i]$ equals the number of suffixes that are lexicographically smaller than $T[i..n]$. For indices $i$ and $j$ it holds that $SA[i] = j \iff SA^{-1}[j] = i$.

For retrieving the next suffixes in $SA$ with respect to text position, a group of functions is defined [GV05, Rao02]:

$$
\Psi^k(i) = \begin{cases} 
    j, & \text{if } SA[j] = SA[i] + k \\
    0, & \text{if } SA[i] + k > n
\end{cases}
$$

3.2.1 Required Operations

Suppose $t_{SA}$ and $t_{\Psi}$ are the access times of each entry on $SA$ and $\Psi^1$ respectively. For the purposes of the approximate pattern matching algorithm, data structures with the following properties are needed:

- Report $SA[i]$ in $t_{SA}$ time.
- Report $SA^{-1}[i]$ in $t_{SA}$ time.
- Report $\Psi^1(i)$ in $t_{\Psi}$ time.
- Report substring$(i, l) = T[SA[i]..SA[i] + l - 1]$ in $O(l t_{\Psi})$ time for any valid length $l$.

3.2.2 Space Requirements

The space requirement of suffix arrays is less than that of suffix trees but can still be considerable. Since the suffix array needs to store all text positions from 1 to $n$ for a text of length $n$, its space requirement is $O(n \log n)$ bits while the original uncompressed text requires $O(n \log \sigma)$ bits, $\sigma$ being the alphabet size. For applications that use a small alphabet but long sequences, such as DNA and protein sequence indexing, the space requirement could be substantial. For this purpose, suffix array compression methods have been studied (e.g. [Mäk00, Sad03, MN05, GV05, FM05]).
3.3 Constant Access Time for the Compressed Suffix Array

A compressed suffix array that supports lookup queries in double logarithmic [GV05] or constant time [Rao02] exists with space complexities of \( O(n \log \log n \log \sigma) \) and \( O(n \log^\varepsilon n \log \sigma) \) bits for any fixed positive constant \( \varepsilon \leq 1 \) respectively. Given a text \( T \) of length \( n \) drawn from alphabet of size \( \sigma \), the permutation of its suffix indices is represented as follows:

1. Given a parameter \( \ell \) which is also a factor of \( n \), the values in \( SA \) that are multiples of \( \ell \) are stored in another array \( SA_1 \) after dividing them by \( \ell \). This takes \( \frac{n}{\ell} \lfloor \log \frac{n}{\ell} \rfloor \) bits.

2. A bit vector \( B \) of length \( n \) indicates which suffix array values were stored in \( SA_1 \). If \( SA[i] \) was stored in \( SA_1 \), \( B[i] \) is set to 1 and otherwise to 0. Additionally \( rank_1 \) support is needed for the bit vector, which by Lemma 1 requires extra \( o(n) \) bits of space.

3. An array \( d \) of length \( n \) stores the difference of \( SA[i] \) and the next multiple of \( \ell \), i.e. \( d[i] = \ell - (SA[i] \mod \ell) \). The array takes \( n \lfloor \log \ell \rfloor \) bits.

4. For each \( k \), \( 1 \leq k \leq \ell - 1 \), the compressed representation of the subsequence \( \psi^k = \{\Psi^k(i) \mid d[i] = k \text{ and } 1 \leq i \leq n\} \) is stored.

\( SA[i] \) may be recovered from the compressed representation with the following steps: Let \( k = d[i] \). Recover \( \Psi^k(i) \) and find its rank \( r \) from the \( rank_1 \) support of \( B \). Now \( SA[i] = \ell \cdot SA_1[r] - k \).

Additionally the same compression method may be applied recursively to \( SA_1 \).

3.3.1 Compact Representation of the \( \Psi^k \) values

The suffix array compression method is based on the observation that the \( \Psi^k \) values may be partitioned into \( \sigma^k \) monotone consecutive lists. Since each list is monotone, it may be treated as a set. After partitioning the function values into sets, a compression method that makes use of monotonicity may be applied.

**Lemma 4.** [GV05, Rao02] Suppose \( z_k(i) \) is the value in base-\( \sigma \) of the \( k \) symbols appearing before \( SA[\Psi^k(i)] \). The sequence \( S = \{\Psi^k(i) \mid 1 \leq i \leq n\} \) is the concatenation of \( \sigma^k \) monotone lists \( L_0, L_1, ..., L_{\sigma^k-1} \) where \( L_j = \{\Psi^k(i) \mid z_k(i) = j\} \).

Similarly, if \( t \) is the recursion level, \( k\ell^{t-1} \) symbols are considered and there are \( \sigma^{k\ell^{t-1}} \) monotone lists.

**Proof.** It needs to be shown that each of the \( L_j \) lists is monotone and that \( S \) is a concatenation of the lists.
Each of the $L_j$ lists is monotone. It needs to be shown that given suffix array indices $i$ and $j$ where $i < j$, if the suffixes starting at positions $SA[Ψ^k(i)]$ and $SA[Ψ^k(j)]$ have the same $k$ symbols preceding them, then $Ψ^k(i) < Ψ^k(j)$.

Suppose not, i.e. for some $i_0 < j_0$ the suffixes starting at positions $SA[Ψ^k(i_0)]$ and $SA[Ψ^k(j_0)]$ have the same $k$ symbols preceding them but it holds that $Ψ^k(i_0) > Ψ^k(j_0)$. Since $SA[Ψ^k(i)] = SA[i] + k$ unless $SA[i] + k > n$, suffixes starting at positions $SA[i_0] + k$ and $SA[j_0] + k$ have the same $k$ characters preceding them. This gives the following inequality.

\[
Ψ^k(i_0) > Ψ^k(j_0)
\]

\[
SA^{-1}[SA[i_0] + k] > SA^{-1}[SA[j_0] + k]
\]

\[
SA^{-1}[SA[i_0 + k]] > SA^{-1}[SA[j_0 + k]] \quad \text{Same } k \text{ characters precede both positions.}
\]

\[
i_0 + k > j_0 + k
\]

\[
i_0 > j_0
\]

However, this is a contradiction. Hence, given the original assumptions, it must hold that $Ψ^k(i) < Ψ^k(j)$.

**S is a concatenation of the lists.** In essence each element in $L_i$ appears before any element in $L_j$ in $ψ^k$ given $i < j$. Suppose $a$ and $b$ are suffix array indices and $Ψ^k(a) ∈ L_i$ and $Ψ^k(b) ∈ L_j$ for some list indices $i ≠ j$. It needs to be shown that $a < b$ implies $i < j$.

Since $a < b$, the suffix at index $a$ must be lexicographically smaller than the one at position $b$. Since $i ≠ j$, the value of the $k$ symbols that appear before the suffix starting at position $SA[Ψ^k(a)]$ must be less than that of those $k$ symbols that appear before the suffix starting at position $SA[Ψ^k(b)]$.

As a result, it must hold that $i < j$.

**Corollary 5.** As the sequence $S$ is a concatenation of sorted lists, so are all of its subsequences.

It remains to store the sorted subsequences compactly. To this end, a variant of Elias inventories is applied.

**Lemma 6.** [Eli74, GV05, Rao02] Given a set $S$ of $m$ non-negative integers (in ascending order) each containing $w$ bits and $m ≤ 2^w - 1$, $S$ may be represented with at most $m(3 + w - \lfloor \log m \rfloor) + o(m)$ or $m(2 + w - \lfloor \log m \rfloor) + o(m)$ bits so that retrieving any integer takes constant time.

**Proof.** Of each integer in $S$ the most significant $z = \lfloor \log m \rfloor$ bits are taken. Suppose $q_1, ..., q_m$ are the integers, called quotients, so obtained. Similarly, suppose $r_1, ..., r_m$ are the integers, called remainders, retrieved by taking the $w - z$ least significant bits.
The quotients are placed in list $Q$ by storing the unary representations of their differences. In other words, $Q = \{q_1, q_2 - q_1, ..., q_m - q_{m-1}\}$ and the representation of integer $i$ is the bit string $0^i1$, i.e. $i$ copies of zero followed by one. Since there are $m$ integers and each quotient has $z$ bits, $Q$ requires $m + 2^z = m + 2^{\lceil \log m \rceil} < m + 2^{1+\log m} = 3m$ bits. To retrieve the $i$-th value in $Q$, which is equal to the sum of the first $i$ items, select$_1$ support is used, which by Lemma 1 requires additional $o(m)$ bits.

The remainders are placed uncompressed in list $R = \{r_1, r_2, ..., r_m\}$. The space requirement is $m(w-z) = m(w-\lceil \log \frac{m}{m} \rceil)$ bits. This results in the total space requirement of $3m + m(w-\lceil \log m \rceil) + o(m) = m(3+w-\lceil \log m \rceil) + o(m)$ bits.

To retrieve the $i$-th original integer, the value $q_i$ is first retrieved, which takes $O(1)$ time. Then $q_i$ concatenated with $r_i$, which may be done with arithmetic left shift and bitwise OR. The time requirement is thus constant.

On the other hand, by choosing $z = \lfloor \log m \rfloor$, the space bound of $m(2+w-\lfloor \log \frac{m}{m} \rfloor) + o(m)$ may be achieved.

**Corollary 7.** [Eli74, GV05, Rao02] Given a set $S$ of $m$ integers from $\{1, ..., M\}$ such that $m \leq M$, it may be represented with $m(3+\lceil \log \frac{M}{m} \rceil) + o(m)$ bits so that retrieving any integer takes constant time.

**Proof.** Substituting $w$ in Lemma 6 yields

\[ m(3 + w - \lceil \log m \rceil) + o(m) \]
\[ = m(3 + \lceil \log M \rceil - \lceil \log m \rceil) + o(m) \]
\[ \leq m(3 + \lceil \log \frac{M}{m} \rceil) + o(m). \]

**Corollary 8.** [Eli74, GV05, Rao02] Suppose $S_1, S_2, ..., S_l$ are non-empty subsets of $\{0, ..., M-1\}$ such that $m$ is the sum of the counts of their items, $m \leq M$ and $M$ is a power of two. The sets may be represented with $m(3+\lceil \log \frac{M}{m} \rceil) + o(m)$ bits so that retrieving any integer takes constant time given its index in the concatenation.

**Proof.** Store the set $S = \{i \cdot M + x \mid x \in S_i, i \in \{0, ..., l\}\}$ using the representation of Corollary 7 which uses $m(3 + \lceil \log \frac{M}{m} \rceil) + o(m)$ bits of space. Also store the value $M-1$ which takes $O(1)$ bits of space. The $i \cdot M$ summand may be removed from each item with bitwise AND.

**Remark.** If the index of each integer in its subsequence rather than in the concatenation were known, additional data structures would be needed. For each index its subsequence number would be stored, which would require additional $m \log l$ bits. To determine the count of the preceding elements in the concatenation, a cumulative sum of the lengths of the subsequences
would be stored in additional \( l \log m \) bits [Rao02]. However, this is not required to store \( \psi^k \) since its values are always accessed by index in the whole sequence.

To store the subsequence \( \psi^k \), the following steps are taken:

- Store a bit vector \( V_k \) of \( n \) items such that \( V_k[i] = 1 \) if \( d[i] = k \) and 0 otherwise. Store \( \text{rank}_1 \) support for the vector.
- Store the lists \( L_0, L_1, \ldots, L_{\sigma_k - 1} \) (or, if \( t \) is the recursion level, \( L_{\sigma_k \ell^{t-1} - 1} \)) using the representation of Corollary 8 into list \( \mathcal{L}_k \).

Given \( i \) and \( k = d[i] \), \( \Psi^k(i) \) may be retrieved by finding the rank \( r \) of \( i \) in \( V_k \) using its \( \text{rank}_1 \) support. Now \( r \) is the index of \( \Psi^k(i) \) in \( \mathcal{L} \).

There are \( \frac{n}{\ell} \) items in \( \psi^k \) split into \( \sigma^k \) sorted lists. Storing \( V_k \) and its \( \text{rank}_1 \) support requires \( n + o(n) \) bits by Lemma 1. Since \( \max \psi^k \leq n \), by Corollary 8 \( \mathcal{L}_k \) requires \( \frac{n}{\ell} (3 + \lceil \log \ell \sigma^k \rceil) + o(n) = \frac{n}{\ell} (3 + \lceil \log \ell + k \log \sigma \rceil) + o(n) \) bits of space.

### 3.3.2 Applying Recursion

Using this representation, the \( V_k \) and \( \mathcal{L}_k \) lists have \( O(n \ell + n + n \log \ell + n \ell \log \sigma) = O(n \ell \log \sigma) \) space complexity in bits for any parameter value \( 1 \leq \ell \leq \log n \). \( S A_1 \), \( B \) and \( d \) have lesser space requirements.

Suppose \( t \) is the recursion level. Now multiples of \( \ell \) in \( SA_{t-1} \) are stored in \( SA_t \) and \( \psi^k \) are constructed for the latter array. Each \( \psi^k \) contains \( \frac{n}{\ell^t} \) entries and, by Lemma 4, splits into \( \sigma^{k \ell^{t-1}} \) sorted lists. The space requirement in bits to store \( \psi^k \) on a given level is calculated as follows by Corollary 8.

\[
\frac{n}{\ell^t} (3 + \lceil \log \ell \sigma^{k \ell^{t-1}} \rceil) + o\left(\frac{n}{\ell^t}\right) \\
= \frac{n}{\ell^t} (3 + \lceil \log \ell^t + k \ell^{t-1} \log \sigma \rceil) + o\left(\frac{n}{\ell^t}\right)
\]

Since there are always \( \ell \) lists, combined on a given level they have the following space requirement in bits.

\[
\frac{n}{\ell^{t-1}} (3 + \lceil \log \ell^t + \ell^t \log \sigma \rceil) + o\left(\frac{n}{\ell^{t-1}}\right)
\]

Solving the corresponding space complexities of the terms one by one gives the following results.

\[
\frac{3n}{\ell^{t-1}} = O(n) \\
\frac{n \log \ell^t}{\ell^{t-1}} = O(n) \\
\frac{n \ell^t \log \sigma}{\ell^{t-1}} = O(n \ell \log \sigma)
\]
Hence the total space complexity for one recursion level becomes $O(nt\ell \log \sigma)$. Suppose $t$ is the number of recursion levels. The space complexity for all of them combined is therefore $O(nt\ell \log \sigma)$.

The values in $SA_t$ are stored without further compression. Since there are $\frac{n}{t\ell}$ entries left, this requires $\frac{n}{t\ell} \log \frac{n}{t\ell}$ bits. Choosing $\ell = \log \frac{1}{t\ell} n$ yields the following results.

**Theorem 9.** [Rao02] The suffix array may be compressed so that it supports lookup queries in $O(t)$ time with space complexity $O(nt\ell \frac{1}{t\ell} n \log \sigma)$ in bits for any parameter value $1 \leq t \leq \log \log n - 1$.

**Corollary 10.** [GV05] The suffix array may be compressed so that it supports lookup queries in $O(\log \log n)$ time with space complexity $O(n \log \log n \log \sigma)$ bits.

**Proof.** Choosing $t = \log \log n - 1$ in Theorem 9 yields the following space complexity.

\[
\begin{align*}
nt \log \frac{1}{t\ell} n \log \sigma \\
= (n \log \sigma)(t \log \frac{1}{t\ell} n) \\
= (n \log \sigma)(\log \log n - 1)(\log \frac{1}{\log \log n} n) \\
= (n \log \sigma)(\log \log n - 1)((2^{\log \log n} \frac{1}{\log \log n}) \\
= 2(n \log \sigma)(\log \log n - 1) \\
= O(n \log \log n \log \sigma)
\end{align*}
\]

**Corollary 11.** [Rao02] The suffix array may be compressed so that it supports lookup queries in constant time with space complexity $O(n \log^\varepsilon n \log \sigma)$ for any fixed positive constant $\varepsilon \leq 1$.

**Proof.** The result is achieved by choosing $t = \frac{1}{\varepsilon}$ in Theorem 9.

3.4 Constant Access Time for the Inverse Suffix Array

The $\Psi^k$ function is used to implement the inverse suffix array as well. Unlike in the case of the suffix array, however, its values are sampled from the same indices as the uncompressed inverse suffix array values.

**Lemma 12.** [LSW08] Suppose $S_1, \ldots, S_l$ are $l$ subsets of $\{0, \ldots, n - 1\}$ such that there are $\frac{n}{l}$ items in each and $S$ is their concatenation. Then $\psi^k(i) = \{\Psi^k(i) \mid i \in S_k\}$ for all $1 \leq k \leq l$ may be stored in $O(n \log \sigma)$ bits, $\sigma$ being the alphabet size. Given $i$ where $S_k[i] = z$, $\Psi^k(z)$ may be accessed in constant time.
Proof. By Lemma 4, \( \psi^k \) contains at most \( \sigma^k \) sorted lists for any given \( k \). By Corollary 8, \( \psi^k \) may be represented in \( |S_k| (3 + \lceil \log \frac{\sigma^k}{|S_k|} \rceil) + o(|S_k|) = \frac{n}{l} (3 + \lceil \log (l \sigma^k) \rceil) + o(|S_k|) \) = \( O(n) \) bits. Since there are \( l \) subsets \( S_1, \ldots, S_l \), they require \( O(n l \log \sigma) \) bits in total. The data structure in question supports constant time access.

\[ \text{Theorem 13.} \ [\text{LSW08}] \text{ An inverse suffix array that reports } SA^{-1} \text{ values in } O(1) \text{ time and requires } O(n \sqrt{\log n} \log \sigma) \text{ bits exists, } n \text{ being the text length and } \sigma \text{ the alphabet size, given that } \sigma = O(2^{\sqrt{\log n}}). \]

Proof. Suppose \( l = \sqrt{\log n} \). A sample of \( SA^{-1} \) is stored by taking the values at indices that are multiples of \( l \), i.e. \( SA^{-1}[x] \) for all \( 0 \leq x \leq \lfloor \frac{n}{l} \rfloor \), which requires \( O(n \sqrt{\log n}) \) bits. Additionally, \( \Psi^k(i) \) values are stored for all \( 1 \leq k \leq l \) and \( 0 \leq i \leq \lfloor \frac{n}{l} \rfloor \). By Lemma 12 the data structure may be stored in \( O(n \sqrt{\log n} \log \sigma) \) bits given that \( \sigma = O(2^{\sqrt{\log n}}) \).

To report \( SA^{-1}[i] \) in constant time, suppose \( y = \lfloor \frac{i}{l} \rfloor \), \( k' = i - yl \) and \( z' = SA^{-1}[yl] \). Now \( SA^{-1}[i] = \Psi^{k'}(z') \) and \( k' \leq l \). Since the values \( z' \), for which \( \Psi^{k'}(z') \) are stored, are not guaranteed to be consecutive, the index of \( \Psi^{k'}(z') \) needs to be determined by using a bit vector of length \( n \) with rank support; the index in the compressed sequence is the rank of \( z' \). The space complexity is not affected since the bit vector and the rank support only require additional \( O(n) \) bits of space. This is achieved by using the bit vector and rank support for \( \Psi^1 \) and storing placeholder values into the Elias inventory where \( \Psi^{k'} \) have zero values where \( \Psi^1 \) does not. By choosing the preceding value as the placeholder value, each requires a single additional bit. 

\[ \text{3.5 Supporting Data Structures for Retrieving the Original Text} \]

In order to support substring, the inverse of the cumulative counts of the characters in the text \( T \) of length \( n \) need to be accessed. To this end, the function \( C^{-1} \) is defined as follows.

\[ C^{-1}(j) = T[SA[j]], 1 \leq j \leq n \]

In other words, \( C^{-1}(j) \) returns the \( j \)-th character of the text in suffix array order.

\[ \text{Lemma 14.} \ [\text{Sad00}] \text{ The values of } C^{-1} \text{ may be retrieved in constant time by using } O(n) \text{ bits for storage given } \sigma \log \sigma = O(n). \]

Proof. A bit vector \( D \) of length \( n \) is stored as follows.

\[ D[j] = \begin{cases} 1, & T[SA[j]] \neq T[SA[j-1]], \\ 0, & \text{otherwise} \end{cases} \]
Additionally, rank$_1$ support is stored for $D$. By Lemma 1, these require $n + o(n)$ bits of space together. Now rank$_1(j)$ on $D$ takes constant time and represents the number of distinct characters in $T[SA[1]], T[SA[2]], \ldots, T[SA[j]]$ minus one. Since rank$_1(j)$ returns $\sigma$ different values, another vector of size $\sigma \log \sigma$ bits may be used to store the corresponding character.

3.6 Suffix Array Operations

With the suffix array, the inverse suffix array and $C^{-1}$, the remaining operations may be implemented.

3.6.1 Support for $\Psi^1$

Both the suffix array and the inverse suffix array store a sample of the $\Psi^k$ function values. These are not used directly, though, since both data structures support access in constant time by Corollary 11 and Theorem 13. As $\Psi^1(i) = SA^{-1}[SA[i] + 1]$, also the $\Psi^1$ values may be accessed in constant time. Similarly, values for $\Psi^k$ may be accessed in constant time for any given $l$.

3.6.2 Support for substring

In a manner similar to $\Psi^1$ support, the suffix array, the inverse suffix array and $C^{-1}$ may be used to report substrings in the original text $T$. Suppose the characters beginning at text index $i$ are to be retrieved. To this end, the corresponding suffix array index $j_1 = SA^{-1}[i]$ is first retrieved. By Theorem 13 this may be done in constant time. The corresponding character may then be obtained with $C^{-1}(j_1)$ in constant time by Lemma 14. The next suffix array index and character may be retrieved with $j_2 = \Psi^1(j_1)$ and $C^{-1}(j_2)$. This may be continued to retrieve a substring of length $l$ in $O(l \Psi) = O(l)$ time.

3.7 Longest Common Prefixes

The longest common prefix (LCP) array [MM93] is a data structure that may be used with a suffix array. At index $i$, it stores the length of the longest common prefix of entries at suffix array indices $i$ and $i - 1$.

3.7.1 Range Minimum Queries for Longest Common Prefixes

While the longest common prefix array is created for use with the suffix tree, it only contains the prefix lengths with respect to the previous suffix array entry. With added range minimum query support, the following result is attained.
Lemma 15. The LCP array may be queried for the longest common prefix length in a constant time with a requirement of additional $O(n)$ bits of space.

Proof. The proof follows directly from Lemma 3. \qed

3.8 Representing the Suffix Tree

There is a compressed suffix tree that requires $|SA| + 6n + o(n)$ bits of space [Sad07]. The compressed representation is based on a compressed representation of the longest common prefix array, range minimum queries and a balanced parentheses representation of the tree topology.

The balanced parentheses string may be constructed as follows. The suffix tree is traversed in pre-order and an opening parenthesis is written to the string. When all subtrees of a node have been traversed, a closing parenthesis is written. For example, all leaf nodes are represented with the string "()". A subtree, on the other hand, is represented with a substring that starts with an opening parenthesis and ends with a closing parenthesis. All parentheses between these are balanced. (See Table 5 and Figure 2 for an example.)

3.8.1 Suffix Tree Underpinning

The particular compressed suffix tree has three operations upon which the other required operations are built. The balanced parentheses representation of the tree topology $B$ is encoded so that ones represent opening parentheses and zeros closing parentheses. Since there are at most $2n - 1$ nodes in a suffix tree, $B$ requires at most $4n = O(n)$ bits of space. The representation is used in various ways, which also require support for $rank_1$, $rank_{10}$, $select_1$, $select_{10}$ and $select_{01}$. Together these require additional $o(n)$ bits of space by Lemma 1.

Node index in $B$. The index $v$ of a node $\bar{v}$ in the balanced parentheses representation may be retrieved with $select_1$ on $B$. Similarly, $v$ may be converted to $\bar{v}$ with $select_1$. These operations require constant time.

\[
\begin{align*}
v & = B.select_1(\bar{v}) \\
\bar{v} & = B.rank_1(v)
\end{align*}
\]

inorder($\bar{u}$). Given an internal node $\bar{u}$, inorder($\bar{u}$) gives its smallest inorder rank, i.e. the smallest number of visited internal nodes including $\bar{u}$ in depth-first traversal. The inorder rank $i$ of $\bar{u}$ may be retrieved in constant time similar to the inverse operation by using the balanced parentheses representation of the suffix tree structure as follows. Suppose $u$ is the index of $\bar{u}$
in $B$.

$$i = B.\text{rank}_{10}(B.\text{close}(u + 1))$$

$$u = B.\text{enclose}(B.\text{select}_{01}(i) + 1)$$

**$Hgt(i)$**. The values for the height array $Hgt$ may be calculated from the longest common prefix array as follows. Suppose $i$ is a suffix array index and $n$ is the text length.

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

The values of $Hgt$ are not sorted as such but adding the corresponding suffix array value to each and sorting the sums in the suffix array order gives a sorted sequence. This is based on the following observation. Suppose $p = SA^{-1}[1]$.

$$Hgt[p] \leq Hgt[\Psi^1(p)] + 1$$

$$\leq Hgt[\Psi^k(p)] + k$$

$$\leq Hgt[\Psi^{n-1}(p)] + n - 1$$

Since there is only one sorted sequence, it may be stored in an Elias inventory in $2n + o(n) = O(n)$ bits by Lemma 6 with $O(1)$ time complexity for access.

Accessing the original value at index $i$ is done by calculating $j = SA[i] - 1$, extracting the $j$-th entry $q$ from the inventory and subtracting $q - j$. This is the value of $Hgt[i]$. Because of the suffix array access this requires $O(t_{SA})$ time.

**$lca(\bar{v}, \bar{w})$**. Additionally, the lowest common ancestor $\bar{u}$ of two nodes $\bar{v}$ and $\bar{w}$ often needs to be retrieved. This is done with an array $L$, which is the depth sequence of $B$ and defined as follows. Suppose $v$ and $w$ are the positions of $\bar{v}$ and $\bar{w}$ in $B$ respectively.

$$L[i] = B.\text{rank}_1(i) - B.\text{rank}_0(i)$$

$$u = B.\text{enclose}(L.\text{RMQ}(v, w) + 1)$$

Since each element of $L$ may be calculated in constant time from $B$, $L$ itself need not be stored. By Lemma 3, the range minimum query may be answered in constant time with a space requirement of $2n + o(n) = O(n)$ bits.
3.8.2 Suffix Tree Operations with Suffix Array, LCP and Balanced Parentheses Support

It remains to be shown how to use the supporting data structures with the suffix array to implement the suffix tree operations. Rather than giving a detailed explanation, the general idea is described.

\textbf{parent}(\bar{u}). The parent of a given node may be found by performing \textit{enclose}(u). With converting \bar{u} to its index \(u\) in \(B\) and the resulting index back to a node number this requires \(O(1)\) time.

\textbf{label}(\bar{u}, \bar{v}). The label on the edge that joins nodes \(\bar{u}\) and \(\bar{v}\) is \(T[SA[i] + \text{Hgt}(j) \ldots SA[i] + \text{Hgt}(i) - 1]\) with \(i = \text{inorder}(\bar{v})\) and \(j = \text{inorder}(\bar{u})\). Hence the time requirement is \(O(lt_{SA})\), \(l\) being the edge label length.

\textbf{plen}(\bar{v}). The length of the path label of node \(\bar{v}\) is equal to \(\text{Hgt}(\text{inorder}(\bar{v}))\) which takes constant time to retrieve.

\textbf{lb}(\bar{v}) and \textbf{rb}(\bar{v}). After retrieving the index \(v\) of node \(\bar{v}\) in \(B\), the index of the left bound in \(B\) may be found with \(1 + B.\text{rank}_{10}(v)\), since all the leaf nodes are in suffix array order. Similarly, the index of the right bound may be found with \(B.\text{rank}_{10}(B.\text{close}(v)) - 1\). Both operations take constant time.

\textbf{slink}(\bar{v}). The suffix link target node \(\bar{w}\) for node \(\bar{v}\) may be calculated as follows. Again the index \(v\) of \(\bar{v}\) in \(B\) is first located.

\[
x' = \Psi_1(B.\text{rank}_{10}(v - 1) + 1)
\]
\[
y' = \Psi_1(B.\text{rank}_{10}(B.\text{close}(v)))
\]
\[
w = \text{lca}(B.\text{select}_{10}(x'), B.\text{select}_{10}(y'))
\]

This takes \(O(t_{\Psi})\) time.

\textbf{child}(\bar{u}, c). The first child \(\bar{v}\) of node \(\bar{u}\) may be found by adding one to its index in \(B\). The next sibling may then be found with \(1 + B.\text{close}(v)\), \(v\) being the index of \(\bar{v}\). After each step the first character of the label of the edge may be retrieved in the same manner as for \textit{label} but only once, which then results in \(O(\sigma t_{SA})\) time complexity.

Another option is to use binary search, which, in like manner to \textit{lca}, requires constant time range minimum query support on the depth sequence \(L\). This results in \(O(\log \sigma t_{SA})\) time complexity.
3.8.3 Polylogarithmic Suffix Array Access Time as an Alternative

Grossi and Vitter describe yet another compressed suffix array variant the properties of which allow for $t_{SA} = O(\log^\varepsilon n)$ and $t_{\Psi} = O(1)$ [GV05]. It may be used as a basis of the suffix tree to reach the space complexity of $O(n \log \sigma)$.

3.8.4 Resulting Suffix Tree

Making use of the suffix tree operations results in the following theorem. The space bound in the constant $t_{SA}$ case result from the observations that the compressed suffix tree requires $|SA| + 6n + o(n)$ bits of space. By Corollary 11 the compressed suffix array requires $O(n \log^\varepsilon n \log \sigma) = O(n \sqrt{\log n \log \sigma})$ bits by choosing $\varepsilon = \frac{1}{2}$. Correspondingly, the inverse suffix array requires $O(n \sqrt{\log n \log \sigma})$ bits by Theorem 13.

Theorem 16. [LSW08] A suffix tree with the required properties exists so that either of the following conditions is satisfied:

1. Assuming $\sigma = O(2^{\sqrt{\log n}})$, space requirement of $O(n \sqrt{\log n \log \sigma})$ bits with $t_{SA} = O(1)$ and $t_{\Psi} = O(1)$.

2. Space requirement of $O(n \log \sigma)$ bits with $t_{SA} = O(\log^\varepsilon n)$ and $t_{\Psi} = O(1)$.

4 Other Data Structures

4.1 Sparse Balanced Parentheses Support

A representation of strings with sparse balanced parentheses, i.e. parentheses “(” and “)’ and separator characters “ “, is required. The representation needs to support the following operations.

- **open(i)**: Finds the matching opening parenthesis for the closing parenthesis at position $i$ and reports its index in constant time.
- **close(i)**: Finds the matching closing parenthesis for the opening parenthesis at position $i$ and reports its index in constant time.

Lemma 17. The support for sparse balanced parentheses may be implemented in $O(n)$ bits.

Proof. A bit vector $B$ of length $n$ and its $\text{rank}_1$ and $\text{select}_1$ support are stored such that the value one indicates opening and closing parentheses and zero separator characters. The parentheses are stored in another bit vector $V$ with balanced parentheses support. This requires $O(n)$ bits of space. Indices of $B$ may be converted to indices of $V$ with the rank support and back with the select support in constant time. Together these require $O(n)$ bits of space.

23
4.2 Perfect Hashing

Given a finite set of non-negative integers $U$, a function $h$ is a perfect hash function for a subset $S \subseteq U$ if $h$ is an injection on $S$. A perfect hash function is also a minimal, if $h$ is a bijection. The following result on generating such functions has been attained.

**Lemma 18.** [BBD09] There is an algorithm that can generate perfect hash functions and minimal perfect hash functions the space requirement of which is $O(|S|)$ and time requirement is $O(1)$.

4.3 X and Y-Fast Tries

A data structure for storing a set of integers is required such that the following operations are supported. $M$ is the maximum of the stored integers.

- **predecessor$(x)$**: Reports the stored value that is lesser than or equal to $x$ in $O(\log \log M)$ time.

- **successor$(x)$**: Reports the stored value that is greater than or equal to $x$ in $O(\log \log M)$ time.

**Lemma 19.** [Wil83] The X-fast trie supports the operations with $O(n \log M)$ space requirement, $n$ being the count of the integers and $M$ their maximum.

**Proof.** To represent the integers, a binary trie of height $1 + \lceil \log M \rceil$ is created. The levels of the trie are numbered from 0 to $\lceil \log M \rceil - 1$. Internal nodes are stored only if they have leaves in their subtree and are represented as key-value pairs. The key is a prefix of length equal to the level index of each integer. In other words, considering value $x$, key on level 0 would be an empty bit string, on level 1 it would be $x[0..0]$, on level 2 it would be $x[0..1]$ and so on. The value is a pair of edges that point to nodes derived by appending 0 and 1 to the key of the current node. If no such node exists, in case of the left branch the edge points to the leftmost leaf of the right subtree, and in case of the right branch to the rightmost leaf of the left subtree. Such edges are called *descendant pointers*.

To represent the levels of the trie in $O(n \log M)$ space with $O(1)$ access to the nodes, each level is stored into a hash map that applies minimal perfect hashing. Since the functions require linear space by Lemma 18, the total space requirement is $O(n \log M)$.

The internal nodes may be encoded with three integers using $3w$ bits where $w = \lceil \log M \rceil$. One of the integers is the key of the internal node and the two remaining ones represent the edges. An edge being a descendant pointer is indicated by its $i$-th bit on level $i - 1$ as the edge points to a different subtree than suggested by its position in the triplet.
The leaf nodes store pointers to the next and the previous leaf each, which also requires $3w$ bits.

The predecessor and successor for value $x$ may be found by searching for the lowest common ancestor $v$ of $x$ and a value stored into the trie (considering the longest prefix in the bit string representation of the values). To this end, binary search is applied over the levels of the trie, which takes $O(\log \log M)$ time. If the found value is not $x$, either of the out-edges of the internal node must be a descendant pointer, since $v$ was found from an internal node. If the descendant pointer is the left out-edge, it points to the successor of $x$ and otherwise to the predecessor. If the value is not the queried one, the predecessor or successor may be located by using the leaf node pointers. Since at most two edges need to be followed, the additional steps take only constant time.

**Theorem 20.** [Wil83] The Y-fast trie supports the operations with $O(n)$ space requirement, $n$ being the count of the integers and $M$ their maximum.

**Proof.** The values are stored in $\frac{n}{\log M}$ balanced binary search trees each containing at least $\log M$ and at most $2\log M$ consecutive values. From each binary tree the smallest value, which is called representative, is also stored into an X-fast trie. Since each representative value occurs on $O(\log M)$ levels of the X-fast trie by Lemma 19 and there are $\frac{n}{\log M}$ representatives, these require $O(n)$ space. As the balanced binary trees also require $O(n)$ space together, the total space requirement is $O(n)$.

Predecessor and successor of a given value $x$ are found by looking for preceding and succeeding representatives $u$ and $v$ from the X-fast trie. By Lemma 19 this takes $O(\log \log M)$ time. The predecessor or successor may then be located from the binary search tree pointed to by $u$ in $O(\log \log M)$ time as each binary tree contains $O(\log M)$ values. For successor queries, in case a value was not found, $v$ must be the successor as it is the minimum of the next subtree.

5 Pattern Matching with One Difference or Mismatch

5.1 Heavy Path Decomposition

In the general case, the suffix tree is not balanced. Despite this, it may be partitioned into $O(\log n)$ levels [BGW00, CGL04, LSW08]. Given a suffix tree, an integral level is assigned to its every node. The root node is assigned level one. Every internal node is then processed as follows. If the node has level $l$, one of the child nodes that has the largest subtree in terms of the number of leaf nodes is also assigned level $l$ with ties broken arbitrarily. The other child nodes are assigned level $l + 1$. Edges that join two nodes that
have the same level are called *core edges* while others are called *side edges*. A node with an in-edge that is also a core edge is called a *core node*. Other nodes including the root are called *side nodes*. (See Figure 2 for an example.)

Based on the decomposition, the following observations can be made.

**Lemma 21.** [LSW08] There are $O(\log n)$ levels in a suffix tree with $n$ nodes.

**Proof.** Suppose an internal node $\tilde{u}$ at level $l$ has $k$ leaves in its subtree. Its child node $\tilde{v}$ that is a side node, and therefore has level $l + 1$, has at most $\frac{k}{2}$ leaves. As the suffix tree has $n$ leaves, there must be $O(\log n)$ levels. \hfill $\Box$

**Corollary 22.** [LSW08] There are $O(\log n)$ side edges on the path from root to any node in the suffix tree.

**Proof.** Since there are $O(\log n)$ levels by Lemma 21, there must be $O(\log n)$ side edges on the path. \hfill $\Box$

**Lemma 23.** [LSW08] Suppose that side edges $e_1$ and $e_2$ are in-edges of nodes $\tilde{v}_1$ and $\tilde{v}_2$ respectively. If $\tilde{v}_1$ and $\tilde{v}_2$ have the same level, then the subtrees rooted at $\tilde{v}_1$ and $\tilde{v}_2$ are disjoint.

**Proof.** By contradiction suppose that the subtrees are not disjoint. Without loss of generality, suppose that $\tilde{v}_2$ is located in the subtree the root of which is $\tilde{v}_1$. Since $\tilde{v}_1$ and $\tilde{v}_2$ have the same level, the path from $\tilde{v}_1$ to $\tilde{v}_2$ must be a core path, but this contradicts the assumption that $e_2$ is a side edge. \hfill $\Box$

### 5.1.1 Accessing Core Path Endpoints

By using a suitable data structure, the endpoints of a core path may be accessed quickly from each other.

**Lemma 24.** [LSW08] Given a side node $\tilde{x}$ that begins a core path, the leaf node $\tilde{y}$ that terminates the core path and vice versa may be found in $O(1)$ time using an $O(n)$ bit data structure, $n$ being the number of nodes in the suffix tree.

**Proof.** The sparse balanced parentheses representation described in Lemma 17 is used to store the core paths. Every side node that begins a core path is marked with an opening parenthesis and every leaf node that ends the path is marked with a closing parenthesis. Other nodes are marked with the separator character. Since there are $n$ nodes in the suffix tree, $O(n)$ bits of space is required and the queries take $O(1)$ time.

The parentheses are balanced as a result of node numbering. Counting the subtrees of node $\tilde{x}$ in lexicographic order, suppose $\tilde{v}$ is the child node of $\tilde{x}$ that is on the core path to $\tilde{y}$, $\tilde{u}$ is the root of the previous subtree and $\tilde{w}$ is the root of the next subtree. The nodes in $\tilde{u}$’s subtree are numbered from $\tilde{u}$ to $\tilde{v} - 1$ and the nodes in $\tilde{v}$’s subtree from $\tilde{v}$ to $\tilde{w} - 1$, which yields
\[ \bar{x} < \bar{u} \leq (\bar{v} - 1) < \bar{v} \leq (\bar{w} - 1) < \bar{w} \]. Therefore the core paths in \( \bar{u} \)'s subtree have indices enclosed between \( \bar{x} \) and \( \bar{y} \) and the core paths in \( \bar{w} \)'s subtree are outside the range from \( \bar{x} \) to \( \bar{y} \).

5.2 Suffix Array Samples

For some nodes of the suffix tree, a sample of the suffix array is stored in a data structure that provides fast queries. With every side node \( \bar{v} \), the parent node of which is \( \bar{u} \), a set \( \Gamma_{\bar{v}} \) is associated as follows. Suppose \( v_{le} = lb(\bar{v}) \) and \( v_{ri} = rb(\bar{v}) \).

\[
\Gamma_{\bar{v}} = \{ SA^{-1}[SA[i] + \text{plen}(\bar{u}) + 1] \mid i \equiv 1 \pmod{\log^2 n} \text{ and } v_{le} \leq i \leq v_{ri} \}
\]

\[
= \{ \Psi^{\text{plen}(\bar{u})+1}(i) \mid i \equiv 1 \pmod{\log^2 n} \text{ and } v_{le} \leq i \leq v_{ri} \}
\]

For every core leaf node \( \bar{v} \) for which \( lb(\bar{v}) = rb(\bar{v}) = v \), suppose \( \bar{u} \) is the start of the corresponding core path. The suffix array is sampled from its left and right sides as follows. The indices are stored in such a way that they are ordered by increasing longest common prefix length \( |lcp(v, i)| \), where \( i \) is a suffix array index, and may be queried by them.

\[
H^L_{\bar{v}} = \{ i \mid i \equiv 1 \pmod{\log^2 n} \text{ and } i \leq v \text{ and } |lcp(v, i)| \geq \text{plen}(u) \}
\]

\[
H^R_{\bar{v}} = \{ i \mid i \equiv 1 \pmod{\log^2 n} \text{ and } i > v \text{ and } |lcp(v, i)| \geq \text{plen}(u) \}
\]

The purpose of the sets \( H^L_{\bar{v}} \) and \( H^R_{\bar{v}} \) is to speed up the search on core paths for which binary search may not be applied directly. This is the case if the number of leaf nodes hanging from the path is high enough. The sets store suffix array indices that may be used to find a suitable branching point on the core path in question as follows. If the path label of the core leaf node at the end of the path does not match the pattern, a suitable suffix array index is located from the sets and the LCP operation is applied to find the branching point. The suffix array index of the core leaf node need not be considered (contrary to what Lam, Sung and Wong propose) as it is already known not to match the pattern. (See Lemma 31 Case 2.)

Lemma 25. [LSW08] Suppose \( n \) is the text length. \( \Gamma_{\bar{v}} \) for all side nodes \( \bar{v} \) may be stored using \( O(n) \) bits. Additionally, for any range from \( x \) to \( y \) a value \( i \) may be found in \( O(\log \log n) \) time such that \( \{ i \in \Gamma_{\bar{v}} \mid x \leq i \leq y \} \).

Proof. Every \( \Gamma_{\bar{v}} \) has a suffix array sample with a \( \log^2 n \) interval, which is possibly from the whole suffix array, but by Lemma 23 all the subtrees that have the same level are disjoint. As a result of this, for nodes on a given level there must be at most \( \frac{n}{\log^2 n} \) values in all \( \Gamma_{\bar{v}} \) combined. Since by Lemma 21 there are \( O(\log n) \) levels, the total size of \( \Gamma_{\bar{v}} \) for all side nodes \( \bar{v} \) is \( O(\frac{n}{\log n}) \).

\( \Gamma_{\bar{v}} \) are stored in Y-fast tries as described in Theorem 20 which support the queries in \( O(\log \log n) \) time. The size of the data structures combined is \( O(|\Gamma_{\bar{v}}| \log n) = O(n) \) bits. \( \square \)
Lemma 26. [LSW08] $H^l_\bar{v}$ and $H^r_\bar{v}$ may be stored for all core leaf nodes $\bar{v}$ for which $\text{lb}(\bar{v}) = \text{rb}(\bar{v}) = v$ using $O(n)$ bits. Additionally, for any range from $x$ to $y$ values $i_l \in H^l_\bar{v}$ and $i_r \in H^r_\bar{v}$ may be reported such that $x \leq |\text{lcp}(i_l, v)| \leq y$ and $x \leq |\text{lcp}(i_r, v)| \leq y$ in $O(\log \log n)$ time.

Proof. Since the suffix array is sampled with a $\log^2 n$ interval for every core leaf node, at most $\frac{n}{\log^2 n}$ leaf nodes are considered for inclusion in $H^l_\bar{v}$ and $H^r_\bar{v}$ associated with a given node $\bar{v}$. Since there are $O(\log n)$ levels, every leaf node is reachable from $O(\log n)$ side nodes. By Corollary 22 there are $O(\log n)$ side edges on a path from root to a given leaf node. Each of the nodes at the end of the side edges, that are also internal nodes, also begins a core path. Therefore each leaf node must be included in $O(\log n)$ different $H$ lists. Hence, the total size of $H^l_\bar{v}$ and $H^r_\bar{v}$ for all core leaf nodes $\bar{v}$ is $O\left(\frac{n}{\log n}\right)$ values.

$H^l_\bar{v}$ and $H^r_\bar{v}$ are stored in Y-fast tries which support the queries in $O(\log \log n)$ time as described in Theorem 20. The longest common prefix lengths and suffix array indices take together $O(\log n)$ bits each. Therefore the total size of all the $H$ data structures is $O(n)$ bits. \qed

5.3 Pattern Ranges

The suffix array ranges of each suffix of the pattern are indexed into arrays $F_{st}$ and $F_{ed}$. Both of these are of equal length compared to the pattern. Suppose $m$ is the pattern length and $n$ is the suffix array length.

\[
[F_{st}[i], F_{ed}[i]] = \text{range}(T, P[i..m]) \quad \text{for } 1 \leq i \leq m \\
F_{st}[j] = 0 \quad \text{for } j > m \\
F_{ed}[j] = n \quad \text{for } j > m
\]

Lemma 27. The arrays $F_{st}$ and $F_{ed}$ may be constructed in $O(mt_\Psi + m\sigma_{SA})$ time.

Proof. Suppose the pattern occurs exactly in the suffix tree. The values $\text{range}(T, P[i..m])$ may be retrieved by traversing the tree in $O(m\sigma_{SA})$ time and using the $\text{lb}$ and $\text{rb}$ operations, which take constant time. Once the corresponding suffix tree node has been found, the remaining $F_{st}$ and $F_{ed}$ values may be obtained by repeatedly following suffix links and using the aforementioned operations to fetch the suffix array range. This takes $O(mt_\Psi)$ time.

Suppose the complete pattern only matches the text up through pattern index $j$. At this point, $O(j\sigma_{SA})$ has been used. From the retrieved suffix tree node, the suffix link may be followed, and the matching may be continued from the node to which the link points. Matching the rest of the pattern to obtain $\text{range}(T, P[j..m])$ takes then $O((m - j - 1)\sigma_{SA})$. If a mismatch occurs at another position, the process may be reapplied. Traversing the
tree still requires $O(m \sigma t_{SA})$ time and suffix links need to be followed at most $m$ times.

5.4 Pattern Matching

The data structures required by the pattern matching algorithm do not increase the total space complexity from that of the compressed suffix tree.

Lemma 28. [LSW08] The one-approximate data structure may be stored in two ways.

1. In $O(n \log \sigma)$ bits for $t_{SA} = O(\log^\varepsilon n)$ and $t_{\Psi} = O(1)$.
2. In $O(n \sqrt{\log n \log \sigma})$ bits for $t_{SA} = O(1)$ and $t_{\Psi} = O(1)$.

Preparing the longest common prefix array of the suffix tree for range minimum queries takes $O(n)$ bits of space.

In addition, the $F_{st}$ and $F_{ed}$ arrays needed for each pattern require $O(m \log \sigma)$ bits of space, $m$ being the pattern length.

Proof. The lemma follows directly from Theorem 16 and Lemmas 25 and 26.

5.4.1 Using the Data Structure to Match Strings with One Difference

The data structure may be used to match a string concatenated to a path label of the suffix tree as follows.

Lemma 29. [LSW08] Suppose $T$ is a text of length $n$, $P$ is a pattern and $[v_{le}..v_{ri}] = range(T, P)$ a suffix array range. Then, for any position $i$ in $T$, $P$ is a prefix of $T[i..n]$ if and only if $v_{le} \leq SA^{-1}[i] \leq v_{ri}$.

Proof. The equivalence follows from these two implications.

$P$ is a prefix of $T[i..n]$ implies $v_{le} \leq SA^{-1}[i] \leq v_{ri}$. If $P$ is a prefix of $T[i..n]$, there must be a leaf node $\tilde{v}$ in the suffix tree of $T$ such that $P$ is the prefix of its path label. Considering the nodes from the path from root to $\tilde{v}$, suppose $\tilde{v}$ is the lowest node that has $P$ as a prefix. Then it follows that $v_{le} = lb(\tilde{u})$ and $v_{ri} = rb(\tilde{u})$.

Suppose not, i.e. there is a suffix tree node $\tilde{x}$ the suffix array index of which is $x$, for which $v_{le} \leq x \leq v_{ri}$ but $rb(\tilde{u}) < x$. As a result, there is a suffix tree node $\tilde{y}$ that is the lowest common ancestor of $\tilde{x}$ and $\tilde{u}$. However, since $\tilde{u}$ is the lowest node for the path label of which $P$ is a prefix and suffix tree edge labels may not begin with the same character, the path label of $\tilde{y}$ must not have $P$ as a prefix, which is a contradiction. Hence it follows that $v_{le} = lb(\tilde{u})$ and $v_{ri} = rb(\tilde{u})$. The case in which there is a suffix array index $z$ for which $v_{le} \leq z \leq v_{ri}$ but $z < lb(\tilde{u})$ is symmetrical.
\(v_{le} \leq SA^{-1}[i] \leq v_{ri}\) implies \(P\) is a prefix of \(T[i..n]\). Since by the given assumption \([v_{le}..v_{ri}]\) is the suffix array range that contains the suffixes the prefix of which is \(P\), then by the definition of the suffix array it must hold that \(v_{le} \leq SA^{-1}[i] \leq v_{ri}\).

\[\text{Lemma 30.} \] [LSW08] Given pattern \(P\), text \(T\) of length \(n\) and the corresponding suffix tree, suppose \(\bar{v}\) is a node such that \([v_{le}..v_{ri}] = range(T, P)\), that is, \(P\) is a prefix of \(\bar{v}\)’s path label. Then the following holds.

\[SA^{-1}[SA[v_{le}] + |P|] < SA^{-1}[SA[v_{le} + 1] + |P|] < ... < SA^{-1}[SA[v_{ri}] + |P|]\]

In other words, since all suffixes in the range have \(P\) as their prefix, their mutual lexicographic order is determined beginning from position \(|P| + 1\).

\[\text{Proof.}\] As \(\Psi^k(i) = SA^{-1}[SA[i] + k]\), the observation may be written as follows.

\[\Psi^{|P|}(v_{le}) < \Psi^{|P|}(v_{le} + 1) < ... < \Psi^{|P|}(v_{ri})\]

The \(|P|\) symbols that appear before \(SA[\Psi^{|P|}(v_{le})], SA[\Psi^{|P|}(v_{le} + 1)], ..., SA[\Psi^{|P|}(v_{ri})]\) are always equal to \(P\) which may be seen from the following.

Suppose \(i\) is a suffix array index.

\[
\begin{align*}
SA[\Psi^{|P|}(i)] &= SA[SA^{-1}[SA[i] + |P|]] \\
&= SA[i] + |P|
\end{align*}
\]

For \(v_{le} \leq i \leq v_{ri}\) the \(|P|\) characters that appear before \(SA[i] + |P|\) are always equal to \(P\). By Lemma 4 the values are monotone.

\[\text{Lemma 31.} \] [LSW08] Suppose \(\bar{u}\) is a suffix tree node and \(P_1\) its path label, \(c\) is a character drawn from \(\Sigma\) and \(P_2\) is another string the suffix array range is \([st..ed] = range(T, P_2)\). All occurrences of \(P = P_1cP_2\) may be computed in \(O(t_{SA}(\log \log n + occ))\) time where \(occ\) is the total number of occurrences of \(P\) in \(T\).

\[\text{Proof.}\] Suppose \(\bar{v} = \text{child}(\bar{u}, c)\) and \([v_{le}..v_{ri}] = range(T, P_1c) = [lb(\bar{v}), rb(\bar{v})]\).

If \(P\) occurs in \(T\), \(P\) must occur at position \(SA[i]\) for some \(v_{le} \leq i \leq v_{ri}\), that is, in the subtree of \(\bar{v}\). By Lemma 29 the occurrence of \(P\) at position \(SA[i]\) may be verified by checking the following inequality.

\[
\begin{align*}
st &\leq \Psi^{|P_1|+1}(i) \quad \leq ed \\
&\iff st \leq SA^{-1}[SA[i] + |P_1| + 1] \leq ed
\end{align*}
\]

As a result of this, the occurrence of \(P\) may be found by checking all \(v_{le} \leq i \leq v_{ri}\). Binary search may be applied as by Lemma 30 the values are sorted. Since \(SA^{-1}[SA[i] + |P_1| + 1]\) may be retrieved in \(O(t_{SA})\) time by using the suffix array and the inverse suffix array, one occurrence of \(P\) may be found in \(O(t_{SA} \log (v_{ri} - v_{le}))\) time. There are two cases to be considered.
Case 1: $\bar{v}$ is a side node. In this case $\bar{v}$ has $\Gamma_{\bar{v}}$ associated with it. Since $\bar{u}$ is the parent node of $\bar{v}$, it holds that $plen(\bar{u}) = |P_1|$ and $\Gamma_{\bar{v}} = \{SA^{-1}[SA[\bar{v}] + |P_1| + 1] | i \equiv 1 \pmod{\log^2 n} \text{ and } v_{l_e} \leq i \leq v_{r_i}\}$. The steps taken depend on the contents of $\Gamma_{\bar{v}}$.

Case 1.1. $\Gamma_{\bar{v}}$ is empty. This means that the number of leaves in the subtree of $\bar{v}$ is less than $\log^2 n$. Therefore binary search may be applied directly to find one occurrence of $P$ in $O(t_{SA} \log(v_{r_i} - v_{l_e})) = O(t_{SA} \log \log n)$ time.

Case 1.2. $\Gamma_{\bar{v}}$ is non-empty and using predecessor and successor queries a suffix array index $i$ may be found such that $st \leq SA^{-1}[SA[i] + P_1 + 1] \leq ed$. By Lemma 25 this takes $O(\log \log n)$ time.

Case 1.3. $\Gamma_{\bar{v}}$ is non-empty but the index $i$ in the previous case could not be found. Instead, predecessor and successor queries are applied to find values $a, b \in \Gamma_{\bar{v}}$ such that $a \leq st$ and $ed \leq b$. If either is not found, the list boundary is used instead. It holds that $b - a \leq \log^2 n$. As a result of this, one occurrence of $P$ may be found in $O(t_{SA} \log(b - a)) = O(t_{SA} \log \log n)$ time.

Case 2: $\bar{v}$ is a core node. Suppose $Q$ is the core path that contains $\bar{v}$. The side node $\bar{y}$ that begins $Q$ may be retrieved by checking core path beginnings as part of tree traversal. By Lemma 24 this may be done in constant time for each node. Moreover, the terminating leaf node $\bar{x}$ may be retrieved in constant time by using the same data structure.

Next the count of leaves hanging from $Q$ is considered. This is done by using the sets $H_{\bar{x}}^{l}$ and $H_{\bar{x}}^{r}$ that are associated with the core leaf node $\bar{x}$ at the end of $Q$. Again the steps taken depend on the contents of the sets. Suppose $x$ is the suffix array index of $\bar{x}$.

Case 2.1. The sets are empty. This means that the number of leaves in the subtree of $\bar{y}$ and therefore also in the subtree of $\bar{v}$ is less than $\log^2 n$. Binary search may be applied to find one occurrence of $P$ in $O(t_{SA} \log(v_{r_i} - v_{l_e})) = O(t_{SA} \log \log n)$ time.

Case 2.2. Next step is to determine the longest common prefix length $q = |lcp(SA^{-1}[SA[x] + |P_1| + 1], st)|$. By Lemmas 15 and 28 this takes $O(t_{SA})$ time. Since $st$ is a suffix array index for a suffix that has $P_2$ as a prefix, if $|P_2| \leq q$, there must be an occurrence of $P$ at $x$. 

31
Case 2.3. The LCP length $q$ was determined but it holds that $q < |P_2|$. In this case $H^l_x$ and $H^r_x$ are searched to find either of two suffix array index ranges such that $j_l \in H^l_x$ and $j_r \in H^r_x$ and either of the following statements holds.

\begin{align*}
|lcp(j_i - \log^2 n, x)| &\leq |P_1| + q + 1 \leq |lcp(j_i, x)| \quad \text{or} \\
|lcp(j_r, x)| &\leq |P_1| + q + 1 \leq |lcp(j_r + \log^2 n, x)|
\end{align*}

By Lemma 26 this may be computed in $O(\log \log n)$ time. Next, a suffix array index $i$ is located from one of the following ranges.

\begin{align*}
&j_l - \log^2 n \leq i \leq j_l \quad \text{or} \\
&j_r \leq i \leq j_r + \log^2 n \quad \text{or} \\
&x - \log^2 n \leq i \leq x + \log^2 n
\end{align*}

Binary search is applied so that $|lcp(i, x)| = |P_1| + q + 1$. In all cases the binary search takes $O(t_{SA} \log \log n)$ time. Suppose $\bar{i}$ is the leaf node in the suffix tree that corresponds to $i$. A node $\bar{r} \in Q$ is then located by determining the lowest common ancestor of nodes $\bar{i}$ and $\bar{x}$. Since accessing the inverse suffix array takes $O(t_{SA})$ time and the suffix tree can determine the lowest common ancestor of two nodes in constant time, finding $\bar{r}$ takes $O(t_{SA})$ time. $P$ is then matched against the path label of $\bar{r}$. If $P$ is still not completely matched, the search may be repeated by using $\bar{r}$ as the starting position and enumerating its side edges. This takes additional $O(t_{SA} \log \log n)$ time.

Finding the remaining occurrences. Once a suffix array index $i$ has been found such that $P$ is a prefix of the suffix in question, the remaining suffix array indices may be found by checking entries $i'$ starting from $i$ such that $st \leq SA^{-1}[SA[i'] + |P_1| + 1] \leq ed$ until either $v_{le}$ or $v_{ri}$ is reached or a mismatch is found. This requires $O(t_{SA}(occ + 2))$ time. □

5.4.2 Approximate Matching Algorithm

Given a suffix tree node, a single character and the suffix array range of the pattern suffix, a procedure that follows Lemma 31 may be implemented so that it takes $O(t_{SA}(\log \log n + occ))$ time to run. The procedure is defined as $TreeSearch(node, c, st, ed)$. The pattern suffix indices may be provided by the $F_{st}$ and $F_{ed}$ arrays in constant time, which in turn may be constructed in $O(mt_q + \sigma t_{SA})$ time by Lemma 27.

$TreeSearch$ may be called so that all positions in the given pattern are enumerated and at each position a deletion, substitutions and insertions are introduced. This is shown in Algorithm 34. To solve the one-mismatch problem, the steps to test deletions and insertions should be omitted. This does not affect the time complexity of the algorithm. Hence the following
results are attained with the space complexities being an outcome of Lemma 28.

**Theorem 32.** [LSW08] Given a data structure of $O(n\sqrt{\log n} \log \sigma)$ bits, the one-mismatch or one-difference problem may be solved in $O(\sigma m \log \log n + \text{occ})$ time given a pattern of length $m$ and a text of length $n$ drawn from an alphabet of size $\sigma = O(2^{\sqrt{\log n}})$.

**Theorem 33.** [LSW08] Given a data structure of $O(n \log \sigma)$ bits, the one-mismatch or one-difference problem may be solved in $O(\log^\epsilon n(\sigma m \log \log n + \text{occ}))$ time given a pattern of length $m$ and a text of length $n$ drawn from an alphabet of size $\sigma$. 
Algorithm 34 ([LSW08] One-Approximate Algorithm. \( P \) is the pattern of length \( m \)).

1: Construct \( F_{st} \) and \( F_{ed} \).
2: \( u \leftarrow \text{root node} \)
3: \( i \leftarrow 1 \)
4: loop

   ▶ Test deletion at \( i \), i.e. find occurrences of \( P[1..i-1]P[i+1..m] \).
5:      if \( P[i] \neq P[i+1] \) then
6:         Report the occurrences found by \( \text{TreeSearch}(u, P[i+1], F_{st}[i+2], F_{ed}[i+2]) \).
7:      end if

   ▶ Test substitutions at \( i \), i.e. find occurrences of \( P[1..i-1]cP[i+1..m] \) for all \( c \in \Sigma\{P[i]\} \).
8:      for \( c \in \Sigma\{P[i]\} \) do
9:         Report the occurrences found by \( \text{TreeSearch}(u, c, F_{st}[i+1], F_{ed}[i+1]) \).
10:   end for

   ▶ Test insertions at \( i \), i.e. find occurrences of \( P[1..i-1]cP[i..m] \) for all \( c \in \Sigma\{P[i]\} \).
11:      for \( c \in \Sigma\{P[i]\} \) do
12:         Report the occurrences found by \( \text{TreeSearch}(u, c, F_{st}[i], F_{ed}[i]) \).
13:     end for

   ▶ Test exact match at \( i \).
14:     \( v \leftarrow \text{child}(u, P[i]) \)
15:     \( E \leftarrow \text{label}(u, v) \)
16:     if \( P[i..i+|E|-1] = E \) then
17:        \( u \leftarrow v \)
18:        \( i \leftarrow i + |E| \)
19:     else
20:        Find the smallest \( j > i \) such that \( P[j] \neq E[j-i+1] \).
21:        Report all the occurrences of \( P \) such that the error is at \( j \).
22:        Terminate.
23:     end if
24: end loop
6 Pattern Matching with Multiple Differences or Mismatches

6.1 Dynamic Programming Algorithm

In order to apply the one-difference algorithm in solving the \( k \)-difference problem, a dynamic programming approach is applied [Sel80]. Given two strings \( P \) and \( Q \) of lengths \( m \) and \( n \), a matrix \( E \) of size \((m + 1) \cdot (n + 1)\) is filled. The matrix entry \( E[i, j] \) stores the edit distance of \( P[1..i] \) and \( Q[1..j] \).

The matrix is filled as follows.

\[
\begin{align*}
E[0, 0] &= 0 \\
E[0, j] &= E[0, j - 1] + c(\varepsilon, Q[j]), \quad 1 \leq j \leq n \\
E[i, 0] &= E[1 - i, 0] + c(P[i], \varepsilon), \quad 1 \leq i \leq m \\
E[i, j] &= \min\left\{ E[i - 1, j] + c(P[i], \varepsilon), \\ E[i - 1, j - 1] + c(P[i], Q[j]), \\ E[i, j - 1] + c(\varepsilon, Q[j]) \right\}
\end{align*}
\]

Here, \( c(\cdot, \cdot) \) is the cost function and \( \varepsilon \) the empty string; \( c(\varepsilon, Q[j]) \) and \( c(P[i], \varepsilon) \) are the costs for insertion and deletion and their values are equal to 1. For substitution, \( c(P[i], Q[j]) = 1 \) if \( P[i] \neq Q[j] \) and for equivalent characters (i.e. \( P[i] = Q[j] \)) \( c(P[i], Q[j]) = 0 \).

Suppose \( m + k \leq n \). The matrix needs to be filled column-wise from left to right up to the \((m + k)\)-th column. Then a match, if there is one, may be read from the last row of the matrix where the value of an entry is at most \( k \).

Considering \( E[i, j] \), entries where \( i > j + k \) or \( i < j - k \) will have an edit cost greater than \( k \). Hence, there are only \( 2k + 1 \) rows to be filled in each column and the algorithm may be run in \( O(mk) \) time. By storing only the points where the edit cost increases on each diagonal of the matrix, the space requirement may be reduced further. This gives the following lemma.

**Lemma 35.** [Ukk85] Given two strings \( P \) and \( Q \) of length \( m \) and \( n \) respectively where \( m \leq n \), the prefixes of \( Q \) the edit distance from \( P \) are at most \( k \) may be found in \( O(mk) \) time and \( O(k \cdot \min\{k, m\}) \) space using unit cost for edit distance.

For the \( k \)-mismatch problem, only the diagonal of the matrix needs to be filled as insertions and deletions are not allowed.

6.1.1 The Number of Edit Operations Is Bounded

To evaluate the time complexity of the algorithm, the number of possible sequences of edit operations to convert one string to another needs to be considered.
Lemma 36. [Ukk93] Suppose $P$ and $Q$ are strings that have been drawn from an alphabet of size $\sigma$ and have edit distance less than or equal to $k$. Suppose $|P| = m$. There are $O(\sigma^k m^k)$ possible edit traces, that is, sequences of edit operations, that convert $P$ to $Q$ using unit cost for edit distance.

Proof. For operations that either convert one character $a$ to another character $x$ or delete $a$ (by choosing $\varepsilon$ for $x$), there are $\sigma + 1$ different values for $x$. The number of ways to apply $k$ such operations is thus at most $\binom{m}{k}(\sigma + 1)^k$.

For operations that insert a character into $P$, there are $m + 1$ positions between characters into which a new character may be placed. As there are $\sigma$ options to choose the new character, the number of ways to apply $k$ such operations is at most $(m + 1)^k \sigma^k$.

The different possible combinations of the aforementioned types of operations give the following inequality.

\[
\sum_{t=0}^{k} \binom{m}{t} (\sigma + 1)^t + (m + 1)^{k-t} \sigma^{k-t}
\]

\[
= \sum_{t=0}^{k} \binom{m}{t} (\sigma + t)^t + (m + 1)^t \sigma^t
\]

\[
\leq 2 \sum_{t=0}^{k} [(m + 1)^t (\sigma + 1)^t]
\]

\[
= O(m^k \sigma^k)
\]

\[
\square
\]

6.2 Combining With the One-Approximate Algorithm

The one-difference or one-mismatch algorithm may be combined with the dynamic programming algorithm to find the matching substrings in less time. This is done by considering the path labels of the suffix tree. Similar to the number of edit operations, the number of partially matching path labels is bounded.

Lemma 37. [LSW08] Suppose $P$ is a string of length $m$, $T$ is a text and both are drawn from the same alphabet of size $\sigma$. There are $O(\sigma^k m^k)$ different path labels $Q$ in the suffix tree of $T$ such that the edit distance between $P$ and $Q$ is at most $k$.

Proof. The number of possible path labels $Q$ is bounded by the number of edit traces that may be applied to $P$. Hence the given limit follows from Lemma 36. \square

The result affects the upper bound for time complexity of solving the $k$-difference or $k$-mismatch problem by comparing suffix tree path labels.
Lemma 38. [LSW08] Suppose $T$ is a text of length $n$, $P$ is a pattern of length $m$ and both are drawn from the same alphabet of size $\sigma$. After preprocessing $T$ to obtain either an $O(n\sqrt{\log n} \log \sigma)$ or $O(n \log \sigma)$ bits data structure, the $k$-difference or $k$-mismatch problem may be solved in $O(\sigma^k m^{k+1}(k + t_{SA}) + t_{SA} \cdot \text{occ})$ time where $\text{occ}$ is the number of occurrences of $P$ in $T$ given $k \cdot \min\{k, m\}$ is $O(n\sqrt{\log n})$ or $O(n)$ respectively.

Proof. The result is achieved by finding the shortest prefixes on all path labels that start from the root and have an edit distance of at most $k$ from $P$. To this end, the the suffix tree is traversed in preorder and the dynamic programming algorithm is applied to each path label.

Suppose $Q$ is the path label of the current suffix tree node. The characters of $Q$ are enumerated and for each character position $j$ a new column $E[\cdot, j]$ is computed in the dynamic programming matrix to calculate the edit distance of $P[1..m]$ and $Q[1..j]$. As accessing a character in a path label requires $O(t_{SA})$ time, the column may be computed in $O(k t_{SA})$ time by Lemma 35. The space requirement is $O(k \cdot \min\{k, m\})$. Enumerating the characters is terminated either when the edit distance in the new column $E[\cdot, j]$ is greater than $k$, when $E[m, j]$ is filled as the end of the pattern was reached or when a leaf node is reached. If $E[m, j] \leq k$, a match was found and the matching suffix array range may be reported by using the $lb$ and $rb$ operations on the suffix tree node the path label of which has $Q$ as prefix.

When a path label has been handled, the next one is processed by taking the next sibling of the node to which the path label corresponds. Only as many columns need to be computed in the dynamic programming matrix as how many characters were enumerated from the path label of the previous node. When a rightmost sibling has been processed, a subtree has been traversed in preorder and the next sibling of the root of the subtree may be processed.

Since there is a bound on the number of the path labels the edit distance of which from $P$ is $k$, the total search time is $O(\sigma^k m^k \cdot m(k + t_{SA}) + t_{SA} \cdot \text{occ}) = O(\sigma^k m^{k+1}(k + t_{SA}) + t_{SA} \cdot \text{occ})$ by Lemma 37.

Theorem 39. [LSW08] Suppose $T$ is a text of length $n$, $P$ is a pattern of length $m$ and both are drawn from the same alphabet of size $\sigma$. After preprocessing $T$ to obtain either an $O(n\sqrt{\log n} \log \sigma)$ (given $\sigma = O(2^{\sqrt{\log n}})$) or $O(n \log \sigma)$ bits data structure, the $k$-difference or $k$-mismatch problem may be solved in $O(\sigma^k m^k (k + \log \log n) + \text{occ})$ or $O(\log^\varepsilon n(\sigma^k m^k (k + \log \log n) + \text{occ}))$ time where $\text{occ}$ is the number of occurrences of $P$ in $T$ and $0 < \varepsilon \leq 1$ given $k \cdot \min\{k, m\}$ is $O(n\sqrt{\log n})$ or $O(n)$ respectively.

Proof. With the method used in Lemma 38 the problem is solved for the $(k-1)$-difference or $(k-1)$-mismatch problem. This requires $O(\sigma^{k-1} m^k (k + t_{SA}) + t_{SA} \cdot \text{occ}_{k-1})$ time where $\text{occ}_{k-1}$ is the number of approximate matches of $P$ in $T$ with edit distance less than $k$. This results in zero or more
paths that terminate with edit distance $k - 1$. These may be used to solve the problem by using the one-difference or one-mismatch algorithm. As there are $O(\sigma^{k-1} m^{k-1})$ paths with edit distance $k - 1$ by Lemma 37, the occurrences of $P$ with at most $k$ differences or mismatches may be found in $O(\sigma^{k-1} m^k (k + t_{SA} + t_\Psi + t_{SA} \sigma \log \log n) + t_{SA} \cdot occ)$ time. The stated time and space complexities follow from Theorem 16.

7 Experiments

Two sets of experiments were conducted. Firstly, the benchmarks provided with the SDSL library were executed in order to test the performance of Rao’s compressed suffix array. Secondly, a sequence alignment tool based on Lam, Sung and Wong’s algorithm was compared with two different aligners.

The tests were executed on a Linux 3.13 workstation running Ubuntu 14.04.4 LTS\(^2\). The workstation had one Intel Core i7-3770 processor with four cores and the capability of running two threads on each core, and 16 gigabytes of memory.

7.1 SDSL benchmarks

The benchmark provided with the SDSL library was used to evaluate the performance of the compressed suffix array. The suffix array was used as part of the compressed suffix tree provided in the library. The data used were 100 and 200 megabytes of DNA sequences and 200 megabytes of protein sequences (in text format) from The Human Genome Project [Del+98] and the Swissprot database as provided by the Pizza & Chili Corpus\(^3\). All tests were compiled with the clang 3.7 C++ compiler\(^4\) and the libc++ standard library\(^5\).

7.1.1 Suffix Array Benchmarks

The suffix array benchmarks included in SDSL are similar to those implemented by the Pizza & Chili Corpus. Three types of benchmarks were run.

Counting. The counting query checks how many times the pattern occurs in the text. The corresponding benchmark evaluates the speed and memory usage as a function of the pattern length.

\(^2\)http://www.ubuntu.com/
\(^3\)http://pizzachili.dcc.uchile.cl/
\(^4\)http://clang.llvm.org/
\(^5\)http://libcxx.llvm.org/
**Locating.** The locating query checks at which positions the pattern occurs in the text. Similar to the counting benchmark, the locating benchmark evaluates the speed and the memory usage. Additionally, different suffix array sampling densities are tested.

**Extracting.** The extracting query evaluates the performance of extracting continuous sequences of text from the index. Different suffix array sampling densities are also tested.

Rao’s compressed suffix array was compared to that of Sadakane [Sad03] as well as to suffix arrays based on either a Huffman-shaped wavelet tree or one that uses run-length encoding [MN05].

### 7.1.2 Suffix Tree Benchmarks

SDSL’s suffix tree benchmarks measure the speed of six operations of the suffix tree. Each test is repeated for every node of the suffix tree.

**LCA.** The lowest common ancestor operation of the suffix tree is tested. The nodes the ancestor of which is determined are chosen at random from the leaf nodes of the subtree of the current node.

**Letter.** The label operation of the suffix tree is tested.

**SLink.** For each tested node the first suffix link retrieved.

**Child.** For each tested node a random child node is retrieved.

**SDepth.** The string depth of each node is retrieved.

**Parent.** For each tested node a random child node is chosen and its parent is retrieved.

Rao’s compressed suffix array and the inverse suffix array by Lam, Sung and Wong were used as a backing of Sadakane’s suffix tree [Sad07]. The other tested suffix trees were that of Sadakane backed by Sadakane’s suffix array and one with a Huffman-shaped wavelet tree, as well as a suffix tree by Ohlebusch, Fischer and Gog [OFG10] backed by either a Huffman-shaped wavelet tree or Rao’s suffix array, and a suffix tree by Russo, Navarro and Oliveira [RNO11]. The chosen sample rate was 4.
7.1.3 Results

The space requirement of the tested suffix array is very high compared to the original text or the other suffix arrays. The index could take more than six times the space of the original, uncompressed text whereas the compared indices could fit in less than one-third of that. Neither the count query nor original text extraction is particularly fast. However, locating a pattern is very fast, even though the benefit is doubtful given the size of the index. The speed of the other indices also approaches that of Rao’s suffix array. As a result, choosing one of them for text indexing purposes could be worthwhile.

Despite this, Sadakane’s suffix tree backed by Rao’s suffix array and Lam, Sung and Wong’s inverse suffix array performed speedwise very well compared to the other compressed suffix arrays, and the similarly backed suffix tree by Ohlebusch, Fischer and Gog performed even better. Especially the Letter and Child tests yielded good results, and in the LCA and Parent tests the results were similar to those of the compared suffix trees. As with the suffix array, this may not be a sufficient justification for the space requirement.

The complete results have been listed in Appendix A.

7.2 Sequence alignment tool comparison

To test the practicality of the Lam, Sung and Wong’s algorithm (abbreviated LSW), a semi-local aligner that makes use of the algorithm was written. The tool takes its input in FASTA format. In indexing mode, the tool reads one DNA sequence and creates the necessary indexing data structures which may be read from disk when needed. In alignment mode, the data structures are read into memory. The FASTA input file, which may contain multiple patterns, is processed line by line. As this is done, the alignment tasks are created, which may run in parallel. To this end, the tasks are given to libdispatch\textsuperscript{6}\textsuperscript{7} library which creates a number of threads determined from the system load. To report unique matches, a post-processing step was added, in which the matching suffix array ranges are combined and converted to text indices if so requested by the user.

The implementation was compared with two other aligners: Erne 2.1.1 [DTP14] and readaligner November 2012 release [MVLK10]. Their workflow is similar, except that the user specifies the number of threads.

\textsuperscript{6}https://developer.apple.com/library/ios/documentation/Performance/Reference/GCD_libdispatch_Ref/
\textsuperscript{7}http://nickhutchinson.me/libdispatch/
7.2.1 Implementation of the Algorithm and Supporting Data Structures

Implementations provided by the SDSL library [GBMP14] were used for bit and integer vectors, rank and select support, range minimum queries, balanced parentheses support and the suffix tree.

For containers that required storing keys or keys and values so that the keys were not necessarily consecutive, the implementation of the CHD perfect hash algorithm [BBD09] by William Ahern was used.

X-fast trie was implemented with SDSL integer vectors and the CHD algorithm. Additionally, Y-fast trie uses a balanced binary search tree that was implemented with an array and the Eytzinger method for calculating descendant indices.

The parameters of the compressed suffix array were chosen so that $t_{SA}$ and $t_\Psi$ are constant.

7.2.2 Test setup

For the tests, the implementation of the algorithm by Lam, Sung and Wong was built with the clang 3.7 C++ compiler\(^8\) and the libc++ standard library\(^9\). Readaligner was built with GCC 4.8.4\(^{10}\) and GNU libstdc++\(^{11}\) as these were believed to provided better support for the OpenMP API for parallel programming\(^{12}\). Of Erne, the prebuilt version provided by the developers was used.

The data used for testing was the genome of *E. coli*, strain K-12, substrain MG1655, version U00096.3 [PBTM02], the size of which is 4.6 Mb. The complete sequence was indexed with each of the tools. To test alignment, ten subsequences of each length 16, 32, 64, 128 and 256 were randomly chosen from the original sequence. In addition, the sets of ten subsequences were copied so that there would be a copy of every set in which there would be 1, 2, 3, 4 or 6 randomly positioned mismatches or indels in each subsequence. The errors were generated for each set independently, i.e. not by using a set with a lesser amount of errors as a basis. As a result, 50 different subsequences divided into five sets of ten subsequences were obtained. Each tool was then tested with the same collection of subsequences or a subset thereof.

Each tool provided the option of running either in parallel or single-threaded mode as well as reporting either only one occurrence or all occurrences of the searched patterns. Every combination of these modes was tested. All subsequences with the same number of errors were provided to

---

\(^8\)http://clang.llvm.org/
\(^9\)http://libcxx.llvm.org/
\(^10\)https://gcc.gnu.org/
\(^11\)https://gcc.gnu.org/libstdc++/
\(^12\)http://openmp.org/
each tool in one FASTA file in order to allow parallel searching when enabled. Each test was run five times in succession to reduce the effect of disk access and potential spikes in system load caused by system processes. In all cases, the necessary indices had been prepared earlier, and the test run only included the time necessary to read the data structures into memory.

7.2.3 Results

The results may be seen in Figures 4 and 5. The index sizes are listed in Table 6. The two other aligners make use of search space pruning, which is believed to appear in the results of the tested aligner in three ways.

**Significant decrease in speed.** The compared aligners yielded results in time shorter by multiple orders of magnitude compared to the tested aligner. Even if the time spent on reading the index of the tested aligner from disk was subtracted from the results, the tool still required at least ten times the amount of time needed by readaligner. As a consequence, not all tests were completed as they would have taken too much time.

Examination of the results indicated that most of the time, so much as more than half of the total, was spent on balanced parentheses support via e.g. types of suffix tree access. This would indicate that replacing the BPS data structure by Sadakane and Navarro [SN10] could make the algorithm faster in practice. On the other hand, the dynamic programming algorithm and the Y-fast trie did not appear in the profiling results with the chosen inputs. This would indicate that they did not contain significant bottlenecks with respect to running time.

**Significant increase in index size.** The indices of the compared aligners required much less space than the one needed by the tested aligner. Examination of the index structure indicated that while the compressed suffix array requires a notable amount of space, as indicated by SDSL’s benchmarks, the portion of the total space required by the suffix array samples stored in the $\Gamma$ and $H$ sets in form of Y-fast tries is greater.

**Significant increase in accuracy.** The tested aligner was able to locate all of the subsequences when searching for only the first match while the compared aligners generally were not. For applications in which false negative results are not acceptable, this is a significant benefit. However, when reporting all matches, readaligner was able to locate the highest amount of matches in some instances. The reason for this is unclear.
Table 6: Index sizes.

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncompressed genome</td>
<td>4.5 MB</td>
</tr>
<tr>
<td>Erne index</td>
<td>5.3 MB</td>
</tr>
<tr>
<td>Readaligner indices</td>
<td>9.2 MB</td>
</tr>
<tr>
<td>LSW index</td>
<td>245 MB</td>
</tr>
</tbody>
</table>

7.3 Possibilities for Improvement

Examination of the data structures and running the aligner with a profiler revealed some options for improving the space-efficiency of the algorithm in practice. Especially modifications to the Y-fast tries were considered needful. However, these were not implemented.

7.3.1 Replacing the Y-fast Trie with a Balanced Binary Search Tree

An optimization had already been made to store a nullptr in place of the trie in case of an empty set. A further improvement would be storing a balanced binary search tree in place of the trie in case $n \leq k \log M$ where $n$ is the number of items, $M$ is the domain limit and $k$ some constant, as this will yield the same $O(\log \log M)$ time complexity for predecessor and successor queries.

7.3.2 Reducing the Number of Y-Fast Tries

Storing a Y-fast trie for each suffix tree node requires the size of the pointer with the current implementation and for the remaining nodes, an additional overhead may exist. An alternative would be to store only one Y-fast trie instead of one for each node as described in the algorithm. This would result in two tries, one for the $\Gamma$ and $H$ sets each. The tries would contain all the values associated with the nodes. To identify the values stored with a given node, a transformation would be applied to the values.

One way to make the transformation would be to add an identifier to each value. The identifiers would be constructed by using a similar means as with the Elias inventories; a constant that is greater than any of the

Legend for Figures 4 and 5.
Figure 4: Benchmarks for reporting one match.
Figure 5: Benchmarks for reporting all matches.
stored values would be chosen and then multiplied by the suffix tree node number. Such a transformation is reversible and may be done in constant time. Moreover, by choosing a constant that is also a power of two, modular algebra could be avoided. As a result, all values associated with a given node would be stored consecutively and the results of predecessor and successor queries would not be affected.

8 Conclusion

A verbatim implementation of the algorithm by Lam, Sung and Wong was written. Of the supporting data structures required by the algorithm, implementations of Y-fast tries and the compressed suffix array described by Rao were written as well and combined with existing implementations of other required data structures. To test the algorithm, a sequence alignment tool was written as well and compared with two existing tools that make use of heuristics.

The compressed suffix array implementation was tested separately. The benchmarks indicated that, while the compression is not nearly as good as with Sadakane’s compressed suffix array, the performance was very good.

The time and space requirements of the implemented aligner were higher by several orders of magnitude with respect to the two other aligners. However, the tested aligner was able to locate all patterns in the case where only the first match was searched. As the number of correct matches increases as the allowed number of errors increases, the correct number of total matches could not be determined. Assuming that the number reported by the implementation was correct, the performance comparison is not fair: the other two aligners yielded different results and producing the same set of matches could affect performance.

The reason for the discrepancy is not clear. A problem in the test setup cannot be completely ruled out. Another explanation is that there are bugs in the implementation of the compared aligners as they did not report a match for some of the patterns where one did exist. While a correct implementation of the aligners could perform worse, it is unlikely that fixing any bugs would cause so much slowdown that the big picture of the experiments would change.

It is worth noting that the algorithm on which Erne 2 is based did not achieve complete correctness as defined by the authors in their tests [VDTP12]. However, for the tests in question, finding unique matches near the expected position was considered correct while here the number of matches was examined.

A number of small optimizations were proposed to make the implementation of the algorithm more practical. However, these are unlikely to significantly improve its performance.
8.1 Chan et al.’s Algorithm for Pattern Matching with $k$ Errors

The algorithm by Chan et al. [Cha+06] achieves a time complexity of $O(m + 3^k k^3 occ + (c \log n)^{k(k+1)} \log \log n)$ with an index of size $O(n)$ words where $c$ is a positive constant. It uses the algorithm by Lam, Sung and Wong for patterns shorter than $k \cdot 5^k \log^{k+1} n$ characters and a checkpoint-based approach for longer patterns. The checkpoint interval results from the space complexity of the data structures involved. Even a $k$-error match for a long pattern is bound to contain checkpoints, so indexing these will make matching relatively efficient.

The general idea of the algorithm is to take a sample of the text at checkpoint positions. At these positions, the text is divided into a prefix and a suffix. All the suffixes of the text that begin from a checkpoint are stored into the set TAIL and all the prefixes are stored into another set HEAD. Suppose $P$ is again a pattern of length $m$, and $\beta$ is the sample rate. All prefixes of $P$ the length $i$ of which is less than $\beta + k$, and the corresponding suffixes are then considered. A match is found if HEAD contains a match of $P[1..i-1]$ with some number of errors $k_1$ ending at text position $j$, TAIL contains a match of $P[i..m]$ with some number of errors $k_2$ starting at text position $j + 1$, and $k_1 + k_2 = k$. Such pairs of pattern prefixes and suffixes are called connecting pairs.

To make the algorithm efficient, a suitable indexing data structure is needed. To this end, all text prefixes and suffixes in HEAD and TAIL respectively are stored into a type of $k$-errata tries [CGL04] (called edit-trees by Chan et al.) such that for each value from zero to $k$ one pair of tries is stored. Each pair of tries is also stored into a tree cross-product data structure [BGW00] which is then used to find all suitable connecting pairs given all nodes in the tries that represent the prefixes and suffixes. In order to implement the algorithm, implementations of these additional data structures are needed.

References


Information and control, 64(1-3):100–118, 1985 (cited on page 35).

In, Combinatorial pattern matching, pages 228–242, 1993 (cited 
on pages 3, 36).

[VDTP12] Vezzi, Francesco, Del Fabbro, Cristian, Tomescu, Alexandru I 
and Policriti, Alberto. rNA: a fast and accurate short reads 
on page 46).

[Wei73] Weiner, Peter. Linear pattern matching algorithms. In, Switch-
ing and automata theory, 1973. SWAT’08. IEEE conference 
record of 14th annual symposium on, pages 1–11, 1973 (cited 
on page 9).

[Wil83] Willard, Dan E. Log-logarithmic worst-case range queries are 
possible in space $\Theta(n)$. Information processing letters, 17(2):81– 
84, 1983 (cited on pages 24, 25).
Appendices

A SDSL Benchmark Results

A.1 Suffix Array Benchmarks

The suffix array benchmarks provided with the SDSL library [GBMP14] were ran for the implementation of Rao’s suffix array [Rao02] as well as some of the suffix array implementations included in SDSL. The suffix arrays were based on either a Huffman-shaped (wt_huff) or a run-length encoded wavelet tree (wt_rlmn) [MN05]. In addition, Sadakane’s suffix array (csa_sada) [Sad03] was tested.

A.1.1 Count Benchmark

The count benchmark was done for all the listed compressed suffix array types with fixed sampling rates for suffix array and inverse suffix array values. The tested types have been listed in Table 9. The results are shown in Table 8.

A.1.2 Locate Benchmark

The tested types were the same as in the count benchmark but the sampling rates were varied. The similar parameters of Rao’s suffix array depend on the content length, so they were not varied. The smallest rate used was 256 and the largest one was 2. The tested types have been listed in Table 7 and the results are shown in Figure 6.

A.1.3 Extract Benchmark

The test setup was similar to that of the locate benchmark. The tested types were the same as listed in Table 7. The results are shown in Figure 7.
Table 7: Class definitions of the indices used in the experiment. The sampling rates $S_{SA}$ and $S_{ISA}$ for suffix and inverse suffix array values were varied.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SDSL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FM-HF-BV</td>
<td><code>csa_wt &lt; wt_huff &lt; bit_vector, rank_support_v5,, select_support_scan &gt;, select_support_scan &lt;0&gt; &gt;, S_SA, S_ISA</code></td>
</tr>
<tr>
<td>FM-HF-BV-MCL</td>
<td><code>csa_wt &lt; wt_huff &lt; bit_vector, rank_support_v5,, select_support_mcl &gt;, select_support_mcl &lt;0&gt; &gt;, S_SA, S_ISA</code></td>
</tr>
<tr>
<td>FM-RLMN</td>
<td><code>csa_wt &lt; wt_rlmn,, S_SA, S_ISA&gt;</code></td>
</tr>
<tr>
<td>CSA-Sada-Elias-Delta-64</td>
<td><code>csa_sada &lt; enc_vector &lt; coder::elias_delta, 64 &gt;, S_SA, S_ISA</code></td>
</tr>
<tr>
<td>CSA-Sada-Elias-Delta-128</td>
<td><code>csa_sada &lt; enc_vector &lt; coder::elias_delta, 128 &gt;, S_SA, S_ISA</code></td>
</tr>
<tr>
<td>CSA-Rao</td>
<td><code>csa_rao &lt;&gt;</code></td>
</tr>
</tbody>
</table>
Table 8: Time in $\mu$sec per pattern symbol in a count query. Index space as fraction of original file size. Compile options: `-std=c++14 -O2 -march=native -g`.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time ($\mu$s)</td>
<td>Space (%)</td>
<td>Time ($\mu$s)</td>
<td>Time ($\mu$s)</td>
<td>Time ($\mu$s)</td>
<td>Time ($\mu$s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E.coli 25.954</td>
<td>1.445</td>
<td>34</td>
<td>2.043</td>
<td>0.215</td>
<td>0.207</td>
<td>0.813</td>
</tr>
<tr>
<td>DNA.100MB 24.919</td>
<td>1.681</td>
<td>56</td>
<td>2.504</td>
<td>0.206</td>
<td>0.203</td>
<td>0.979</td>
</tr>
<tr>
<td>DNA.200MB 31.139</td>
<td>1.773</td>
<td>56</td>
<td>2.591</td>
<td>0.224</td>
<td>0.224</td>
<td>1.055</td>
</tr>
<tr>
<td>PROTEINS.200MB</td>
<td>1.722</td>
<td>71</td>
<td>2.508</td>
<td>0.475</td>
<td>0.473</td>
<td>1.520</td>
</tr>
</tbody>
</table>

Table 9: Index identifier and corresponding SDSL type.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SDSL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSA-Rao</td>
<td>csa_rao &lt;&gt;</td>
</tr>
<tr>
<td>CSA-Sada-Elias-Delta-64</td>
<td>csa_sada &lt;enc_vector &lt;coder::elias_delta, 64&gt;, 1 &lt;&lt; 20, 1 &lt;&lt; 20&gt;</td>
</tr>
<tr>
<td>CSA-Sada-Elias-Delta-128</td>
<td>csa_sada &lt;enc_vector &lt;coder::elias_delta, 128&gt;, 1 &lt;&lt; 20, 1 &lt;&lt; 20&gt;</td>
</tr>
<tr>
<td>FM-HF-BV</td>
<td>csa_wt &lt;wt_huff &lt;bit_vector, rank_support_v5 &lt;&gt;&gt;, select_support_scan &lt;&gt;&gt;, select_support_scan &lt;0&gt;&gt;, 1 &lt;&lt; 20, 1 &lt;&lt; 20&gt;</td>
</tr>
<tr>
<td>FM-HF-BV-MCL</td>
<td>csa_wt &lt;wt_huff &lt;bit_vector, rank_support_v5 &lt;&gt;&gt;, select_support_mcl &lt;&gt;&gt;, select_support_mcl &lt;0&gt;&gt;, 1 &lt;&lt; 20, 1 &lt;&lt; 20&gt;</td>
</tr>
<tr>
<td>FM-RLMN</td>
<td>csa_wt &lt;wt_rlmn &lt;&gt;&gt;, 1 &lt;&lt; 20, 1 &lt;&lt; 20&gt; &lt;&gt;</td>
</tr>
</tbody>
</table>
Figure 6: Time-space trade-offs for operation locate.
Figure 7: Time-space trade-offs for operation extract.
A.2 Suffix Tree Benchmarks

The suffix tree benchmarks provided with the SDSL library were also ran for various types of compressed suffix trees. The tested suffix trees were Ohlebusch, Fischer and Gog’s (cst_sct3) [OFG10], Sadakane’s (cst_sada) [Sad07] and Russo, Navarro and Oliveira’s (cst_fully) [RNO11]. A Huffman-shaped wavelet tree (wt_huff) was used with all of these. In addition, Rao’s compressed suffix array was tested with the two first-mentioned suffix arrays. In this test the sampling rates were not varied but were fixed to 4 instead.

The tested types have been listed in Table 10. The index sizes are listed in Table 11 and the results are shown in Table 12.

Table 10: Class definitions of the indices used in the experiment. The sampling rates $S_{SA}$ and $S_{ISA}$ for suffix and inverse suffix array values were varied.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>SDSL type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCT3-WT</td>
<td>cst_sct3 &lt;</td>
</tr>
<tr>
<td></td>
<td>csa_wt &lt;</td>
</tr>
<tr>
<td></td>
<td>wt_huff &lt;</td>
</tr>
<tr>
<td></td>
<td>bit_vector,</td>
</tr>
<tr>
<td></td>
<td>rank_support_v5 &lt;&gt;</td>
</tr>
<tr>
<td></td>
<td>&gt;,</td>
</tr>
<tr>
<td></td>
<td>4,</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>SCT3-WT-MCL</td>
<td>cst_sct3 &lt;</td>
</tr>
<tr>
<td></td>
<td>csa_wt &lt;</td>
</tr>
<tr>
<td></td>
<td>wt_huff &lt;</td>
</tr>
<tr>
<td></td>
<td>bit_vector,</td>
</tr>
<tr>
<td></td>
<td>rank_support_v5 &lt;&gt;</td>
</tr>
<tr>
<td></td>
<td>&gt;,</td>
</tr>
<tr>
<td></td>
<td>4,</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>&gt;,</td>
</tr>
<tr>
<td></td>
<td>lcp_dac &lt;,</td>
</tr>
<tr>
<td></td>
<td>bp_support_sada &lt;,</td>
</tr>
<tr>
<td></td>
<td>bit_vector,</td>
</tr>
<tr>
<td></td>
<td>rank_support_v5 &lt;,</td>
</tr>
<tr>
<td></td>
<td>select_support_mcl &lt;&gt;</td>
</tr>
<tr>
<td></td>
<td>&gt;,</td>
</tr>
</tbody>
</table>

A6
SCT3-Rao
cst_sct3 <
    csa_rao <csa_rao_spec >>,
    lcp_dac >,
    bp_support_sada >,
    bit_vector,
    rank_support_v5 >,
    select_support_mcl >
>
Sada-WT
cst_sada <
    csa_wt <
        wt_huff <
            bit_vector,
            rank_support_v5 >,
        >,
        4,
        4
    >
>
Sada-Rao
cst_sada <
    csa_rao <csa_rao_spec >>,
    lcp_support_sada >
>
Fully
cst_fully <
csa_wt <
    wt_huff <
        bit_vector,
        rank_support_v5 >,
    >,
    4,
    4,
    text_order_sa_sampling >,
    text_order_isa_sampling_support >
>
>
### Table 11: Space Requirements of CSTs.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>DNA</th>
<th>PROTEINS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>100MB</td>
<td>200MB</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>$n/2^{20}$</td>
<td>100.0</td>
<td>200.0</td>
</tr>
<tr>
<td>$</td>
<td>T</td>
<td>/2^{20}$</td>
</tr>
<tr>
<td>SCT3-WT (MB)</td>
<td>325.28</td>
<td>670.92</td>
</tr>
<tr>
<td>SCT3-WT-MCL (MB)</td>
<td>327.21</td>
<td>674.80</td>
</tr>
<tr>
<td>SCT3-Rao (MB)</td>
<td>694.10</td>
<td>1429.39</td>
</tr>
<tr>
<td>Sada (MB)</td>
<td>153.27</td>
<td>307.23</td>
</tr>
<tr>
<td>Sada-WT (MB)</td>
<td>291.32</td>
<td>595.38</td>
</tr>
<tr>
<td>Sada-Rao (MB)</td>
<td>658.21</td>
<td>1349.97</td>
</tr>
<tr>
<td>Fully (MB)</td>
<td>142.69</td>
<td>292.34</td>
</tr>
</tbody>
</table>

### Table 12: Query times of CSTs, in microseconds.

<table>
<thead>
<tr>
<th>Test</th>
<th>CST</th>
<th>DNA</th>
<th>PROTEINS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>100MB</td>
<td>200MB</td>
</tr>
<tr>
<td>LCA</td>
<td>SCT3-WT</td>
<td>0.52</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>SCT3-WT-MCL</td>
<td>0.51</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>SCT3-Rao</td>
<td>0.54</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>Sada</td>
<td>0.26</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td>Sada-WT</td>
<td>0.28</td>
<td>0.26</td>
</tr>
<tr>
<td></td>
<td>Sada-Rao</td>
<td>0.27</td>
<td>0.26</td>
</tr>
<tr>
<td></td>
<td>Fully</td>
<td>46.54</td>
<td>59.09</td>
</tr>
<tr>
<td>Letter</td>
<td>SCT3-WT</td>
<td>1.58</td>
<td>1.65</td>
</tr>
<tr>
<td></td>
<td>SCT3-WT-MCL</td>
<td>2.07</td>
<td>1.65</td>
</tr>
<tr>
<td></td>
<td>SCT3-Rao</td>
<td>1.58</td>
<td>1.68</td>
</tr>
<tr>
<td></td>
<td>Sada</td>
<td>11.71</td>
<td>12.64</td>
</tr>
<tr>
<td></td>
<td>Sada-WT</td>
<td>2.26</td>
<td>2.22</td>
</tr>
<tr>
<td></td>
<td>Sada-Rao</td>
<td>2.05</td>
<td>2.15</td>
</tr>
<tr>
<td></td>
<td>Fully</td>
<td>59.05</td>
<td>69.86</td>
</tr>
<tr>
<td>-------</td>
<td>-------------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td><strong>SLink</strong></td>
<td>SCT3-WT</td>
<td>1.99</td>
<td>2.10</td>
</tr>
<tr>
<td></td>
<td>SCT3-WT-MCL</td>
<td>2.85</td>
<td>2.08</td>
</tr>
<tr>
<td></td>
<td>SCT3-Rao</td>
<td>3.61</td>
<td>3.89</td>
</tr>
<tr>
<td></td>
<td>Sada</td>
<td>1.53</td>
<td>1.57</td>
</tr>
<tr>
<td></td>
<td>Sada-WT</td>
<td>1.98</td>
<td>1.92</td>
</tr>
<tr>
<td></td>
<td>Sada-Rao</td>
<td>3.59</td>
<td>3.76</td>
</tr>
<tr>
<td></td>
<td>Fully</td>
<td>59.00</td>
<td>68.85</td>
</tr>
<tr>
<td><strong>Child</strong></td>
<td>SCT3-WT</td>
<td>2.92</td>
<td>3.06</td>
</tr>
<tr>
<td></td>
<td>SCT3-WT-MCL</td>
<td>2.92</td>
<td>3.10</td>
</tr>
<tr>
<td></td>
<td>SCT3-Rao</td>
<td>2.70</td>
<td>3.02</td>
</tr>
<tr>
<td></td>
<td>Sada</td>
<td>15.52</td>
<td>16.67</td>
</tr>
<tr>
<td></td>
<td>Sada-WT</td>
<td>3.76</td>
<td>3.69</td>
</tr>
<tr>
<td></td>
<td>Sada-Rao</td>
<td>3.50</td>
<td>3.51</td>
</tr>
<tr>
<td></td>
<td>Fully</td>
<td>69.26</td>
<td>81.46</td>
</tr>
<tr>
<td><strong>SDepth</strong></td>
<td>SCT3-WT</td>
<td>0.25</td>
<td>0.27</td>
</tr>
<tr>
<td></td>
<td>SCT3-WT-MCL</td>
<td>0.33</td>
<td>0.27</td>
</tr>
<tr>
<td></td>
<td>SCT3-Rao</td>
<td>0.33</td>
<td>0.27</td>
</tr>
<tr>
<td></td>
<td>Sada</td>
<td>9.14</td>
<td>9.41</td>
</tr>
<tr>
<td></td>
<td>Sada-WT</td>
<td>1.20</td>
<td>1.18</td>
</tr>
<tr>
<td></td>
<td>Sada-Rao</td>
<td>1.38</td>
<td>1.43</td>
</tr>
<tr>
<td></td>
<td>Fully</td>
<td>55.58</td>
<td>66.84</td>
</tr>
<tr>
<td><strong>Parent</strong></td>
<td>SCT3-WT</td>
<td>0.56</td>
<td>0.58</td>
</tr>
<tr>
<td></td>
<td>SCT3-WT-MCL</td>
<td>0.76</td>
<td>0.58</td>
</tr>
<tr>
<td></td>
<td>SCT3-Rao</td>
<td>0.55</td>
<td>0.59</td>
</tr>
<tr>
<td></td>
<td>Sada</td>
<td>0.14</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td>Sada-WT</td>
<td>0.14</td>
<td>0.14</td>
</tr>
<tr>
<td></td>
<td>Sada-Rao</td>
<td>0.14</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>Fully</td>
<td>106.95</td>
<td>127.40</td>
</tr>
</tbody>
</table>