Department of Computer Science
Series of Publications A
Report A-2012-5

# Compressed Full-Text Indexes for Highly Repetitive Collections

Jouni Sirén

*To be presented, with the permission of the Faculty of Science of the University of Helsinki, for public criticism, in Auditorium XII, University Main Building, on June 29th, 2012, at 12 o'clock noon.*

University of Helsinki
Finland

**Supervisor**
Veli Mäkinen, University of Helsinki, Finland

**Pre-examiners**
Kunihiko Sadakane, National Institute of Informatics, Japan
Jorma Tarhio, Aalto University, Finland

**Opponent**
Giovanni Manzini, University of Eastern Piedmont, Italy

**Custos**
Veli Mäkinen, University of Helsinki, Finland

**Contact information**

Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
FI-00014 University of Helsinki
Finland

Email address: postmaster@cs.helsinki.fi
URL: http://www.cs.Helsinki.fi/
Telephone: +358 9 1911, telefax: +358 9 191 51120

ISSN 1238-8645
ISBN 978-952-10-8051-7 (paperback)
ISBN 978-952-10-8052-4 (PDF)
Computing Reviews (1998) Classification: E.1, E.4, H.3
Helsinki 2012
Unigrafia

## Compressed Full-Text Indexes for Highly Repetitive Collections

Jouni Sirén

Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
jouni.siren@cs.helsinki.fi
http://www.cs.helsinki.fi/jouni.siren/

PhD Thesis, Series of Publications A, Report A-2012-5
Helsinki, June 2012, 97 + 63 pages
ISSN 1238-8645
ISBN 978-952-10-8051-7 (paperback)
ISBN 978-952-10-8052-4 (PDF)

### Abstract

This thesis studies problems related to compressed full-text indexes. A full-text index is a data structure for indexing textual (sequence) data, so that the occurrences of any query string in the data can be found efficiently. While most full-text indexes require much more space than the sequences they index, recent compressed indexes have overcome this limitation. These compressed indexes combine a compressed representation of the index with some extra information that allows decompressing any part of the data efficiently. This way, they provide similar functionality as the uncompressed indexes, while using only slightly more space than the compressed data.

The efficiency of data compression is usually measured in terms of entropy. While entropy-based estimates predict the compressed size of most texts accurately, they fail with highly repetitive collections of texts. Examples of such collections include different versions of a document and the genomes of a number of individuals from the same population. While the entropy of a highly repetitive collection is usually similar to that of a text of the same kind, the collection can often be compressed much better than the entropy-based estimate.

Most compressed full-text indexes are based on the Burrows-Wheeler transform (BWT). Originally intended for data compression, the BWT has deep connections with full-text indexes such as the suffix tree and the suffix array.

With some additional information, these indexes can be simulated with the Burrows-Wheeler transform. The first contribution of this thesis is the first BWT-based index that can compress highly repetitive collections efficiently.

Compressed indexes allow us to handle much larger data sets than the corresponding uncompressed indexes. To take full advantage of this, we need algorithms for constructing the compressed index directly, instead of first constructing an uncompressed index and then compressing it. The second contribution of this thesis is an algorithm for merging the BWT-based indexes of two text collections. By using this algorithm, we can derive better space-efficient construction algorithms for BWT-based indexes.

The basic BWT-based indexes provide similar functionality as the suffix array. With some additional structures, the functionality can be extended to that of the suffix tree. One of the structures is an array storing the lengths of the longest common prefixes of lexicographically adjacent suffixes of the text. The third contribution of this thesis is a space-efficient algorithm for constructing this array, and a new compressed representation of the array.

In the case of individual genomes, the highly repetitive collection can be considered a sample from a larger collection. This collection consists of a reference sequence and a set of possible differences from the reference, so that each sequence contains a subset of the differences. The fourth contribution of this thesis is a BWT-based index that extrapolates the larger collection from the sample and indexes it.

**Computing Reviews (1998) Categories and Subject Descriptors:**

E.1 [Data Structures]: String data structures
E.4 [Coding and Information Theory]: Data compaction and compression — compressed data structures
H.3.3 [Information Storage and Retrieval]: Information search and retrieval — full-text indexes

**General Terms:**
data structures, data compression

**Additional Key Words and Phrases:**
compressed data structures, full-text indexes, string processing, suffix array, Burrows-Wheeler transform, highly repetitive collections

# Acknowledgements

I thank my supervisor Veli Mäkinen for the support and guidance. I also thank my coauthors Diego Arroyuelo, Francisco Claude, Paolo Ferragina, Sebastian Maneth, Gonzalo Navarro, Kim Nguyen, Rossano Venturini, and Niko Välimäki for the fruitful collaboration. In addition to these people, Travis Gagie, Riku Katainen, Serikzhan Kazi, Juha Kärkkäinen, Simon Puglisi, Giovanna Rosone, Leena Salmela, and the numerous anonymous referees deserve thanks for the ideas, advice, and feedback.

Special thanks go to the IT team of the Department of Computer Science, and especially to Jani Jaakkola, for the excellent IT infrastructure and the helpful support well beyond normal working hours.

During the course of my doctoral studies, I was affiliated with the Department of Computer Science at the University of Helsinki, Helsinki Institute for Information Technology (HIIT), Helsinki Doctoral School in Computer Science and Engineering (Hecse), Graduate School in Computational Biology, Bioinformatics, and Biometrics (ComBi), Finnish Doctoral Programme in Computational Sciences (FICS), Finnish Centre of Excellence for Algorithmic Data Analysis Research (Algodan), and Finnish Centre of Excellence in Cancer Genetics Research. My work was also partially funded by the Academy of Finland, the Research Foundation of the University of Helsinki, and the Nokia Foundation.

Finally, I would like to thank the Student Union of the University of Helsinki, the numerous other student organizations at the University, and Ropecon for all these years. Without these organizations and the fascinating people in them, I would have finished my studies much, much faster.

# Original Papers

This thesis consists of an introduction and the following peer-reviewed publications, which are referred to as Papers I–IV in the text. These publications are reproduced at the end of the thesis.

I. Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki: **Storage and Retrieval of Highly Repetitive Sequence Collections**.  
Journal of Computational Biology 17(3):281–308, 2010.  
Preliminary versions in SPIRE 2008 and RECOMB 2009.

II. Jouni Sirén: **Compressed Suffix Arrays for Massive Data**.  
16th Symposium on String Processing and Information Retrieval (SPIRE 2009), LNCS 5721, pp. 63–74, Springer, 2009.

III. Jouni Sirén: **Sampled Longest Common Prefix Array**.  
21st Annual Symposium on Combinatorial Pattern Matching (CPM 2010), LNCS 6129, pp. 227–237, Springer, 2010.

IV. Jouni Sirén, Niko Välimäki, and Veli Mäkinen: **Indexing Finite Language Representation of Population Genotypes**.  
11th Workshop on Algorithms in Bioinformatics (WABI 2011), LNCS 6833, pp. 270–281, Springer, 2011.

Implementations and data sets used in the experiments can be found at `http://www.cs.helsinki.fi/group/suds/rlcsa/` for Papers I–III, and at `http://www.cs.helsinki.fi/group/suds/gcsa/` for Paper IV.

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

When one thinks of indexes, the type of index found in books is often the first one to come into mind. These *inverted indexes* are basically a list of words, combined with a list of occurrences for each word. Such indexes are space-efficient, and allow fast word and phrase queries on texts with well-defined word boundaries, such as many natural languages.

Yet many types of texts, such as DNA sequences, programming languages, and some natural languages such as Chinese, lack clear or meaningful word boundaries. Even in some natural languages, such as Finnish or German, many of the clearly separated words are actually concatenations of the words one might want to search for.

*Full-text indexes* overcome many of the limitations of inverted indexes. They allow efficient queries for any substring of the text, making them useful not only for texts without good word boundaries, but for natural language texts as well. The most common full-text indexes include the *suffix tree* and the more limited *suffix array*. The *q-gram index*, basically an inverted index for all substrings of length $q$ occurring in the text, is another common full-text index.

The main drawback of full-text indexes is their size. A minimal implementation of the suffix array requires approximately $n \log n$ bits for a text of length $n$.[1] Suffix trees are several times larger. While this is acceptable for small texts, the index can become larger than the available memory in many potential applications. For example, the suffix tree for a human genome requires tens of gigabytes, making it much larger than the main memory in current desktop computers.

[1] In this thesis, all logarithms are base-2, unless otherwise specified.

One can try to use full-text indexes optimized for secondary memory [52] to overcome these limitations. However, these indexes can be slow in practice, as hard disks only allow about $10^2$ random reads per second. Even the new solid-state drives with $10^4$ to $10^5$ reads per second can be too slow for complex operations using these indexes.

Another solution is to use *compressed full-text indexes* [72] based on the *Burrows-Wheeler transform*, such as the *compressed suffix array* [35] and the *FM-index* [20].[2] Such indexes are often *self-indexes* that do not require the original text to operate. They support a number of queries, with the most common being the ones required for suffix array-like functionality:

- *counting* the number of occurrences of a pattern in the text,
- *locating* the occurrences, and
- *extracting* an arbitrary substring.

In addition to exact pattern matching, compressed suffix arrays support *approximate matching*, allowing a limited number of mismatching characters, insertions, and deletions between the pattern and the text. Such queries are especially important in *read alignment* that has been the most successful application of compressed suffix arrays in bioinformatics [53, 54, 59, 61, 60, 62]. Sequencing machines produce short DNA fragments that must be mapped to their most likely positions in the reference genome. Allowing mismatches and other errors is vital, as there can be sequencing errors, and the sequenced genome will be somewhat different from the reference.

The size of a compressed index is often given relative to the *empirical entropy* of the text.[3] For many common types of compressible texts, the size of these indexes is close to the size of the text compressed using common compressors such as *gzip* and *bzip2*.

However, this is not the case with *highly repetitive collections.* Informally, a collection of texts is highly repetitive, if most of the texts are highly similar to some other text in the collection. Examples of such collections include collections of individual genomes, web archives, and version control systems storing different versions of the same documents. Highly repetitive collections usually have large amounts of redundancy that cannot be captured by any fixed-order statistical model.

[2] In the rest of the thesis, compressed full-text indexes based on the Burrows-Wheeler transform will be called compressed suffix arrays.

[3] Empirical entropy states roughly that, given $k$ consecutive characters, how much uncertainty there is on the average over the next character in the sequence.

For such collections, empirical entropy is not a good measure of compressibility. For example, for a text $T$ of length $n$ with $H$ bits of entropy per character, the total entropy of a collection of $r$ identical copies of $T$ is approximately $rnH$ bits. A compressed index for such collection would also be roughly $rnH$ bits in size, making it $r$ times larger than the index for a single text.

However, common compression methods based on *Lempel-Ziv parsing* or the Burrows-Wheeler transform can utilize the large-scale redundancy in such collections. A good Lempel-Ziv-based compressor would compress the collection basically to the same size as a single text, while a Burrows-Wheeler-based compressor would make the collection at most $\log r$ times larger (see Lemma 4.1 and the analysis at the end of Section 2.3).

As the collections can be very large, there is a real need for indexes capable of compressing their large-scale redundancy. We also need new ways to measure the compressibility of such collections, and to analyze the size of the new indexes. Furthermore, as the compressed index can be much smaller than the original collection, it would be preferable to have fast construction algorithms, whose memory usage depends on the compressed size of the text instead of the original size, so that large collections can be indexed in practice.

## 1.2 Original papers and contributions

**Paper I.** We described run-length encoded variants of several compressed indexes. Compared to earlier run-length encoded proposals [84, 67], these indexes offer much better compression for highly repetitive collections. This is because the earlier proposals implicitly assumed that the compressed text requires at least $n$ bits of space for a text of length $n$, and hence using $o(n)$ bits for the additional structures would not increase the total size significantly.

Instead of empirical entropy, we used the number of equal letter runs in the Burrows-Wheeler transform of the collection as our complexity metric. We analyzed how various edit operations affect the number of runs, explaining the superior performance of our indexes with highly repetitive collections. Experiments on indexing the genomes of 36 strains of *Saccharomyces paradoxus* confirmed the theoretical results.

My contributions to the paper include one of the new indexes, which proved to be the most efficient of them in practice. I also did most of the analysis on the effect of edit operations on the number of runs, and performed most of the experiments.

**Paper II.** I described a fast algorithm for merging the Burrows-Wheeler transforms of two text collections, and used the merging algorithm as a basis for a space-efficient construction algorithm for compressed suffix arrays. The earlier construction algorithms could not handle more than a few gigabytes in practice, as they either were slow, or used much more memory than the size of the data. By using a parallel implementation of the merging algorithm, I was able to construct compressed suffix arrays for data sets of tens of gigabytes in size.

**Paper III.** The *longest common prefix array* is one of the main building blocks in several *compressed suffix tree* proposals [85, 27, 69, 76, 75]. In this paper, I adapted an earlier longest common prefix array construction algorithm [50] to use the compressed suffix array instead of the text and its suffix array. The resulting algorithm is the most space-efficient one known, but also slower than the alternatives. I also proposed a new compressed representation of the longest common prefix array that is faster to use than the alternatives, but still achieves similar compression in many cases.

**Paper IV.** In this paper, we generalized the compressed suffix arrays to index an automaton-based representation of finite languages. While the index can be exponentially larger than the automaton in the worst case, we showed that there are important cases where the growth remains minimal. An example of such cases is a finite automaton representing the genetic variation within a population, often constructed from a reference genome and a set of known variants.

My contributions to the paper include the generalized index, its construction algorithm, the analysis of the effect of mutation rate on the size of the index, and most of the experiments.

## 1.3 Outline

The rest of the thesis is organized as follows. Chapter 2 provides basic definitions and background to the thesis. Chapter 3 is a short survey on the most common types of compressed suffix arrays, discussing the basic techniques used and the most important results obtained with them.

Chapter 4 describes the run-length compressed suffix array, a compressed suffix array intended for highly repetitive collections. The discussion is based on Papers I and II. There is also a short description of other compressed indexes for highly repetitive collections at the end of the chapter.

Chapter 5 discusses the space-efficient compressed suffix array construction algorithm from Paper II. As an extension to the paper, we introduce a new variant of the algorithm, based on sorting the suffixes of the new texts by their lexicographic ranks among the suffixes of already indexed texts.

In Chapter 6, we discuss the space-efficient construction algorithm and the sampling technique for the longest common prefix array from Paper III. This chapter does not significantly extend the results in the original paper.

Chapter 7 describes the generalization of compressed suffix arrays for indexing finite automata from Paper IV. The variant of the index discussed here is both simpler and faster than in the original paper. Included is the analysis of the effect of mutation rate on the index size, which previously appeared only in the arXiv version of the paper [89].

The thesis ends with conclusions and future directions in Chapter 8.

# Chapter 2

# Background

## 2.1 Definitions

**Strings.** Let $\Sigma = \{1, \ldots, \sigma\}$ be an *alphabet* of size $\sigma$. A *string* $S = S[1, n]$ over alphabet $\Sigma$ is a *sequence* of *characters* $S[1] \cdots S[n]$, where $S[i] \in \Sigma$ for all $i$. For any string $S[1, n]$, we write $|S| = n$ to denote its length. A *substring* of $S$ is written as $S[i, j] = S[i] \cdots S[j]$. Important types of substrings include *prefixes* $S[1, j]$ and *suffixes* $S[i, n]$. The concatenation of two strings $S[1, n]$ and $S'[1, n']$ is written as $SS' = S[1] \cdots S[n]S'[1] \cdots S'[n']$.

We define the *lexicographic order* $<$ among strings in the usual way. For any two strings $S[1, n]$ and $S'[1, n']$, we have $S < S'$, if either $S[1] < S'[1]$ or $S[1] = S'[1]$ and $S[2, n] < S'[2, n']$. The *empty string* $\lambda$ of length 0 is a special case, with $\lambda < S$ for any non-empty string $S$ of length $|S| > 0$. To avoid this special case, we often consider *text* strings $T = T[1, n]$ that are terminated by an end marker $T[n] = \$ \not\in \Sigma$ with lexicographic value 0. The *lexicographic rank* of string $S$ among a collection (set) of strings $\mathcal{C}$ is the number of strings $S' \in \mathcal{C}$ such that $S' < S$, plus one. We often write $\mathrm{rank}(T, S)$ to denote the lexicographic rank of string $S$ among all suffixes of text $T$, and $\mathrm{rank}(\mathcal{C}, S)$ to denote the lexicographic rank of $S$ among the suffixes of all strings of collection $\mathcal{C}$.

The (Levenshtein) *edit distance* between two strings $S$ and $S'$ is the minimum number of edit operations required to transform string $S$ into string $S'$. Allowed edit operations include the *substitution* of one character with another, the *insertion* of one character into any position, and the *deletion* of one character. Any set of edit operations transforming string $S$ into string $S'$ can be represented as an *alignment* of the strings. This can be generalized for a set of strings, producing a *multiple alignment* of the strings (see Figure 2.1).

```
G A C G T A - C T G C A G A T G - T A A T G C
G A C G T A - - - G C A G A T G C T A A T C C
G A T G T A - C T G C T G A T G C T - - T G C
G A C - T A C C T G C A G - T G C T A A T C C
```

Figure 2.1: A multiple alignment of four sequences. Character $-$ denotes either a deletion in the current sequence or an insertion in some other sequence.

**Graphs.** A *graph* $G = (V, E)$ consists of a set $V = \{v_1, \ldots, v_{|V|}\}$ of *nodes* and a set $E \subset V^2$ of *edges* such that $(v, v) \notin E$ for all $v \in V$. We call $(u, v) \in E$ an edge from node $u$ to node $v$. A graph is *directed*, if edge $(u, v)$ is distinct from edge $(v, u)$. For every node $v \in V$, we define the *indegree* of the node $in(v)$ to be the number of incoming edges $(u, v)$, and the *outdegree* $out(v)$ to be the number of outgoing edges $(v, w)$.

In a *labeled* graph, we attach a *label* $\ell(v)$ to each node $v \in V$. A *path* $\mathsf{P} = u_1 \cdots u_{|\mathsf{P}|}$ is a sequence of nodes such that $(u_i, u_{i+1}) \in E$ for all $i < |\mathsf{P}|$. The label of path $\mathsf{P}$ is the string $\ell(\mathsf{P}) = \ell(u_1) \cdots \ell(u_{|\mathsf{P}|})$. A *cycle* is a path from a node to itself containing at least one other node. If a graph contains no cycles, it is called *acyclic*.

A *tree* is an undirected graph $G = (V, E)$ with exactly one path between any pair of nodes $u, v \in V$. In a *rooted* tree, one node $v_r \in V$ is selected as the *root* node. For any edge $(u, v) \in E$, where node $u$ is on the path from root to node $v$, we call node $u$ the *parent* of node $v$, and node $v$ a *child* of node $u$. An *internal* node is a node with children, while a *leaf* node is a node without children.

A *trie* is a rooted tree, where every edge $e \in E$ is labeled by a character $\ell(e) \in \Sigma$. For any internal node $v$, the edges leading to its children must have distinct labels. A path $\mathsf{P} = e_1 \cdots e_{|P|}$ from the root node to a leaf node is labeled by a string $\ell(\mathsf{P}) = \ell(e_1) \cdots \ell(e_{|\mathsf{P}|})$. The set of strings $L(G)$ contained in trie $G$ is the set of all path labels from the root to a leaf.

**Automata and languages.** A *finite automaton* is a directed labeled graph $A = (V, E)$.[1] The *initial node* $v_1$ is labeled with $\ell(v_1) = \#$ with lexicographic value $\sigma + 1$, while the *final node* $v_{|V|}$ is labeled with $\ell(v_{|V|}) = \$$. The rest of the nodes are labeled with characters from alphabet $\Sigma$. We assume that every node $v \in V$ is contained in some path from $v_1$ to $v_{|V|}$.

[1] Unlike the usual definition, we label nodes instead of edges.

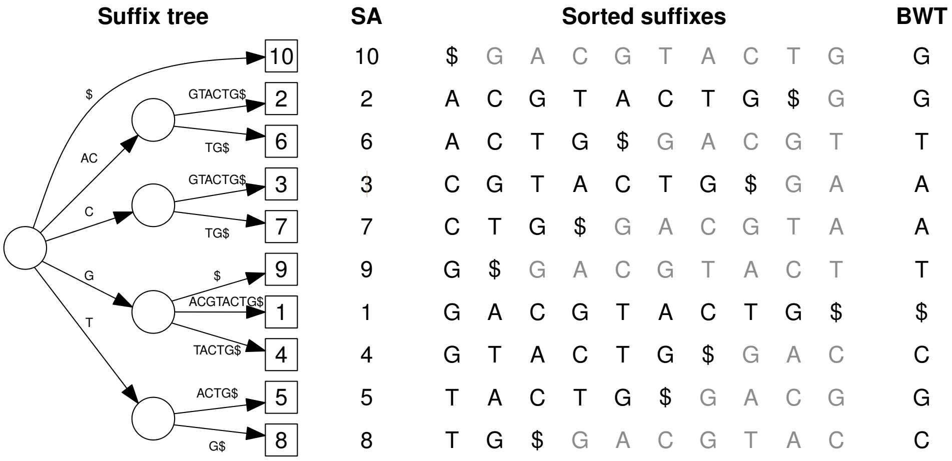
The *language* $L(A)$ *recognized* by automaton $A$ is the set of all path labels from $v_1$ to $v_{|V|}$. We say that automaton $A$ recognizes any string $S \in L(A)$, and that a suffix $S'$ can be recognized from node $v$, if there is a path from $v$ to $v_{|V|}$ with label $S'$. Note that all strings in the language are of form $\#x\$$, where $x$ is a string over alphabet $\Sigma$. If the language contains a finite number of strings, it is called *finite*. A language is finite if and only if the automaton recognizing it is acyclic. Two automata are said to be *equivalent*, if they recognize the same language.

Automaton $A$ is forward (reverse) *deterministic* if, for every node $v \in V$ and every character $c \in \Sigma \cup \{\#, \$\}$, there exists at most one node $u$ such that $\ell(u) = c$ and $(v, u) \in E$ $((u, v) \in E)$. For any language recognized by some finite automaton, we can always construct an equivalent automaton that is forward (reverse) deterministic.

## 2.2 Full-text indexes

Traditional indexes such as the *inverted index* treat the text as a sequence of words. Because of this limitation, they can only support queries based on words or their prefixes efficiently. If the text cannot be split into words, or if we want to search for arbitrary substrings, we must use *full-text indexes* instead. In later chapters, when we need to differentiate between the indexes of different texts, we put the original text or collection into subscript (e.g. $\mathsf{BWT}_T$ or $\mathsf{SA}_{\mathcal{C}}$).

**Suffix tree and suffix array.** The suffix trie of text $T[1, n]$ is a trie containing all suffixes of the text. As there are $\Theta(n^2)$ nodes in the worst case, the suffix trie is not a practical index for large texts. We get the *suffix tree (ST)* [92] by replacing every unary path $\mathsf{P}$ of the trie with a single edge $e$ with label $\ell(e) = \ell(\mathsf{P})$. As there is one leaf node for each of the $n$ suffixes and no unary internal nodes, there are at most $2n - 1$ nodes. If we store the edge labels as pointers to the text, we can store the suffix tree in $O(n \log n)$ bits. In practice, the size of a suffix tree is usually 10–20 bytes per character [47]. The suffix tree can be constructed in linear time with negligible working space in addition to the text and the final index [65, 91]. Given a *pattern* of length $|P|$, we can find the subtree containing its occurrences by following the edge labels in $O(|P|)$ time (assuming a constant-sized alphabet). If there are *occ* occurrences, we can list their positions in the text by traversing the subtree in $O(occ)$ time. An example of the suffix tree and some related indexes can be seen in Figure 2.2.



| SA | Sorted suffixes | BWT |
|---|---|---|
| 10 | $ G A C G T A C T G | G |
| 2 | A C G T A C T G $ G | G |
| 6 | A C T G $ G A C G T | T |
| 3 | C G T A C T G $ G A | A |
| 7 | C T G $ G A C G T A | A |
| 9 | G $ G A C G T A C T | T |
| 1 | G A C G T A C T G $ | $ |
| 4 | G T A C T G $ G A C | C |
| 5 | T A C T G $ G A C G | G |
| 8 | T G $ G A C G T A C | C |

Figure 2.2: Suffix tree, suffix array, and the Burrows-Wheeler transform of text `GACGTACTG$`.

The *suffix array (SA)* [63, 32] of text $T[1,n]$ is an array of pointers $\mathsf{SA}[1,n]$ to the suffixes of $T$ in lexicographic order. Alternatively, the suffix array is the set of the leaves of the suffix tree, listed according to the lexicographic order of the path labels from root to leaf. The suffix array for text $T[1,n]$ requires $n \log n$ bits of space in addition to the text, and can be constructed in $O(n)$ time with $2n$ bits of working space in addition to the text and the final index [74]. Given pattern $P$, we can find the range $\mathsf{SA}[sp,ep]$ containing the suffixes that have the pattern as their prefix in $O(|P| \log n)$ by using binary search. Other operations can be supported, but in general the suffix array is more limited than the suffix tree.

**Definition 2.1.** A data structure provides suffix array-like functionality, if it supports the following queries efficiently: (a) *find* the suffix array range $\mathsf{SA}[sp,ep]$ containing the suffixes prefixed by pattern $P$; (b) given $i$, *locate* suffix $\mathsf{SA}[i]$ in the text; and (c) given $i$ and $j$, *extract* substring $T[i,j]$.

An alternate definition for *locate* is to locate all occurrences of a pattern in the text. This definition is more useful for discussing indexes that are not based on the suffix array, such as the LZ-index (see Section 3.1). With these indexes, *find* is not a meaningful operation, and should be replaced by counting the number of occurrences of a pattern in the text, or just by determining whether there are any occurrences at all.

The suffix tree and the suffix array can be easily *generalized* to handle multiple sequences. Assume we are given a *collection* (a set) of texts

```
function find($P$)
    $[sp, ep] \leftarrow [C[P[|P|], C[P[|P|]+1]]$
    for $i \leftarrow |P| - 1$ to 1
        $sp \leftarrow C[P[i]] + \mathrm{rank}_{P[i]}(\mathsf{BWT}, sp - 1) + 1$
        $ep \leftarrow C[P[i]] + \mathrm{rank}_{P[i]}(\mathsf{BWT}, ep)$
        if $[sp, ep] = \emptyset$
            return $\emptyset$
    return $[sp, ep]$
```

Figure 2.3: Backward searching on Burrows-Wheeler transform [20]. The body of the loop constitutes one step of backward searching.

$T_1, \dots, T_r$, and let $\$_i$ be the end marker of text $T_i$. To have strict lexicographic ordering between the suffixes, we define $\$_i < \$_j$ whenever $i < j$. The generalized suffix trie is now simply a trie containing all suffixes of texts $T_1, \dots, T_r$. Generalized versions of the suffix tree and the suffix array are derived from the trie in the same way as above.

**Burrows-Wheeler transform.** *Burrows-Wheeler transform (BWT)* [6] is a permutation of the text closely related to the suffix array. The BWT of text $T[1, n]$ is a sequence $\mathsf{BWT}[1, n]$ such that $\mathsf{BWT}[i] = T[\mathsf{SA}[i] - 1]$, if $\mathsf{SA}[i] > 1$, and $\mathsf{BWT}[i] = T[n] = \$$ otherwise. The transform can be reversed by a permutation called *LF-mapping* [6, 20]. Let $C[0, \sigma + 1]$ be an array such that $C[c]$ is the number of characters in $\{\$, 1, 2, \dots, c-1\}$ occurring in the BWT, with $C[0] = C[\$] = 0$ and $C[\sigma + 1] = n$. We define $LF$-mapping as $LF(i) = C[\mathsf{BWT}[i]] + \mathrm{rank}_{\mathsf{BWT}[i]}(\mathsf{BWT}, i)$, where $\mathrm{rank}_c(\mathsf{BWT}, i)$ is the number of occurrences of character $c$ in prefix $\mathsf{BWT}[1, i]$.

The $\mathrm{rank}_{\mathsf{BWT}[i]}(\mathsf{BWT}, i)$ in the definition can be interpreted as the lexicographic rank of suffix $T[\mathsf{SA}[i], n]$ among those suffixes preceded by character $\mathsf{BWT}[i]$. Hence $LF(i)$ is the lexicographic rank of suffix $T[\mathsf{SA}[i]-1, n]$ (or $T[n]$, if $\mathsf{SA}[i] = 1$) among all suffixes of the text. This allows us to move from the suffix array position corresponding to suffix $T[\mathsf{SA}[i], n]$ to that of suffix $T[\mathsf{SA}[i] - 1, n]$ without using the text or its suffix array.

By using $LF$-mapping, we can support *find* with just arrays $C$ and $\mathsf{BWT}$ through *backward searching* [20] (see Figure 2.3). When searching for pattern $P$, the algorithm maintains an invariant that $\mathsf{SA}[sp_i, ep_i]$ is the range of suffixes prefixed by $P[i, |P|]$. If $\mathsf{BWT}[j]$ and $\mathsf{BWT}[j']$ are the first and the last occurrences of character $P[i - 1]$ in range $\mathsf{BWT}[sp_i, ep_i]$, then $\mathsf{SA}[LF(j), LF(j')]$ is the range of suffixes prefixed by $P[i-1, |P|]$. In Chapter 3, we show how backward searching can be implemented efficiently in compressed suffix arrays.

**Theorem 2.1.** *Assume that an index performs one step of backward searching in $t_B$ time. Then it supports* $\mathrm{find}(P)$ *in* $O(|P| \cdot t_B)$ *time.*

The inverse function of $LF$-mapping is $\Psi(i) = \mathrm{select}_c(\mathsf{BWT}, i - C[c])$, where $c$ is the highest value with $C[c] < i$, and $\mathrm{select}_c(\mathsf{BWT}, j)$ is the position of the $j$th occurrence of character $c$ in $\mathsf{BWT}$ [35]. We will often write $\mathrm{char}(i)$ to denote such character $c$. This function allows us to move from the suffix array position of suffix $T[\mathsf{SA}[i], n]$ to that of suffix $T[\mathsf{SA}[i] + 1, n]$. Function $\Psi$ is strictly increasing in the range $C_c = [C[c] + 1, C[c+1]]$ corresponding to suffixes starting with character $c \in \Sigma$. Note that $T[\mathsf{SA}[i]] = c$ and $\mathsf{BWT}[\Psi(i)] = c$ for every $i \in C_c$.

**Enhanced suffix array.** Let $\mathrm{lcp}(A, B)$ be the length of the longest common prefix of sequences $A$ and $B$. The *longest common prefix (LCP) array* of text $T[1, n]$ is an array $\mathsf{LCP}[1, n]$ such that $\mathsf{LCP}[1] = 0$ and $\mathsf{LCP}[i] = \mathrm{lcp}(T[\mathsf{SA}[i-1], n], T[\mathsf{SA}[i], n])$ for $i > 1$. The array requires $n \log n$ bits of space, and can be constructed in $O(n)$ time [43, 80, 50, 30, 26]. The best construction algorithms require little more space than for the text, the suffix array, and the LCP array. By using another $n \log n$-bit array derived from the LCP array, the time complexity of searching for pattern $P$ in a suffix array is reduced from $O(|P| \log n)$ to $O(|P| + \log n)$ [63].

The suffix array, the BWT, and the LCP array can be further augmented with various partial representations of suffix tree topology. This approach, generally known as the *enhanced suffix array* [1], allows us to simulate the suffix tree in the same asymptotic time, while using less space. Many compressed suffix tree proposals are based on a similar idea (see Chapter 6).

## 2.3 Compressed data structures

**Data structure compression.** The field of lossless data compression is centered on finding smaller representations for different types of data. Its three main goals are i) compression efficiency, ii) resources required for compressing the data, and iii) resources required for decompression. The relative importance of these goals varies greatly by the applications considered.

In data structure compression, we are not interested in just compressing the data, but we also want to support various operations on the data. Hence the third goal becomes an efficient support for these operations instead of just the efficiency of full decompression. The main tools used are compression methods that support partial decompression, and space-efficient indexes that allow us to quickly find the right part to decompress.

Most data structures contain no extra information in addition to the data they represent. For example, while the suffix array for text $T[1, n]$ requires $n \log n$ bits of space, we can construct it from the text requiring only $n \log \sigma$ bits of space. Hence it should be possible to represent the suffix array in roughly $n \log \sigma$ bits of space. A data structure that achieves this goal while providing efficient support for the required operations is called *succinct*. More generally, a data structure for data $D$ is succinct, if it takes $|D|(1+o(1))$ bits of space, where $|D|$ is the size of data in *bits*, and efficiently supports the required operations.

*Compressed data structures* are still more powerful than succinct ones. Let $H$ be a *complexity metric* that measures the repetitiveness of the data with respect to some model, and let $f$ be a function such that input data $D$ can be compressed into $f(H(D), |D|)$ bits. Then a data structure is compressed with respect to metric $H$, if it requires $O(f(H(D), |D|))$ bits of space, and supports the required operations efficiently. Later in this section, we discuss two complexity metrics for textual data: empirical entropy and the number of equal letter runs in the Burrows-Wheeler transform.

We call a succinct or compressed full-text index a *self-index*, if it does not require the original text to operate, and is able to fully reproduce the text. This allows us to replace the original text with the self-index that often requires less space.

**Empirical entropy.** The *empirical entropy* [64] $H_k(S)$ of sequence $S[1, n]$ is the average uncertainty over the next character in the sequence, given a *context* of $k \geq 0$ previous characters. If we encode each character in the sequence separately, while considering only the $k$ preceding characters during the encoding, then $nH_k(S)$ bits is a lower bound for the size of the compressed representation.

Order-0 empirical entropy is defined as

$$H_0(S) = -\sum_{c=1}^{\sigma} \frac{n_c}{|S|} \log \frac{n_c}{|S|},$$

where $n_c$ is the number of occurrences of character $c$ in sequence $S$, and $0 \log 0 = 0$. For $k > 0$, let $w \in \Sigma^k$ be a sequence, and let $w_S$ be the concatenation of the characters following the occurrences of $w$ in sequence

$S$. Order-$k$ empirical entropy for $k > 0$ is then defined as

$$H_k(S) = \sum_{w \in \Sigma^k} \frac{|w_S|}{|S|} H_0(w_S).$$

For any $k \geq 0$ and any sequence $S$, it holds that

$$0 \leq H_{k+1}(S) \leq H_k(S) \leq \log \sigma.$$

These definitions can be generalized for the *integer alphabet* $\mathbb{N} = \{0, 1, \ldots\}$. In the rest of this thesis, we will write $H_k$ instead of $H_k(S)$, if sequence $S$ is evident from the context.

**Runs in Burrows-Wheeler transform.** While the empirical entropy is a natural statistical metric of the compressibility of a sequence, it does not reflect well the large-scale repetitiveness of the sequence. Consider an arbitrary sequence $S[1, n]$ and its order-$k$ empirical entropy $H_k(S)$ for $k \ll n$. As $H_k(SS) \approx H_k(S)$, empirical entropy implies that the size of a compressed representation of the concatenation of two copies of sequence $S$ should be about twice that of a single sequence. However, it is evident that we can compress $SS$ to take only a little more space than the compressed representation of sequence $S$.

For sequences with large-scale repetitiveness, we define another complexity metric that is a natural structural property of suffix arrays and related structures. We consider the number of *equal letter runs* in the Burrows-Wheeler transform of the sequence. When a text is repetitive, the characters preceding lexicographically adjacent suffixes are identical with high probability. Hence the number of runs should be small when the text is repetitive.

Let $T[1, n]$ be a text, and let $\mathsf{BWT}$ be its Burrows-Wheeler transform. If $\mathsf{BWT}[i] = \mathsf{BWT}[i+1]$, then positions $i$ and $i+1$ belong to the same run. As $T[\mathsf{SA}[i], n]$ and $T[\mathsf{SA}[i+1], n]$ are lexicographically adjacent suffixes, and $T[\mathsf{SA}[i]-1] = T[\mathsf{SA}[i+1]-1]$, then $T[\mathsf{SA}[LF(i)], n]$ and $T[\mathsf{SA}[LF(i+1)], n]$ must also be lexicographically adjacent, and hence $LF(i) = LF(i+1) - 1$. This means that, for every run $\mathsf{BWT}[i, i+l-1]$, there is a corresponding *self-repetition* $\mathsf{SA}[i', i'+l-1]$ in the suffix array, with $\mathsf{SA}[i+j] = \mathsf{SA}[i'+j]+1$ for $0 \leq j \leq l-1$.

Let $R(T)$ be the number of equal letter runs in the BWT of text $T[1, n]$. If the text is evident from the context, we write $R$ instead of $R(T)$. In addition to the trivial upper bound $R \leq n$, another bound [67]

$$R \leq nH_k(T') + \sigma^k \quad \text{for all}\, k \geq 0,$$

where $T'$ is the reverse of text $T$, is also relevant for low-entropy texts.[2] See Section 4.2 for bounds for the number of runs in texts with large-scale repetitiveness.

**Encoding integers.** Many compression algorithms transform the data into a sequence of integers. These integers are then encoded into *binary strings* with a variety of methods, and the resulting strings are concatenated to form the compressed representation. Encoding scheme $g\colon \mathbb{N} \to \{0,1\}^*$ is *prefix-free*, if for any integers $i, j \in \mathbb{N}$, neither of the codes $g(i)$ and $g(j)$ is a prefix of the other. If the encoding scheme is prefix-free, then the compressed sequence can be decoded unambiguously one integer at a time.

*Huffman codes* [41] are based on binary trees. Initially, each integer $i \in \mathbb{N}$ with $n_i > 0$ occurrences forms a separate tree with one node. As long as there are multiple trees left, we select two trees $G_1$ and $G_2$ with the least number of occurrences $n(G_1)$ and $n(G_2)$, and replace them with a new tree $G$ with $n(G) = n(G_1) + n(G_2)$ occurrences. Tree $G$ consists of a root node with $G_1$ as its left subtree and $G_2$ as its right subtree. The final tree is a *Huffman tree* for frequencies $(n_i)_{i \in \mathbb{N}}$.

To read the codes from the Huffman tree, we label each edge from a node to its left child with 0, and each edge to right child with 1. The code for integer $i$ corresponding to leaf node $v_i$ is then the path label from root to node $v_i$. With the help of a lookup table, both encoding and decoding can be done in constant time per symbol.

Huffman codes are prefix-free and optimal among those prefix-free codes that encode each occurrence of an integer with the same binary string [41]. The average code length is less than $H_0 + 1$ bits per symbol. Assuming that the integers are bounded by some polynomial $poly(n)$, where $n$ is the length of the sequence, we can compress the sequence into $n(H_0 + 1) + O(\sigma \log n)$ bits, where $\sigma$ is the number of integers with a positive number of occurrences. The $O(\sigma \log n)$ term comes from storing the number of occurrences for each of the occurring integers.

*Elias codes* [14] assign fixed codes to all positive integers. They are useful in situations, where the distribution is not known at the time of encoding, or when storing the numbers of occurrences would require significant space. The implicit assumption behind Elias codes is that small integers are more common than large ones, which is often true in e.g. encoding the differences of an increasing sequence of integers.

[2] The term $\sigma^k$ is actually an upper bound for the number of different order-$k$ contexts appearing in the reverse text.

Let $b(i)$ be the length $|b(i)| = \lceil \log(i+1) \rceil$ binary representation of integer $i > 0$, and let $b'(i)$ be the same sequence without the leading 1-bit. The Elias *$\gamma$-code* for integer $i$ is the binary string $\gamma(i) = 0^{|b'(i)|}b(i)$ of length $2|b(i)| - 1$. These codes are prefix-free, and encoding and decoding them takes constant time with the help of a lookup table. If we replace the unary representation $0^{|b'(i)|}$ with another $\gamma$-code, we get *$\delta$-codes* $\delta(i) = \gamma(|b(i)|)b'(i)$ of length $\log i + O(\log \log i)$. In general, $\gamma$-codes are better when most of the integers are small, while $\delta$-codes require asymptotically less space (see also Section 3.2).

For an example, consider a simple *run-length encoding* of the BWT of text $T[1, n]$. For a run consisting of $l$ occurrences of character $c$, we output a pair of integers $(c, l)$. The character is encoded directly using $\lceil \log \sigma \rceil$ bits, while $\delta$-codes are used for encoding the length of the run. As logarithm is a concave function, the worst case for encoding a given number of runs is when all the runs are of similar length. For $R$ runs, the average length of a run is $n/R$ characters, so the encoding uses at most

$$R(\log \sigma + \log(n/R)) + O(R \log \log(n/R))$$

bits of space. See Section 4.1 for the size bound for another variant of run-length encoding.

# Chapter 3

# Compressed suffix arrays

## 3.1 Original indexes

The *compressed suffix arrays (CSA)* discussed in this chapter are compressed self-indexes that provide the functionality of the suffix array (Definition 2.1). They are based on backward searching on the Burrows-Wheeler transform (see Section 2.2), with their major differences in the solutions used to encode the BWT and support *rank* and *select* on it.

The first index in the CSA family was the compressed suffix array of Grossi and Vitter [35]. It was not a self-index, but encoded the suffix array of text $T[1, n]$ recursively in the following way:

- Binary string $B[1, n]$ is used to mark even suffix array values: $B[i] = 1$, if $\mathsf{SA}[i]$ is even.

- For odd suffix array values $\mathsf{SA}[i]$, we store $\Psi(i)$ instead. As $\Psi$ is strictly increasing in range $C_c$ for all $c \in \Sigma$, these values form $\sigma$ increasing sequences that can be encoded efficiently by using gap encoding (see Section 3.2). If odd $\mathsf{SA}[i]$ is needed, it can be computed as $\mathsf{SA}[\Psi(i)] - 1$.

- For even suffix array values $\mathsf{SA}[i]$, we store $\mathsf{SA}[i]/2$ in array $\mathsf{SA}'$. If even $\mathsf{SA}[i]$ is needed, it can be computed as $2 \cdot \mathsf{SA}'[rank_1(B, i)]$. Array $\mathsf{SA}'$ can either be stored explicitly or encoded recursively.

Sadakane later transformed the compressed suffix array into a self-index [84], and showed how to implement backward searching using $\Psi$ instead of $\mathsf{BWT}$ [83].

The *FM-index (FMI)* of Ferragina and Manzini [20] was a simultaneous development that was already a self-index. The BWT was split into blocks that were compressed separately. To support backward searching, the $\mathrm{rank}_c(\mathsf{BWT}, \cdot)$ value corresponding to the beginning of each block was stored explicitly for each character $c$. To compute *rank* for later positions, the block had to be decompressed. Later developments concentrated on improving the time/space trade-offs for large alphabets.

Because of these historical roots, papers on indexes of the CSA family still often discuss compressing $\Psi$, while papers on the FMI family discuss compressing the BWT. This is not a fundamental distinction, however. As function $\Psi$ is strictly increasing in range $C_c$ for all characters $c \in \Sigma$, we can represent this part of $\Psi$ as a binary string $B_c$, where $B_c[i] = 1$ if and only if $\Psi(j) = i$ for some $j \in C_c$. But as $\Psi(j) = \mathrm{select}_c(\mathsf{BWT}, j - C[c])$, where $j \in C_c$, this means that $B_c[i] = 1$ if and only if $\mathsf{BWT}[i] = c$. As most indexes in the CSA family store the values of $\Psi$ separately for each of the ranges $C_c$, they are essentially encoding the BWT by binary strings $B_c$. Hence the differences between the CSA family and the FMI family are more semantical than technical in nature.

We describe the CSA family, based on encoding the binary strings $B_c$, in more detail in Section 3.2. Section 3.3 discusses the FMI family, concentrating on supporting *rank* and *select* directly on $\mathsf{BWT}$ by using wavelet trees. There are also other proposals [31, 2, 59] to support *rank* and *select* on $\mathsf{BWT}$ that we will not consider here. Later sections describe the techniques used to support *locate* and *extract*, as well as dynamic compressed suffix arrays that allow various updates to the text.

In addition to the compressed suffix arrays, there are also compressed indexes that are based on Lempel-Ziv parsing of the text [71, 20, 82, 46]. In general, these *LZ-indexes* have faster *locate* and *extract* than the BWT-based indexes [18]. On the other hand, they do not support *find*, as they are not based on compressing the suffix array. These LZ-indexes are not further discussed in the thesis.

## 3.2 CSA family

**Bit vectors.** *Bit vectors* are the basic building block of compressed data structures. Built for a binary sequence $B[1, n]$, a bit vector provides efficient support for operations $\mathit{rank}_1(B, \cdot)$ and $\mathit{select}_1(B, \cdot)$. Some structures also support $\mathit{select}_0(B, \cdot)$, while $\mathit{rank}_0(B, \cdot)$ can be solved by reduction $\mathit{rank}_0(B, i) = i - \mathit{rank}_1(B, i)$. In the following, bit vector $B$ refers both to the binary sequence and the data structure.

Indexes of the CSA family use bit vectors $B_c$, where $B_c[i] = 1$ if and only if $\mathsf{BWT}[i] = c$, to represent the BWT. They reduce the basic operations $\mathit{rank}_c(\mathsf{BWT}, \cdot)$ and $\mathit{select}_c(\mathsf{BWT}, \cdot)$ to $\mathit{rank}_1(B_c, \cdot)$ and $\mathit{select}_1(B_c, \cdot)$, respectively. The best encoding for the bit vectors depends on the type of data, the size of the alphabet, and the desired time/space trade-off.

A basic succinct bit vector [42, 66, 8] consists of the binary sequence and separate $o(n)$-bit indexes for *rank* and *select*. For *rank*, we split the sequence into *large blocks* of $l = \log^2 n$ bits, and store the number of 1-bits before each block in array $R_l$. A large block is further divided into *small blocks* of $s = \log n/2$ bits, and the number of 1-bits in previous small blocks of the same large block is stored in array $R_s$. These arrays take $O(n/\log n)$ and $O(n \log\log n/\log n)$ bits, respectively. We can now solve

$$rank_1(B, i) = R_l[\lfloor i/l \rfloor] + R_s[\lfloor i/s \rfloor] + \mathrm{popcount}(B[s \cdot \lfloor i/s \rfloor, i]),$$

where $\mathrm{popcount}(S)$ is the number of 1-bits in sequence $S$, in constant time. In practice, this solution requires 3 or 4 random memory accesses per *rank*, depending on whether popcount is solved directly or by using lookup tables.

In *select*, large and small blocks consist of $\kappa = \log^2 n$ 1-bits and $\log^2 \kappa = O((\log\log n)^2)$ 1-bits, respectively. Arrays similar to $R_l$ and $R_s$ in *rank* are used to store the answer to *select* for the first 1-bit of each block. These arrays take $O(n/\log n)$ bits for large blocks and $O(n/\log\log n)$ bits for small blocks. A lookup table is used to answer *select* for the 1-bits within a small block. This solution takes $O(1)$ time and up to 4 random memory accesses.

An exception to the above are long blocks. If a large block spans more than $\log^4 n$ positions in the sequence, relative positions within it might take more than $O(\log\log n)$ bits each. But as there are at most $O(n/\log^2 n)$ 1-bits within such blocks, we can store their positions explicitly in $O(n/\log n)$ bits and ignore the small blocks. Similarly, if a small block spans more than $\log n$ positions, answering *select* within it might not be constant-time. Yet as there are at most $O(n(\log\log n)^2/\log n)$ 1-bits within these blocks, storing their relative positions takes at most $O(n(\log\log n)^3/\log n)$ bits.

As there are $\binom{n}{n_1}$ binary sequences of length $n$ with $n_1$ 1-bits, we can store the sequence in $\lceil \log \binom{n}{n_1} \rceil \le \lceil nH_0 \rceil$ bits. A simple entropy-compressed bit vector achieves $nH_0 + o(n)$ bits of space by storing each small block as a pair of integers $(i, j)$, where $i$ is the number of 1-bits in the block, and $j$ is the lexicographic rank of the block among those blocks with $i$ 1-bits. The bit vector of Raman et al. [81] requires $nH_0 + O(n \log\log n/\log n)$ bits of space, and supports *rank* and *select* in constant time.

An alternative approach is to use *gap encoding* (also called *differential encoding*) to compress the sequence. Each 1-bit is stored as the distance between it and the previous 1-bit, often by using $\gamma$-codes or $\delta$-codes. Indexes are built for large and small blocks to allow fast decoding. The relevant complexity metric here is

$$\text{gap}(B) = \sum_{i=1}^{n_1} \lceil \log(\mathit{select}_1(B,i) - \mathit{select}_1(B,i-1) + 1) \rceil,$$

where $\mathit{select}_1(B,0) = 0$. If the sequence is evident from the context, we write gap instead of gap$(B)$, as with the other complexity metrics. In the worst case, we have gap $\lessapprox n_1 \log(n/n_1) \leq nH_0$, but gap can get much smaller, if the 1-bits are not evenly spaced [37].[1] The bit vector of Gupta et al. [37] achieves sublogarithmic query complexity, while its space occupancy is gap $+O(n_1 \log(n/n_1)/\log n_1) + O(n_1 \log\log(n/n_1))$ bits.

Gap encoding essentially replaces *runs* of 0-bits with their lengths. If there are long runs of 1-bits, it may be beneficial to replace these runs as well. This technique is called *run-length encoding (RLE).* Bit vectors can use RLE directly or reduce it to gap encoding. One such reduction replaces binary sequence $B$ with another sequence $B'$ that contains an 1-bit at the first position of every run of 0-bits and 1-bits in $B$. The relevant complexity metrics are the number of runs of 1-bits $R(B)$ (or just $R$) and run$(B) =$ gap$(B')$. In the worst case, run$(B) \lessapprox 2R(B) \log(n/(2R(B)))$ by the same reasoning as with gap encoding.

**Choosing the bit vector.** The choice between various encodings depends on the sequence $B$ (see Okanohara et al. [77]). If $B$ is essentially random, succinct representation and entropy compression are both good choices. For $n_1 \approx n/2$, we have $H_0 \approx 1$, and both representations achieve similar size. On the other hand, if the distribution is biased, entropy compression can reduce the size significantly. For non-random $B$, gap encoding or run-length encoding may be able to exploit the structure at the cost of query performance. Note that while entropy compression and gap encoding both have $nH_0$ as the most significant term in their size bounds, their actual performances differ. For a *sparse* sequences, where $n_1 \ll n/2$, gap encoding is often a better choice, as it usually has larger small blocks and hence less overhead. Entropy compression tends to perform better in *dense* (non-sparse) sequences, as gap encoding has some overhead per 1-bit.

When the sequence $B$ is very compressible, low-order terms in the size bound can dominate the size of the bit vector. For example, the $O(n \log\log n/\log n)$ term of the bit vector of Raman et al. [81] is almost linear in $n$, so it dominates the overall size when $H_0 \ll 1$. In these cases, it

[1] A pessimistic non-approximate bound is gap $\leq n_1 \log(n/n_1) + n_1$.

is preferable to use a *data-aware* bit vector, where the size overhead scales either with the compressed size of the sequence or with some related property of the sequence. An example of a data-aware bit vector is the one by Gupta et al. [37], with the low-order term almost linear in the number of 1-bits $n_1$. The key for making a bit vector data-aware is to make the small blocks data-aware: they might contain a certain amount of compressed data or a certain number of 1-bits.

The size of gap encoded and run-length encoded bit vectors depends greatly on the integer codes they use. A common solution is to use either $\gamma$-codes or $\delta$-codes. While $\delta$-codes are asymptotically smaller, $\gamma$-codes have smaller code lengths for integers $i \in \{2, 3, 8, \ldots, 15\}$. For $i \geq 32$, $\delta$-codes are shorter than $\gamma$-codes. When used in compressed suffix arrays, $\gamma$-codes perform better for regular texts [28], while $\delta$-codes are better for low-entropy or highly repetitive texts [36].

**Self-indexes of the CSA family.** Assume that we want to use a bit vector that supports *rank* in time $t_R$ and *select* in time $t_S$ to encode the binary sequences $B_c$. To compute $\Psi(i)$, we must be able to determine efficiently the largest character $c$ with $C[c] < i$. One solution is to use bit vector $C'$, where $C'[i] = 1$ if $C[c] = i$ for some character $c > 0$.[2] Then $\Psi(i) = \mathrm{select}_1(B_{c'}, i - C[c'])$, where $c' = \mathrm{rank}_1(C', i)$. Hence $\Psi$ can be computed in $t_\Psi = O(t_R + t_S)$ time.

For backward searching, we need to compute $LF$ for the first and the last occurrences of character $P[i-1]$ in range $\mathsf{BWT}[sp_i, ep_i]$. This can be done as

$$\begin{aligned} sp_{i-1} &= C[P[i-1]] + \mathrm{rank}_1(B_{P[i-1]}, sp_i - 1) + 1; \\ ep_{i-1} &= C[P[i-1]] + \mathrm{rank}_1(B_{P[i-1]}, ep_i). \end{aligned}$$

Hence one step of backward searching takes $t_B = O(t_R)$ time.

**Theorem 3.1** (From Theorems 2.1 and 3.3)**.** *Let* rank *and* select *on binary sequences take $t_R$ and $t_S$ time, respectively. Then a self-index of the CSA family with sample rate $d$ supports* find$(P)$ *in $O(|P| \cdot t_R)$ time,* locate *in $O(d \cdot (t_R + t_S))$ time, and* extract$(i, j)$ *in $O((d + j - i)(t_R + t_S))$ time.*

See Section 3.4 for the definition of sample rate $d$.

To compute $LF(i)$ for an arbitrary position $i$, we need to know the character $\mathsf{BWT}[i]$. As the character can be encoded in any of the binary sequences $B_c$, this takes $t_{LF} = O(\sigma \cdot (t_R + t_S))$ time. Hence computing $LF$ is inefficient in the CSA family, except for sequences with small alphabets.

[2] This solution assumes that every character of alphabet $\Sigma$ appears in the text.

As the encoding of $\mathsf{BWT}$ consists of $\sigma$ bit vectors of length $n$, any $o(n)$-bit terms in the size bound of the bit vector will sum up to $o(\sigma n)$ bits. In practice, this can be much more than the $n \log \sigma$ bits of the original sequence. Hence indexes of the CSA family tend to use bit vectors that are at least partially data-aware, with non-constant *rank* and/or *select* times.

The best-known index of the CSA family is the *compressed suffix array of Sadakane (Sad-CSA)* [84, 83]. It uses a differential encoding of $\Psi$ that is essentially either a gap encoded or a run-length encoded representation of the bit vectors $B_c$. A run-length encoded Sad-CSA requires $nH_k + O(n \log\log\sigma)$ bits for $k \leq \alpha \log_\sigma n$ and some constant $0 < \alpha < 1$ [72]. The performance of Sad-CSA follows from Theorems 2.1 and 3.3 with $t_B = O(\log n)$, $t_R = t_S = O(1)$, and $t_\Psi = O(1)$. In practice, Sad-CSA provides good compression and supports *locate* and *extract* very efficiently, while losing to other indexes in *find* [18].

The *run-length compressed suffix array (RLCSA)* [69, 88] uses data-aware run-length encoded bit vectors to encode the sequences $B_c$. Intended for indexing highly repetitive sequences, RLCSA requires

$$R\left(\log\frac{\sigma n}{R} + \log\frac{n}{R} + O\left(\log\log\frac{\sigma n}{R}\right)\right)\left(1 + \frac{O(\log n)}{b}\right) + O(\sigma \log n)$$

bits of space, where $b$ is the block size in bits. For $b = O(\log n)$, the performance follows from Theorem 3.1 with $t_R = t_S = O(\log n)$. This index is described in detail in Chapter 4.

## 3.3 FMI family

**Wavelet trees.** The *wavelet tree* [34] is a versatile data structure that supports *rank* and *select* on general sequences. In its general form [22], a wavelet tree reduces *rank* and *select* on strings over alphabet $\Sigma$ to the same operations on strings over a smaller alphabet $\Sigma'$. An example of a wavelet tree can be seen in Figure 3.1.

**Definition 3.1.** A wavelet tree for string $S$ over alphabet $\Sigma$ with internal alphabet $\Sigma'$ is a rooted tree, where each internal node has at most $|\Sigma'|$ children, and the leaves are the characters of alphabet $\Sigma$. Let $v_1, \ldots, v_k$ be the children of the root node. The root is labeled with string $S'$, where $S'[i] = j$, if character $S[i]$ is in the subtree with $v_j$ as its root. The subtree of each child $v_j$ is a wavelet tree for the subsequence $S_j$ of $S$ that contains only the characters in that subtree.

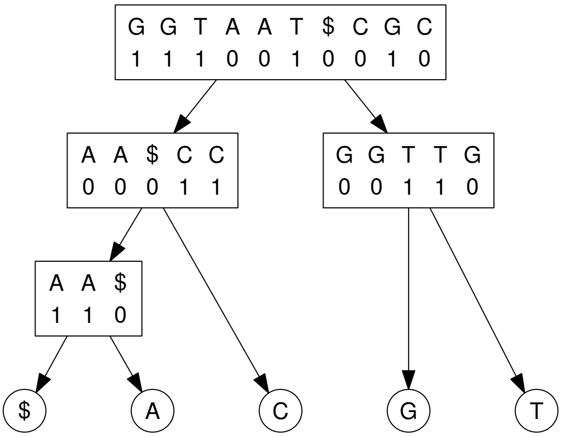


Figure 3.1: A binary wavelet tree for sequence `GGTAAT$CGC` (the Burrows-Wheeler transform in Figure 2.2). The internal nodes of an actual wavelet tree contain only the binary sequences, but not the conceptual character sequences.

Let $v_j$ be the root of the subtree containing character $c$. With the above definition, *rank* and *select* over string $S$ are supported in the following way:

$$\mathrm{rank}_c(S,i) = \mathrm{rank}_c(S_j, \mathrm{rank}_j(S',i));$$
$$\mathrm{select}_c(S,i) = \mathrm{select}_j(S', \mathrm{select}_c(S_j,i)).$$

The computation of *rank* starts at the root and proceeds towards the leaf, where $\mathrm{rank}_c(S,i) = i$, while *select* starts from the leaf with $\mathrm{select}_c(S,i) = i$ and proceeds towards the root. In either case, *rank* or *select* on string $S$ is reduced to $h$ operations with the internal alphabet, where $h$ is the distance of character $c$ from the root. Character $S[i]$ can also be determined in $h$ steps by starting from the root, continuing to the subtree of $v_{S'[i]}$, and searching for character $S_{S'[i]}[\mathrm{rank}_{S'[i]}(S',i)]$ there.

**Compression with wavelet trees.** Burrows-Wheeler transform groups characters followed by the same $k$-character context together, simultaneously for all $k \geq 0$. If we partition $\mathsf{BWT}$ by order-$k$ contexts for some fixed $k$, and encode each part separately by an order-0 encoder, the result will be compressed to the order-$k$ entropy of the reverse text, excluding low-order terms. Compression boosting [17] improves upon this by finding the partition that minimizes the overall size for a given order-0 encoder.

In practice, a quickly adapting order-0 encoder uses an almost optimal partition implicitly, achieving similar compression as explicit boosting [16].

We can use this implicit compression boosting to achieve $nH_k + o(n \log \sigma) + O(\sigma^{k+1} \log n)$ bits of space simultaneously for all $k \geq 0$ by using the bit vector of Raman et al. [81] to encode a binary wavelet tree [68]. This means that, in some sense, indexes of the FMI family can achieve similar compression as the best BWT-based compressors.

**Building the self-index.** Assume that the wavelet tree is a complete binary tree, and the sequences are encoded by a bit vector supporting *rank* in time $t_R$ and *select* in time $t_S$. Then we can compute $LF(i) = C[\mathsf{BWT}[i]] + \mathrm{rank}_{\mathsf{BWT}[i]}(\mathsf{BWT}, i)$ with $\log \sigma$ *rank* operations in $t_{LF} = O(t_R \log \sigma)$ time by recursion

$$\mathrm{rank}_{\mathsf{BWT}[i]}(\mathsf{BWT}, i) = \mathrm{rank}_{\mathsf{BWT}[i]}(S_{S'[i]}, \mathrm{rank}_{S'[i]}(S', i)).$$

With array $C$ implemented in the same way as in the CSA family, we can compute $\Psi(i) = \mathrm{select}_c(\mathsf{BWT}, i - C[c])$ in $t_\Psi = O(t_R + t_S \log \sigma)$ time. A step of backward searching

$$\begin{aligned} sp_{i-1} &= C[P[i-1]] + \mathrm{rank}_{P[i-1]}(\mathsf{BWT}, sp_i - 1) + 1; \\ ep_{i-1} &= C[P[i-1]] + \mathrm{rank}_{P[i-1]}(\mathsf{BWT}, ep_i). \end{aligned}$$

can be done in $t_B = O(t_R \log \sigma)$ time.

**Theorem 3.2** (From Theorems 2.1 and 3.3)**.** *Let* rank *and* select *on binary sequences take $t_R$ and $t_S$ time, respectively. Then a self-index of the FMI family with sample rate $d$ supports* find$(P)$ *in $O(|P| \cdot t_R \log \sigma)$ time,* locate *in $O(d \cdot t_R \log \sigma)$ time, and* extract$(i, j)$ *in $O((d + j - i) \cdot t_R \log \sigma)$ time.*

See Section 3.4 for the definition of sample rate $d$.

With a Huffman-shaped wavelet tree, we can replace the $\log \sigma$ factors in time complexities by the order-0 entropy $H_0$ in the expected case [28]. Note that as the total length of the bit vectors is $n \log \sigma$, the sublinear terms in size bounds sum up to $o(n \log \sigma)$. Hence, unlike in the CSA of Theorem 3.1, we can use the bit vectors with $t_R = t_S = O(1)$ to overcome the extra $\log \sigma$ factors in time complexities.

**Self-indexes of the FMI family.** The *succinct suffix array (SSA)* [67] is a simple Huffman-shaped wavelet tree using succinct bit vectors with $t_R = t_S = O(1)$. It requires $n(H_0 + 1)(1 + o(1))$ bits of space, and its performance follows from Theorem 3.2. In practice, SSA is one of the fastest self-indexes, due to its simplicity [18]. However, other indexes achieve better compression on texts with low high-order entropy.

The *run-length FM-index (RLFM)* [67] is a variant of the SSA that builds the wavelet tree over the *run heads* (the first characters of each run) of the BWT, and uses a separate bit vector to encode the lengths of the runs. Its performance follows from Theorem 3.2 with $t_R = t_S = O(1)$, while the size bound is $nH_k \log \sigma + 2n + o(n \log \sigma)$ bits for $k \le \log_\sigma n - \omega(1)$. In practice, the query performance of RLFM is similar to AFFM below [10]. Index size tends to be larger than AFFM on regular texts [10] and smaller on highly repetitive texts [90]. A data-aware variant of RLFM exists, but it is both larger and slower than RLCSA [69].

Instead of a single wavelet tree, the *alphabet-friendly FM-index (AFFM)* [22] uses explicit compression boosting to partition the BWT, and encodes each part separately with a wavelet tree with internal alphabet of size $o(\log n / \log \log n)$. Compression boosting guarantees a size bound of $nH_k + o(n \log \sigma)$ bits for $k \le \alpha \log_\sigma n - 1$, while the performance of AFFM follows from Theorems 2.1 and 3.3 with $t_B = t_{LF} = O(1 + \log \sigma / \log \log n)$ and $t_R = t_S = O(1)$. AFFM achieves similar compression as Sad-CSA, with fast *find* and relatively slow *locate* and *extract* [18].

There are several variants of SSA using compression boosting. With the implicit compression boosting offered by the bit vector of Raman et al. [81] on a balanced wavelet tree, we get the same size bound as for AFFM. The query performance of this FMI variant follows from Theorem 3.2 with $t_R = t_S = O(\log \sigma)$. Another variant (SSA-RRR) using a Huffman-shaped tree improves both size and performance, making the index smaller than AFFM and RLFM, but also slower than them [10]. We get the same size bound by dividing the BWT into blocks of $\sigma \log^2 n$ characters and building a separate SSA or SSA-RRR for each of the blocks [51]. With SSA-RRR as the basic index, this variant is the smallest entropy-compressed index of the FMI family, and almost as fast as AFFM. When using SSA, we combine the size of AFFM with the performance of SSA.

A data-aware run-length encoded variant of SSA also exists [69]. While this index offers slightly better compression than RLCSA, it is much slower in practice.

## 3.4 Supporting full functionality

**Sampling mechanism.** The basic solution to *locate* and *extract* has not changed much since the original FM-index [20]. We sample a number of pairs $(i, \mathsf{SA}[i])$, and use either $LF$ or $\Psi$ to derive the unsampled positions.

Assume that we want to retrieve $\mathsf{SA}[i]$. If suffix array position $i$ is sampled, we can just use the sampled value. Otherwise we compute $LF(i)$

and continue from that position. Eventually, after $k$ steps, we find a sample $(LF^k(i), \mathsf{SA}[LF^k(i)])$. As $\mathsf{SA}[LF(i)] = \mathsf{SA}[i] - 1$ (unless $\mathsf{SA}[i] = 1$), we now know that $\mathsf{SA}[i] = \mathsf{SA}[LF^k(i)] + k$. The special case can be avoided by always sampling $(\mathsf{SA}^{-1}[1], 1)$. In a similar way, we can also use $\Psi$ to find $\mathsf{SA}[i]$. As $\mathsf{SA}[\Psi(i)] = \mathsf{SA}[i] + 1$ (unless $\mathsf{SA}[i] = n$), we get $\mathsf{SA}[i] = \mathsf{SA}[\Psi^k(i)] - k$, where $(\Psi^k(i), \mathsf{SA}[\Psi^k(i)])$ is a sampled position.

To extract substring $T[i, j]$, we find the smallest $k \geq j$, for which $(SA^{-1}[k], k)$ has been sampled, and use $LF$ to proceed backwards. After $k - j$ steps, we have reached $(LF^{k-j}(\mathsf{SA}^{-1}[k]), j)$. We can now determine $T[j] = c$, where $C[c] < LF^{k-j}(\mathsf{SA}^{-1}[k]) \leq C[c + 1]$. After that, we proceed with $LF$ until we reach $T[i]$, and determine each character in the same way. Instead of using $LF$, we can also use $\Psi$ by starting at the largest sampled value $k \leq i$ and moving forward until we reach $T[j]$.

A practical solution is to sample $(i, \mathsf{SA}[i])$ for those positions $i$, where $\mathsf{SA}[i] = j \cdot d$ for some *sample rate* $d > 0$, in addition to the special cases $(\mathsf{SA}^{-1}[1], 1)$ and $(1, n)$. Then the nearest sample is always within $d - 1$ steps of $LF$ or $\Psi$. The sampled positions are marked in bit vector $B_s$ ($B_s[i] = 1$, if $(i, \mathsf{SA}[i])$ has been sampled). Sampled values are encoded as $\mathsf{SA}[i]/d$, taking $\log(n/d)$ bits each, and stored in array $\mathsf{SA}_s$ in suffix array order. Checking whether $\mathsf{SA}[i]$ has been sampled takes $O(t_R)$ time, where $t_R$ is the time required to answer $\mathrm{rank}_1(B_s, i)$. Retrieving a sampled value $\mathsf{SA}[i] = d \cdot \mathsf{SA}_s[\mathrm{rank}_1(B_s, i)]$ also takes $O(t_R)$ time.

For *extract*, the starting position $k$ is $d \cdot (\lfloor (j-1)/d \rfloor + 1)$ when using $LF$ and $d \cdot \lfloor i/d \rfloor$ when using $\Psi$. For all samples $(\mathsf{SA}^{-1}[k], k)$, we store the values $\mathrm{rank}_1(B_s, \mathsf{SA}^{-1}[k])$ in an array in text order. This array $\mathsf{SA}_s^{-1}$ takes a total of $(n/d)\log(n/d)$ bits. We can determine $\mathsf{SA}^{-1}[k] = \mathit{select}_1(B_s, \mathsf{SA}_s^{-1}[k/d])$ for a sampled text position $k$ in $O(t_S)$ time, where $t_S$ is the time required to answer *select*. The current character can be determined in $O(t_R)$ time by using the bit vector representation of array $C$.

**Theorem 3.3.** *Assume that an index computes $LF$ in $t_{LF}$ time and $\Psi$ in $t_\Psi$ time, and that we are given a bit vector that supports* rank *in $t_R$ time and* select *in $t_S$ time. Then we can answer* locate *in $O(d \cdot (t_R + \min(t_{LF}, t_\Psi)))$ time and extract substring $T[i, j]$ in $O(t_S + (d + j - i)(t_R + \min(t_{LF}, t_\Psi))$ time by using $O((n/d)\log(n/d)) + |B|$ bits of extra space, where $d > 0$ is the sample rate and $|B|$ is the size of the bit vector.*

In theoretical size bounds, the sample rate is often assumed to be $\log^{1+\varepsilon} n$ for some $\varepsilon > 0$. Then, assuming that we use the bit vector of Raman et al. [81] for marking the sampled positions, the extra space becomes $o(n)$ bits, while $t_R = t_S = O(1)$.

**Optimizations.** If the index supports both $LF$ and $\Psi$ efficiently (as the FMI family does), we can effectively double the number of samples by using an extra binary string $B_n$ of length $n$. For each suffix array position $i$, we write $B_n[i] = 1$, if the nearest sample can be found faster by using $\Psi$ than by using $LF$. This reduces the expected distance to the nearest sample from $d/2$ to $d/4$. If $d \approx \log n$, this is already more space-efficient than doubling the number of samples.

A typical use of *locate* is to call it for all positions in the suffix array range returned by *find*. If the text is repetitive, it can be more efficient to compute *locate* for all positions in the range simultaneously, instead of doing it one position at a time. If suffix array positions $i$ and $i+1$ are located in the same equal letter run in BWT (self-repetition in the suffix array), then $LF(i+1) = LF(i) + 1$ ($\Psi(i+1) = \Psi(i) + 1$), and the corresponding bit vector operations are usually faster than for arbitrary positions. In repetitive texts, positions $LF(i)$ and $LF(i+1)$ ($\Psi(i)$ and $\Psi(i+1)$) are also in the same run (self-repetition) with high probability, and a significant amount of work can be saved by doing *locate* simultaneously for the entire range. See Section 4.4 for experimental results.

The standard sampling strategy, as described above, is geared towards worst case performance. As the nearest sample is never more than $d-1$ steps away, the desired position can be found in $O(d \cdot t_{LF})$ or $O(d \cdot t_{\Psi})$ time. On the other hand, if the query distribution is very skewed, another sampling strategy might provide better expected case performance. Experiments suggest that *locate* can be up to 10 times faster, if the samples have been optimally selected for a known distribution [23].[3] Adapting the samples for an unknown query distribution remains an open problem.

## 3.5 Dynamic versions

Consider the differences between the Burrows-Wheeler transforms of texts $T$ and $cT$, for an arbitrary text $T$ and an arbitrary character $c$. In both cases, the position of the end marker, $\mathrm{rank}(T,T)$ or $\mathrm{rank}(cT,cT)$, is determined by the longest suffix. Additionally, $\mathsf{BWT}_{cT}[\mathrm{rank}(cT,T)] = c$, while $\mathsf{BWT}_{T}[\mathrm{rank}(T,T)] = \$$. This suggests of a simple procedure for obtaining $\mathsf{BWT}_{cT}$ from $\mathsf{BWT}_{T}$ (see Figure 3.2). The first step is essentially a step of backward searching. The lexicographic range corresponding to pattern $T$ in the suffix array of text $T$ is $[\mathrm{rank}(T,T), \mathrm{rank}(T,T)]$, and if we continue the search with character $c$, we get $[\mathrm{rank}(cT,cT), \mathrm{rank}(cT,cT) - 1]$ (note that $\mathrm{rank}(T,cT) = \mathrm{rank}(cT,cT)$).

[3] The full version of [23] reports even greater speedups.

```
function update($\mathrm{rank}(T,T), c$)
    $\mathrm{rank}(cT,cT) \leftarrow C[c] + \mathrm{rank}_c(\mathsf{BWT}, \mathrm{rank}(T,T)) + 1$
    $\mathsf{BWT}[\mathrm{rank}(T,T)] \leftarrow c$
    $\mathsf{BWT} \leftarrow \mathsf{BWT}[1, \mathrm{rank}(cT,cT) - 1]\,\$\,\mathsf{BWT}[\mathrm{rank}(cT,cT), n]$
    for $i \leftarrow c + 1$ to $\sigma + 1$
        $C[i] \leftarrow C[i] + 1$
    return $(\mathsf{BWT}, C, \mathrm{rank}(cT,cT))$
```

Figure 3.2: Updating the Burrows-Wheeler transform of text $T$ to that of text $cT$ [38].

Chan et al. [7] used this idea to obtain *dynamic compressed suffix arrays* for a collection of texts, supporting insertion and deletion of individual texts. Their solution uses a *balanced search tree* to store the small blocks of a succinct bit vector, with each block containing $\log n$ to $2 \log n$ bits of the underlying sequence. Each node also contains the number of bits and 1-bits stored in the corresponding subtree. The bit vector takes $O(n)$ bits of space, and supports *rank* and *select* in $t_R = t_S = O(\log n)$ time. Inserting, deleting, and updating a bit takes $t_U = O(\log n)$ time.

If we want to insert a new end marker $\$_{r+1}$ into the BWT of collection $\mathcal{C}$ of $r$ texts, its lexicographic rank is $\mathrm{rank}(\mathcal{C} \cup \{\$_{r+1}\}, \$_{r+1}) = r + 1$. Any text $T$ in the collection can be extended to $cT$ by using the update procedure in Figure 3.2, if we know the lexicographic rank of $T$. We can also remove the first character of any text and finally the end marker by reversing the same procedure.

**Theorem 3.4.** *Let* rank *and* select *on dynamic binary sequences take $t_R$ and $t_S$ time, respectively, and let updating the sequence take $t_U$ time. Then a dynamic self-index of the CSA family supports the insertion and deletion of text $T$ in $O(|T| \cdot (t_R + \sigma \cdot t_U))$ time and $O(|T| \cdot (t_R + t_S + \sigma \cdot t_U))$ time, respectively. For an index of the FMI family, insertion and deletion times are $O(|T| \cdot (t_R + t_U) \log \sigma)$ and $O(|T| \cdot (t_R + (t_S + t_U) \log \sigma))$, respectively. The performance of these indexes follows from Theorems 3.1 and 3.2.*

In the CSA family, *rank* and *select* on BWT are solved with just one bit vector operation, while updating the BWT requires updating all $\sigma$ bit vectors. In the FMI family, *rank*, *select*, and updating all require $\log \sigma$ bit vector operations.

Mäkinen and Navarro [68] used a dynamic gap encoded bit vector to build an index of the FMI family. Their bit vector requires $nH_0 + o(n)$ bits of space, making the total size of the index $nH_k + o(n \log \sigma)$ bits for

$k \leq \alpha \log_\sigma n - 1$ and any constant $0 < \alpha < 1$. The performance of their index follows from Theorem 3.4 with $t_R = t_S = t_U = O(\log n)$.

Lee and Park [58] used a wavelet tree with internal alphabet of size $\log n$ to produce a more efficient dynamic index. Their index achieves AFFM-like query times and requires $O(|T| \cdot (1 + \log \sigma / \log \log n))$ time for inserting or deleting text $T$. Initial variants were either succinct or run-length encoded, but later developments by González and Navarro [33] and Lee and Park [57] obtained the same size bound as Mäkinen and Navarro [68].

So far, the dynamic indexes have been mostly theoretical in nature. Gerlach [29] has implemented a dynamic FM-index using succinct bit vectors in a Huffman-shaped wavelet tree. In practice, the index takes $2n$ to $3n$ bytes for a text of length $n$. Experiments suggest that dynamic updates are faster than rebuilding the index, when the size of insertion or deletion is at most $n/20$.

Salson et al. [86, 87] investigated a different model, where updates are allowed not only at the beginning of a sequence, but anywhere in the collection. They were unable to obtain meaningful time bounds, as the time complexity of a particular edit operation depends on the number of previous suffixes whose lexicographic rank is changed. See Section 4.2 for an expected case analysis of a closely related problem.

# Chapter 4

# Run-length compressed suffix array

In the terminology of Chapter 3, the *run-length compressed suffix array (RLCSA)* is a compressed self-index of the CSA family using data-aware run-length encoded bit vectors. As a theoretical structure, RLCSA has been described and analyzed in Paper I. Many of the implementation choices have been described in Paper II.

RLCSA has been designed for indexing highly repetitive sequences or collections of sequences. A sequence can be considered highly repetitive, if the number of equal letter runs in the Burrows-Wheeler transform is much less than the length of the sequence. Run-length encoding achieves good compression in such cases, if the bit vectors and other structures are data-aware, with no significant dependencies on the length of the sequence in their size bounds. Other design principles include simplicity and practical efficiency, even when a more complicated design might provide better theoretical time or space bounds.

In some sense, RLCSA is a a data-aware variant of Sad-CSA. Paper I also describes data-aware versions of RLFM and SSA, designed and implemented by Niko Välimäki. The *improved run-length FM-index (RLFM+)* uses the bit vector of Gupta et al. [37] to encode the lengths of equal letter runs, but is otherwise identical to the original RLFM. The *run-length encoded wavelet tree (RLWT)* is based on a balanced wavelet tree, using two gap encoded bit vectors of Gupta et al. [37] per binary sequence to achieve run-length encoding. While all three indexes have similar space bounds, RLWT is slightly smaller than RLCSA in practice, while RLFM+ is clearly larger. On the other hand, RLCSA is the fastest of the three indexes, followed by RLFM+ and RLWT.

## 4.1 Analysis

RLCSA uses data-aware run-length encoded and gap encoded bit vectors. Technical details of these bit vectors are described in Section 4.3. For the theoretical analysis, it is enough to note that both bit vectors support *rank* and *select* in $O(\log n + b)$ time, where $b$ is the size of small blocks in bits. On a binary sequence $B$ of length $n$, with $n_1$ 1-bits in $R_1$ runs, the size bounds are

$$\left(\mathrm{gap}(B) + O\left(n_1 \log\log \frac{n}{n_1}\right)\right)\left(1 + \frac{O(\log n)}{b}\right)$$

bits for the gap encoded variant and

$$\left(\mathrm{run}(B) + O\left(R_1 \log\log \frac{n}{R_1}\right)\right)\left(1 + \frac{O(\log n)}{b}\right)$$

bits for the run-length encoded variant.

Assume that there are $R$ equal letter runs in the Burrows-Wheeler transform. Then the total number of runs of 1-bits in bit vectors $B_c$ is also $R$. As the total number of 1-bits is $n$ and the total number of 0-bits is $(\sigma - 1)n$, we get a size bound of

$$R\left(\log \frac{\sigma n}{R} + \log \frac{n}{R} + O\left(\log\log \frac{\sigma n}{R}\right)\right)\left(1 + \frac{O(\log n)}{b}\right)$$

bits for BWT by using the bound gap $\leq n_1 \log(n/n_1) + n_1$. This size bound has an extra $R \log(n/R)$ term, when compared to direct run-length encoding of the BWT (see Section 2.3), as we are encoding the runs of each character separately.

We use the sampling scheme described in Section 3.4 to support *locate* and *extract*. A gap encoded bit vector is used as $B_s$ to mark the sampled positions. With sample rate $d$, the total size of the samples is $O((n/d)\log(n/d))$ bits. Gap encoded bit vectors are also used for encoding array $C$ and marking sequence boundaries. Together with the overhead from alphabet size and the number of sequences, these bit vectors take $O((r + \sigma)\log n)$ bits, where $r$ is the number of sequences in the collection. From Theorem 3.1, we get the following theorem.

**Theorem 4.1.** *Assume we have a collection of $r$ texts with total length $N$. The run-length compressed suffix array for the collection takes*

$$R\left(\log \frac{\sigma N}{R} + \log \frac{N}{R} + O\left(\log\log \frac{\sigma N}{R}\right)\right)\left(1 + \frac{O(\log N)}{b}\right) + O\left(\frac{N}{d} \log \frac{N}{d} + (r + \sigma)\log N\right)$$

*bits of space, where $b$ is the block size in bits, $R$ is the number of equal letter runs in the Burrows-Wheeler transform of the collection, and $d$ is the suffix array sample rate. The index supports* find$(P)$ *in* $O(|P| \cdot (\log N + b))$ *time,* locate *in* $O(d \cdot (\log N + b))$ *time, and* extract$(i, j)$ *in* $O((d+j-i)(\log N + b))$ *time.*

In practice, block size $b$ is chosen to be $\Theta(\log N)$ bits, so that the factor $1+O(\log N)/b$ in the size bound becomes $1+\varepsilon$, for some constant $0 < \varepsilon < 1$. This also simplifies the factor $\log N + b$ in the time bounds to $O(\log N)$.

## 4.2 Runs in BWT

In Section 2.2, we defined that each sequence of a collection has a distinct end marker, making the set of suffixes of the collection well-ordered. A drawback of this definition is that it increases the alphabet size significantly for large collections. In the following analysis, as well as in practice, we avoid the drawback replacing all end markers with a single symbol \$ after sorting. By doing this, we lose the ability to compute $\Psi$ for suffix array positions corresponding to the end markers, as well as the ability to compute $LF$ for positions corresponding to the first characters of a sequence.

**Entropy-based estimates.** As noted in Section 2.3, the number of equal letter runs in the BWT of a text of length $n$ is $R(T) \leq nH_k(T') + w_k(T')$, where $T'$ is the reverse of text $T$ and $w_k(T')$ is the number of order-$k$ contexts occurring in the reverse text. This bound holds simultaneously for all $k \geq 0$.

If we assume that the text has been generated by an order-0 source, we can use the character frequencies to approximate the expected number of runs. The character at position $i$ in the Burrows-Wheeler transform is a run head, if $i = 1$ or if the suffixes with lexicographic ranks $i-1$ and $i$ are preceded by different characters. Assuming that the character distribution is $\mathbf{p} = [n_1/n, \cdots, n_\sigma/n]$, the probability that the preceding characters differ is $1-\mathbf{p}\cdot\mathbf{p}$. Hence we can estimate that for an order-0 source, $R \approx n(1-\mathbf{p}\cdot\mathbf{p})$. Similar estimates can be made for higher order sources as well.

**Highly repetitive collections.** For $k = \lceil \log_\sigma n \rceil$, the number of possible order-$k$ contexts is $\sigma^k \geq n$. If a significant fraction of these contexts occur in the text, then the model will probably take more space than a statistical encoding of the text based on that model. Hence it is not reasonable to use an order-$k'$ model, for $k' > k$, to describe the text, unless only a small fraction of the possible contexts occur in the text, making it highly repetitive. On the other hand, if the number of runs is significantly less than predicted by an order-$k$ model, lexicographically adjacent suffixes will likely be preceded by the same character, making the text highly repetitive.

**Definition 4.1.** A collection of texts with total length $N$ is *highly repetitive*, if the number of equal letter runs in the Burrows-Wheeler transform of the collection is significantly less than predicted by an order-$k$ model, for $k = \lceil \log_\sigma N \rceil$.

In the following, we estimate the number of equal letter runs in the Burrows-Wheeler transform of a highly repetitive collection, created by applying $s$ random edit operations on a collection of $r$ identical random texts of length $n$.

### Combinatorial properties.

**Lemma 4.1.** *Let $\mathcal{C} = \{T_1, \dots, T_r\}$ be a collection of $r$ copies of text $T$. Then $R(\mathcal{C}) = R(T)$.*

*Proof.* For all positions $j$, the suffixes $T_i[j, n]$ and $T_{i'}[j, n]$ are identical for any texts $T_i$ and $T_{i'}$, and the lexicographic order among them is determined by sequence numbers. Hence $\text{rank}(\mathcal{C}, T_i[j, n]) = r \cdot (\text{rank}(T, T[j, n]) - 1) + i$. As the suffixes $T_i[j-1, n]$ are also identical for all texts $T_i$, we have

$$\mathsf{BWT}_{\mathcal{C}}[\text{rank}(\mathcal{C}, T_1[j, n]), \text{rank}(\mathcal{C}, T_r[j, n])] = \mathsf{BWT}_T[\text{rank}(T, T[j, n])]^r.$$

Hence the number of equal letter runs does not depend on the number of copies of text $T$. □

**Definition 4.2.** Let $\mathcal{C} = \{T_1, \dots, T_r\}$ be a collection of texts, all derived by substitutions, insertions, and deletions from some base text $T$, and let $\mathcal{A}$ be a multiple alignment representing those edit operations. The *significant prefix $S_{i,j}$* of suffix $T_i[j, |T_i|]$ is the shortest prefix of $T_i[j, |T_i|]$ that differentiates it from all other suffixes, except possibly from those with first characters aligned with character $T_i[j]$.

The intuition behind the definition is that significant prefixes can be used to sort non-aligned suffixes into lexicographic order. Aligned suffixes are assumed to be equivalent in respect to the number of runs, unless there has been an edit operation nearby.

**Lemma 4.2.** *Let $\mathcal{C} = \{T_1, \dots, T_r\}$ be a collection of $r$ copies of text $T$, and let $\mathcal{C}'$ be the same collection after substituting one character of one text with another character. Then $R(\mathcal{C}') \leq R(T) + O(k)$, where $k$ is the number of suffixes of collection $\mathcal{C}'$ whose significant prefixes cover the substitution.*

*Proof.* Assume that the substituted character was $T_i[j]$. We get the new Burrows-Wheeler transform $\mathsf{BWT}_{\mathcal{C}'}$ by removing the characters corresponding to suffixes $T_i[1,n], \dots, T_i[j+1,n]$ from $\mathsf{BWT}_{\mathcal{C}}$ and inserting them into their new positions, as the rest of the suffixes remain the same.

As $k$ significant prefixes cover the substituted character, suffixes $T_i[1,n]$ to $T_i[j-k,n]$ are not affected by the substitution. The lexicographic order between them and the remaining non-aligned suffixes does not change. The lexicographic order between a removed suffix and aligned suffixes can change, but as the preceding character does not change, this means just inserting the same character back into a different part of the same run. Hence we only need to consider suffixes $T_i[j-k+1,n]$ to $T_i[j+1,n]$.

For suffixes $T_i[j-k+1,n]$ to $T_i[j,n]$, we are inserting the same character into a new position in the BWT. This can break an existing run into two parts, and the inserted character can create another run on its own. Hence these suffixes can create at most $2k$ new runs. Finally we insert the substituted character to the original position of suffix $T_i[j+1,n]$. This can also break an existing run and create a new run, so the total number of new runs is at most $2k+2$. The result follows by noting that according to Lemma 4.1, $R(\mathcal{C}) = R(T)$. □

The lemma immediately generalizes to multiple substitutions.

**Corollary 4.1.** *Let $\mathcal{C} = \{T_1, \dots, T_r\}$ be a collection of $r$ copies of text $T$, and let $\mathcal{C}'$ be the same collection after substituting a total of $s$ characters with other characters. Then $R(\mathcal{C}') \leq R(T) + O(k)$, where $k$ is the number of suffixes of collection $\mathcal{C}'$ whose significant prefixes cover at least one substitution.*

*Proof.* The proof follows that of Lemma 4.2. For each substitution $T_i[j]$, we remove the characters corresponding to suffixes whose significant prefixes contain the substitution, and insert them into their new positions. □

**Expected case properties.** In the following, we assume that all random choices are independent and identically distributed.

**Lemma 4.3.** *Let $T$ be a random text. The expected length of the longest repeated substring is $O(\log_\sigma n)$.*

*Proof.* Let $X_{i,j}^k$, where $i < j \leq n-k+1$, be a variable indicating whether substrings $T[i, i+k-1]$ and $T[j, j+k-1]$ are identical. If $i+k \leq j$, the expected value of $X_{i,j}^k$ is $\sigma^{-k}$.

Consider now the case $j < i+k$, where the substrings overlap at $k-(j-i)$ positions. To get identical substrings, we can choose $T[i, j-1]$ arbitrarily, but character $T[l]$ must be identical to $T[l-(j-i)]$ for all $j \leq l < j+k$. Hence the expected value of $X_{i,j}^k$ is also $\sigma^{-k}$ in this case.

The expected number of repeats of length $k$ is then

$$E\left[\sum_{1 \leq i < j \leq n-k+1} X_{i,j}^k\right] \leq \frac{n^2}{2\sigma^k}.$$

By Markov's inequality, the probability of having at least one repeat of length $2\log_\sigma n + k'$ is at most $1/(2\sigma^{k'})$. As the probability decreases exponentially by $k'$, the expected length of the longest repeat is $O(\log_\sigma n)$. □

We are now ready to prove the main result.

**Theorem 4.2.** *Let $\mathcal{C} = \{T_1, \ldots, T_r\}$ be a collection of $r$ copies of random text $T[1,n]$ over alphabet of size $\sigma$, and let $\mathcal{C}'$ be the same collection after randomly substituting a total of $s$ characters with other characters. Then the expected number of runs in the Burrows-Wheeler transform of collection $\mathcal{C}'$ is at most $R(T) + O(s \log_\sigma(rn))$.*

*Proof.* Consider texts $T_i$ and $T_{i'}$ of collection $\mathcal{C}'$, and some arbitrary positions $j < j'$ in them. If the substring of length $k$ starting at $T_i[j]$ is identical to the one starting at $T_{i'}[j']$, then the corresponding suffixes must have significant prefixes of length at least $k+1$. By following the proof of Lemma 4.3, we find that the expected length of the longest such repeat is $O(\log_\sigma(rn))$, so significant prefixes of length $O(\log_\sigma(rn))$ are enough in the expected case. The result follows from Corollary 4.1. □

**Extensions.** While above results consider only substitutions, they are easy to extend to handle random insertions and arbitrary deletions. If an insertion happens after character $T_i[j]$, and suffix $T_i[j-k+1,n]$ is the longest one whose significant prefix covers at least one of the inserted characters, then we have to remove suffixes $T_i[j-k+1,n]$ to $T_i[j+1,n]$, and insert them to their new positions, along with the new suffixes. Similarly, if we delete substring $T_i[j,j']$, we remove suffixes $T_i[j-k+1,n]$ to $T_i[j'+1,n]$, and insert the ones that were not deleted to their new positions.

Large insertions consisting of a substring already in the collection are also easy to handle. A *recombination*, replacing texts $T_1 = T_{1,a}T_{1,b}$ and $T_2 = T_{2,a}T_{2,b}$ with texts $T_{1,a}T_{2,b}$ and $T_{2,a}T_{1,b}$, is essentially two edit operations. Extending a text by a LZ77 phrase consisting of a substring already occurring in the collection, followed by a single character, is also worth two edit operations. In general, we can estimate the effect of a complex edit operation on the number of runs by counting the *points of discontinuity* the operation introduces into the collection, and treating each of them as a simple edit operation.

## 4.3 Implementation

The implementation of RLCSA has been written in C++. All queries have been implemented as `const` functions that do not alter the state of the index, allowing multiple threads to to use the index in parallel. Index construction (see Chapter 5) has also been parallelized by using *OpenMP* and either *MCSTL*[1] or the version of MCSTL integrated into GCC, the *libstdc++ parallel mode.*

Both gap encoded and run-length encoded bit vectors support a number of derived queries in addition to the plain *rank* and *select*. For example, valueAfter($i$) returns $(i', \text{rank}_1(B, i'))$ for the smallest $i' \geq i$ such that $B[i'] = 1$. Iterator version of *select* allows calling *select* for successive 1-bits faster than by using the plain query. Another variant selectRun($i$) returns the length of the run of 1-bits starting from the one of rank $i$, in addition to answering *select*.

Small blocks consist of $b$ bits of compressed data. In gap encoded bit vectors, the compressed data consists of $\delta$-encoded distances between successive 1-bits. If a code does not fit into the current block, it is stored in the next block, and the rest of the block remains empty. In run-length encoded bit vectors, each run is encoded as the distance from the previous run, followed by the number of 1-bits in the run. Both of the integers are encoded using $\delta$-codes, as they are expected to be fairly large in highly repetitive sequences (see Sections 3.2 and 4.2). The entire encoding of a run must fit into the block, or else it is stored in the next block. However, a maximal run of 1-bits can be split into two shorter runs, so that the first of these runs can be stored in the last remaining bits of a block.

For each small block, the first 1-bit $B[i]$ not included in the previous blocks is sampled and stored as a pair $(i, \text{rank}_1(B, i))$ taking $2\lceil \log(n + 1) \rceil$ bits, where $n$ is the length of the binary sequence. This sample is not included in the corresponding block. Separate large blocks are used for *rank* and *select*. In both cases, the parameter space ($[1, n]$ for *rank* and $[1, n_1]$ for *select*) is divided into $n_b/5$ blocks of equal size, where $n_b$ is the number of small blocks. For each large block containing the range $[a, a']$ of parameter space, we store a pointer to the small block that is used to answer $\text{rank}_1(B, a)$ or $\text{select}_1(B, a)$. Each of these pointers takes $\lceil \log(n_b + 1) \rceil$ bits.

[1] `http://algo2.iti.kit.edu/singler/mcstl/`

To answer $\text{rank}_1(B, i)$ or $\text{select}_1(B, i')$, we start by dividing the parameter by the length of a large block. This gives us $j$, the number of the large block that contains the answer to the query. Pointers for large blocks $j$ and $j + 1$ give us the range of small blocks that contains the answer, and the correct block is determined by binary searching the sampled 1-bits corresponding to these small blocks. Finally, the small block is decompressed to determine the answer. Finding the correct small block takes $O(\log n)$ time, and decompressing the block takes $O(b)$ time. The number of random memory accesses is 3, unless the large block contains many small blocks.

The implementation uses linear search instead of binary search to find the small block. This does not guarantee performance in the worst case, but is faster in practice. If the binary sequence is very skewed, some large blocks can contain many small blocks, reducing query performance for those blocks. While this behavior has been seen in practice, the effect seems to be limited on real data [23].

RLCSA uses the standard sampling mechanism (see Section 3.4) to support *locate* and *extract*. In addition to locating a single position at a time, the implementation also contains a variant of *locate* that locates all positions in a suffix array range simultaneously. This version is often several times faster than the standard one, taking advantage of both memory locality and the run-length encoding in the bit vectors. In addition to selecting samples at regular intervals, it is also possible to select the samples optimally or greedily according to a known query distribution [23].

Default small block sizes are 32 bytes for the run-length encoded bit vectors and 16 bytes for the gap encoded ones. Suffix array sample rate $d$ defaults to 128, with samples taking less than $n$ bits for a sequence of length $n$. Instead of using a bit vector, array $C$ has been implemented with an ad hoc mechanism that is somewhat similar to the high level structure in bit vectors, but faster in practice. Alphabet has been fixed to $0, \ldots, 255$, where 0 plays the role of end marker $ in some construction options, while being a regular character in other alternatives.

Support for multiple sequences is based on having different lexicographic values for the end markers of different sequences, as described in Section 2.2. The sequences are implicitly concatenated to form a single sequence, and a bit vector is used to map positions in this sequence to actual sequences. Some padding is added to between the sequences, so that a new sequence always starts at a position that is a multiple of sample rate $d$. This way, the regular sampling mechanism always samples the first position of every sequence. The end marker of each sequence is implicitly sampled, as its lexicographic rank corresponds to the sequence number.

| | | RLCSA | | Sad-CSA | |
|---|---|---|---|---|---|
| **Data set** | **Size** | **Size** | **Find** | **Size** | **Find** |
| para | 409 MB | 41.34 MB | 1.12 MB/s | 69.92 MB | 0.34 MB/s |
| fiwiki | 400 MB | 5.63 MB | 1.36 MB/s | 53.05 MB | 0.38 MB/s |

Table 4.1: RLCSA and Sad-CSA for two highly repetitive collections. Index size without suffix array samples and *find* performance on 50000 patterns of length 20.

## 4.4 Experiments

In this section, we compare the size and query performance of RLCSA to Sad-CSA (see Section 3.2), the most similar earlier self-index. We chose two highly repetitive data sets: the genomes of 36 strains of *Saccharomyces paradoxus* (para) from Paper I, taking a total of 409 megabytes, and a 400-megabyte prefix of the Finnish language Wikipedia archive with full version history from Paper II. The experiments were performed on a system with two quad-core 2.53 GHz Intel Xeon E5540 processors running Ubuntu 10.04 with Linux kernel 2.6.32. Only one core was used in the experiments. The programs were compiled with g++ version 4.4.3. For a more thorough experimental comparison, see Paper I.

We measured *find* and *locate* (including *find*) performance of the indexes with 50000 randomly selected patterns of length 20. The indexes used default parameter values and suffix array sample rates $d = 32, 64, 128, 256$. In RLCSA, we used both the standard *locate* mechanism (RLCSA) and the one optimized for locating ranges of suffix array values (RLCSA-OPT).

Index sizes and *find* results can be seen in Table 4.1. Of the two data sets, fiwiki turned out to be much more repetitive. RLCSA was able to compress it to just 1.6% of the original size, while the non-data-aware Sad-CSA required more than 13% of the original size. On para, the differences were smaller, with RLCSA taking 10% and Sad-CSA 17% of the original size. The *find* performance of RLCSA was much better than that of Sad-CSA, as the compressed representation of $\Psi$ in Sad-CSA does not support *rank* directly, but uses binary search on *select* instead.

The results for *locate* can be seen in Figure 4.1. RLCSA with optimized *locate* is faster than Sad-CSA with the same sample rate, and also much smaller. With regular *locate*, RLCSA loses to Sad-CSA, especially on the more repetitive fiwiki data set. As both indexes are much faster on fiwiki than on para, the implementation of Sad-CSA seems to have some optimizations for locating suffix array ranges simultaneously as well.

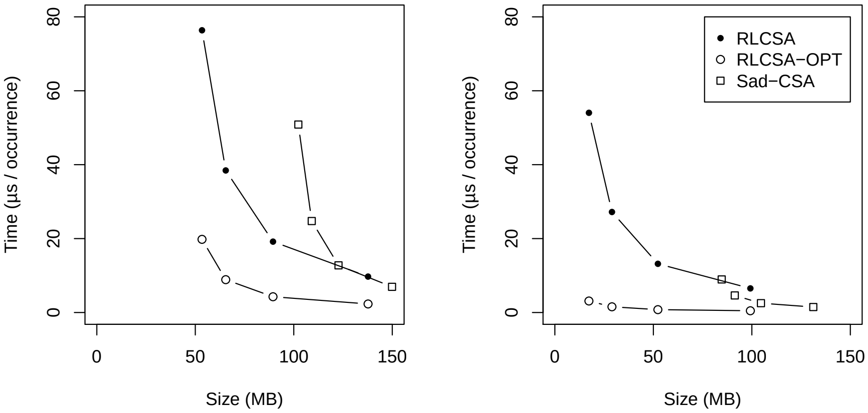


Figure 4.1: *Locate* performance of RLCSA and Sad-CSA on para (left) and fiwiki (right). Index sizes and average *locate* times with sample rates $d = 32, 64, 128, 256$ on 50000 patterns of length 20.

The optimized *locate* is 13–18 times faster than the standard one on fiwiki, and about 4 times faster on para. This shows that even though RLCSA is unable to compress the samples, unlike some other proposals [69, 40], it can compensate this by using a small number of samples more efficiently, when the collection is highly repetitive.

In general, the optimizations depend on both the number and the length of the repetitions. Assume that we want to locate a pattern that occurs at the beginning of a repeated substring of length $l$, with $k$ copies in the collection. Then $k$ is an upper bound for the speedup from the optimizations, and the actual speedup depends on whether the significant prefixes of the sampled suffixes used in the *locate* are within the repeated substring. If this is the case, then the suffixes remain lexicographically adjacent until the samples have been found. Otherwise the suffixes diverge at some point, limiting the speedup from locating all occurrences simultaneously.

It should be noted that Sad-CSA has been optimized for *locate*. The small blocks in the encoding of $\Psi$ contain a fixed number of values (1-bits) each, and hence the correct block can be derived directly from the parameter value, avoiding one level of indirection. The sampling mechanism is also non-standard, storing one out of $d$ suffix array values instead of one out of $d$ text positions. This does not guarantee worst-case time bounds, but makes it much faster to determine whether the current suffix array value has been sampled.

## 4.5 Later indexes

RLCSA was the first compressed index that was intended for indexing highly repetitive collections. Later papers on similar topics have used it as a reference point, against which the authors compare their results.

The self-index of Huang et al. [40] is based on a multiple alignment of similar sequences. The authors start with similar analysis as in Section 4.2, and explicitly encode the differences between the sequences, while building a BWT-based index for the common segments in the sequences. When the number of differences is small, the index is smaller than RLCSA. The main reason for this is that the common segments are stored just once, while RLCSA uses $O(\log r)$ extra bits per run to encode the $r$ copies. Because of the complexity of the index, *find* is slower than in regular BWT-based indexes. On the other hand, *locate* can be much faster, if the pattern occurs mostly in the common segments.

The work of Claude et al. [11, 9] develops self-indexes based on *straight-line programs (SLP)* in general, and *Re-Pair* [55] in particular. Similar to LZ-indexes, these indexes offer fast *locate*, but do not support *find*. A straight-line program is essentially a *context-free grammar* in the *Chomsky normal form* for a language containing a single sequence. In the grammar, each variable $X$ produces either two new variables $YZ$ or a character $c \in \Sigma$. One variant compresses the text with Re-Pair, allowing fast random access to it, and uses a *q-gram index*, where the occurrence lists are differentially encoded and compressed with LZ77. Another variant uses the Re-Pair encoding of the text directly as a self-index. Both variants offer better compression than RLCSA, if the text is very highly repetitive [9].

As already conjectured in the original paper describing RLCSA, a self-index based on LZ77 parsing offers better compression for highly repetitive sequences than RLCSA. The LZ77-index of Kreft and Navarro [46] proves this conjecture. Like the other LZ-indexes, the index is much larger than plain LZ77-compressed text. In practice, the LZ77-index is smaller and offers faster *locate* than RLCSA and SLP-based indexes on highly repetitive texts. The reason for better compression than SLP-based indexes is that while both approaches replace a repetition with a reference, SLPs have more overhead in encoding the original copy.

There have also been recent papers on compressing highly repetitive collections of sequences, while supporting the decompression of individual sequences or arbitrary substrings efficiently [48, 21, 45]. In these tasks, RLCSA is at disadvantage, as LZ77 compresses highly repetitive texts better than BWT, and the bit vector-based encoding of BWT also adds some overhead, when compared to direct compression.

# Chapter 5

# Space-efficient CSA construction

In this chapter, we investigate the direct construction of compressed suffix arrays, extending the results in Paper II. We are mostly interested in algorithms whose space usage is dominated by the CSA itself, making it possible to build indexes for texts that are larger than memory size.

The standard way of constructing compressed suffix arrays is to build a suffix array first, and then compress it. There are many existing algorithms [79], with the best of them working in $O(n)$ time and requiring $2n$ bits of working space in addition to the text and the suffix array [74]. Yet as a compressed suffix array usually requires less than $n$ bytes of memory [18], and can take less than $n$ *bits* with highly repetitive texts (see Section 4.4), this approach limits the use of CSAs to much smaller texts than could be handled in the available memory.

Many of the suffix array construction algorithms can be adapted to construct the Burrows-Wheeler transform directly. This often replaces the $4n$ to $8n$-byte suffix array with a $n$-byte BWT, reducing memory usage considerably. Yet even the best algorithms [49, 78] require the text or the BWT — or both — in memory, limiting the size of the texts that can be indexed. External memory algorithms for constructing the suffix array [13] and the Burrows-Wheeler transform [15] exist, but they tend to be slow in practice. In principle, dynamic self-indexes (see Section 3.5) could be used for space-efficient CSA construction, but the implementations seen so far are both slower and require more memory than the alternatives.

Direct algorithms for compressed suffix array construction [38, 70, 39] are the best solution so far. We are especially interested in the algorithm of Hon et al. [38] that works in $O(n \log n)$ time and requires $|\mathsf{CSA}| + O(n)$ bits of memory. The algorithm essentially uses a static index to simulate CSA construction by a dynamic index, inserting many suffixes in a single update. We describe two practical variants of this algorithm: one that is more space-efficient and another one that can be faster than the original. As the basic building block, we describe an algorithm for merging compressed suffix arrays.

A similar idea has been recently used for constructing the Burrows-Wheeler transform for a large collection of short texts in external memory [3]. Instead of inserting large blocks of text at once, the algorithm extends each text by a single character in each step. This way, the algorithm remains simple and has to maintain only a small amount of state information, making it fast and space-efficient in practice. On the other hand, if the texts are longer than a few hundred characters, the algorithm requires too many passes over the data to be useful.

## 5.1 Merging Burrows-Wheeler transforms

**Algorithm of Hon et al.** The construction algorithm of Hon et al. [38] can be interpreted as updating the Burrows-Wheeler transform of a text. Assume that we have already constructed $\mathsf{BWT}$ for some text $T$, and we want to update it for $ST$, where $S$ is a sequence of length $l$. We call the suffixes of $ST$ starting in $S$ the *long suffixes* of $ST$, and the rest of the suffixes *short suffixes.*

**Definition 5.1.** Let $T$ and $T'$ be two texts. The *rank array* $\mathsf{RA}[1, |T'|]$ of text $T'$ relative to text $T$ is an array such that $\mathsf{RA}[i] = \mathrm{rank}(T, T'[i, |T'|])$. The rank array of a set of suffixes of text $T'$ relative to text $T$ is the corresponding subsequence of array $\mathsf{RA}$.

The definition also generalizes for collections of texts.

Updating the BWT starts with computing the rank array of long suffixes of text $ST$ relative to text $T$. Backward searching can be used to compute the rank array in a similar way as in the update rule for dynamic compressed suffix arrays (see Section 3.5). A detailed algorithm can be found in Figure 5.1.

In addition to determining the lexicographic ranks of long suffixes among short suffixes, we must also determine their ranks among themselves. Conceptually this is done by sorting the long suffixes by their first $l$ characters, breaking ties by using the suffix array of $T$. With both ranks, we can determine the lexicographic ranks of long suffixes among all suffixes.

**Lemma 5.1** (Fact 1 in [38]). *The lexicographic rank of a long suffix $S'$ among all suffixes of $ST$ is the sum of its lexicographic ranks among long suffixes and among short suffixes.*

```
function computeRanks($\mathrm{rank}(T, T), S, l$)
    $pos \leftarrow \mathrm{rank}(T, T)$
    for $i \leftarrow l$ to 1
        $pos \leftarrow C[S[i]] + \mathrm{rank}_{S[i]}(\mathsf{BWT}, pos) + 1$
        $\mathsf{RA}[i] \leftarrow pos$
    return $\mathsf{RA}$
```

Figure 5.1: Computing the rank array of long suffixes of text $ST$ relative to text $T$ by backward searching the Burrows-Wheeler transform of $T$.

At this point, we have the lexicographic ranks among long suffixes in suffix array order, and the ranks among short suffixes in text order. By sorting the rank array in increasing order, we get both ranks in suffix array order. This follows from the fact that $S_1 < S_2$ implies $\mathrm{rank}(T, S_1) \leq \mathrm{rank}(T, S_2)$. Hence the lexicographic ranks of long suffixes among short suffixes must form a non-decreasing sequence, when put into suffix array order. The entire merging algorithm is as follows.

1. Determine the rank array $\mathsf{RA}$ by the algorithm in Figure 5.1. Sort it to get array $\mathsf{RA}'$, and update this array by rule $\mathsf{RA}'[i] \leftarrow \mathsf{RA}'[i] + i$ to get the lexicographic ranks of long suffixes among all suffixes.

2. Sort the long suffixes, and form $\mathsf{BWT}_S$ as the subsequence of $\mathsf{BWT}_{ST}$ corresponding to the long suffixes.

3. Update $\mathsf{BWT}_T[\mathrm{rank}(T, T)] \leftarrow S[l]$, and then merge $\mathsf{BWT}_S$ and $\mathsf{BWT}_T$ to get $\mathsf{BWT}_{ST}$. The merging is done by inserting characters from $\mathsf{BWT}_S$ to positions marked in $\mathsf{RA}'$, filling the rest of the positions with characters from $\mathsf{BWT}_T$.

**Lemma 5.2** (Lemma 10 in [38]). *The above algorithm updates* $\mathsf{BWT}_T$ *to* $\mathsf{BWT}_{ST}$ *in* $O(l \log n + n)$ *time, requiring* $4l \log n + n + o(n)$ *bits of space in addition to* $\mathsf{BWT}_S$ *and* $\mathsf{BWT}_T$.

**Merging compressed suffix arrays.** A simplified version of the above algorithm can be used to merge two compressed suffix arrays. Assume that we have compressed suffix arrays for two texts $T$ and $T'$, and we want to merge them to get a compressed suffix array for the collection $\mathcal{C} = \{T, T'\}$. We select $\mathsf{CSA}_T$ as the basic index, and update it to get $\mathsf{CSA}_{\mathcal{C}}$.

In step 1, we get the rank array by searching $\mathsf{CSA}_T$ for text $T'$. However, as we are inserting entire texts instead of extending existing texts, we start with $\mathsf{RA}[|T'|] = 1$, as the end marker of $T'$ will come immediately after the end marker of $T$ in lexicographic order. Step 2 is not needed, as we already have $\mathsf{CSA}_{T'}$. Step 3 works as above, except that we are merging compressed BWTs instead of plain BWTs. If we merge the bit vectors of an index of the CSA family one at a time, we are forced to scan the array $\mathsf{RA}'$ at least $\sigma$ times. This can be avoided by merging all bit vectors simultaneously, using a buffer of $\Theta(\sigma)$ characters to avoid polling each of the bit vectors for the next 1-bit too often.

**Lemma 5.3.** *The above algorithm merges compressed suffix arrays* $\mathsf{CSA}_T$ *and* $\mathsf{CSA}_{T'}$, *where* $|T'| \leq |T|$, *in* $O(|T'| \cdot (t_B + \log|T'|) + \min(|\mathsf{CSA}_{TT'}|, |T|))$ *time, where* $t_B$ *is the time required for one step of backward searching. Working space is* $|T'| \log|TT'| + O(\sigma \log n)$ *bits in addition to the CSAs and* $T'$.

The lemma applies for indexes of both CSA and FMI families, as long as individual bit vectors can be merged in time relative to their compressed size, and a sequential scan of a bit vector can be done in $O(n_1)$ time. It generalizes immediately to merging compressed suffix arrays of collections of sequences.

We can also output the rank array directly in suffix array order, avoiding the $O(|T'| \log|T'|)$ term in the time bound, if we backward search $\mathsf{CSA}_{T'}$ simultaneously. This allows us to store the array $\mathsf{RA}'$ as a bit vector of length $|TT'|$, which can be much less than the $|T'| \log|TT'|$ bits of a plain array, if the texts are of similar size. We do not even need the text $T'$, as it can be efficiently extracted from $\mathsf{CSA}_{T'}$ (in blocks of $d$ characters in the CSA family, as extraction proceeds in forward direction). In practice, none of these optimizations are very useful, as backward searching is much more expensive than integer sorting.

## 5.2 Construction algorithm

The merging algorithm can be used as the basic building block of a space-efficient CSA construction algorithm. Assume that we have a large collection of texts $\mathcal{C}$ with total length $N$. The algorithm is as follows, with $\mathsf{CSA}_i$ denoting the compressed suffix array of collection $\mathcal{C}_i$.

1. Split the collection into $m$ smaller collections $\mathcal{C}_1, \dots, \mathcal{C}_m$ of size $N/m$.

2. Build $\mathsf{CSA} = \mathsf{CSA}_1$, and use it as a basis for the final index.

3. For $i = 2, \dots, m$, build $\mathsf{CSA}_i$ and merge it with $\mathsf{CSA}$ to get the new $\mathsf{CSA}$.

Note that if there are $r$ sequences in the union of collections $\mathcal{C}_1, \dots, \mathcal{C}_i$, the rank array of collection $\mathcal{C}_{i+1}$ must have value $r$ for each end marker in the collection.

**Theorem 5.1.** *Let $\mathcal{C}$ be a collection of texts with total length $N$ that can be split into $m$ subcollections of size $N/m$. We can use the merging algorithm to construct a compressed suffix array for $\mathcal{C}$ in $O(N + m \cdot \min(|\mathsf{CSA}|, N))$ time using $|\mathsf{CSA}| + \max_i(|\mathsf{CSA}_i|) + O((N/m + \sigma) \log N)$ bits of space, where $\sigma$ is the size of the alphabet and $\mathsf{CSA}_i$ is a CSA for the $i$th subcollection.*

*Proof.* We assume that the CSA is based on bit vectors that support *rank* and *select* in $O(1)$ time. By using backward searching on $\mathsf{CSA}_i$ to output the rank array directly in suffix array order, we can do all merges within the given time and space bounds. The result follows by using any linear-time suffix array construction algorithm for building the partial indexes $\mathsf{CSA}_i$. □

The algorithm can be efficiently parallelized on a single machine. For building the partial indexes, we can either use a parallel suffix array construction algorithm or, if memory allows, index multiple subcollections in parallel. Backward searching can be parallelized, as it does not change the state of the index. Parallel sorting is a well-researched topic, so it does not matter, whether we construct the rank array in text order or in suffix array order. Finally, merging can be parallelized by either merging multiple bit vectors simultaneously, or by dividing a bit vector into multiple parts for merging. If the final CSA is small enough to fit into the memory of a single system, we can also use a computer cluster for construction.

## 5.3 Indexing a single sequence

With practical implementation choices, the algorithm in Section 5.2 uses roughly $(2N/m) \log N$ bits of working space to construct a compressed suffix array for a collection of total size $N$ in $m$ parts, while the algorithm of Hon et al. [38] uses roughly $(4N/m) \log N + N$ bits. As the algorithms are otherwise almost the same, this represents a significant improvement in space usage without any similar penalty in performance. On the other hand, the collection must be partitioned at sequence boundaries, while the algorithm of Hon et al. can use arbitrary partitioning. In this section, we show how this limitation can be lifted by using $(3N/m) \log N$ bits of working space, with the possibility of making the algorithm faster in practice.

As noted in Section 5.1, $S_1 < S_2$ implies $\mathrm{rank}(T, S_1) \leq \mathrm{rank}(T, S_2)$ for any text $T$ and any sequences $S_1$ and $S_2$. In particular, this means that $T'[i, |T'|] < T'[j, |T'|]$ implies $\mathsf{RA}[i] \leq \mathsf{RA}[j]$ in the merging algorithm. Additionally, $T'[i] < T'[j]$ implies $T'[i] + \mathsf{RA}[i] < T'[j] + \mathsf{RA}[j]$. This means that sequence $T' + \mathsf{RA}$ (where $(T' + \mathsf{RA})[i] = T'[i] + \mathsf{RA}[i]$) has the same suffix array as text $T'$.

**Lemma 5.4.** *Let $T$ and $T'$ be two texts. Then $T' + \mathsf{RA}$ has the same suffix array as text $T'$, where $\mathsf{RA}$ is the rank array of text $T'$ relative to text $T$.*

If text $T$ is much longer than text $T'$, most elements in the rank array are likely to be unique. Hence it should be faster to build a suffix array for the rank array than for the text itself. This gives us the following algorithm.

1. Split the collection into $m$ smaller collections $\mathcal{C}_1, \dots, \mathcal{C}_m$ of size $N/m$.
2. Build $\mathsf{CSA} = \mathsf{CSA}_1$, and use it as a basis for the final index.
3. For $i = 2, \dots, m$, determine the rank array $\mathsf{RA}$ of collection $\mathcal{C}_i$ relative to the union of all previous collections $\mathcal{C}'$. Build a suffix array of $\mathcal{C}_i + \mathsf{RA}$, use it to build $\mathsf{CSA}_i$, and merge $\mathsf{CSA}_i$ with $\mathsf{CSA}$ to get the new $\mathsf{CSA}$.

We may have to split a text $T = ST'$ into two collections, so that $T' \in \mathcal{C}_i$ and $S \in \mathcal{C}_{i+1}$. In this case, we append an end marker to string $S$, and put $x = \mathrm{rank}(\mathcal{C}', T')$ in the corresponding position in the rank array. As the end marker must be unique to get the correct suffix array, we increment the rank array by 1 for all other positions $i$, where $\mathsf{RA}[i] \geq x$. Furthermore, if collection $\mathcal{C}'$ contains $r$ texts, and there are $r'$ texts with regular end markers in collection $\mathcal{C}_{i+1}$, we reserve ranks $r, \dots r + r' - 1$ for the end markers to get the correct ordering between them, and increment all other values by $r' - 1$. Note that we have to remove the position corresponding to the end marker of string $S$ from the suffix array before constructing $\mathsf{CSA}_{i+1}$, as the end marker is not included in the original collection. Note that all changes to the rank array must be reversed, before it can be used in merging.

**Theorem 5.2.** *Let $\mathcal{C}$ be a collection of texts with total length $N$, and let $m > 0$ be an integer. We can use the merging algorithm to construct a compressed suffix array for $\mathcal{C}$ in $O(N + m \cdot \min(|\mathsf{CSA}|, N))$ time using $|\mathsf{CSA}| + \max_i(|\mathsf{CSA}_i|) + O((N/m + \sigma) \log N)$ bits of space, where $\sigma$ is the size of the alphabet and $\mathsf{CSA}_i$ is a CSA for the ith subcollection.*

The working space is now roughly $(3N/m) \log N$ bits, as suffix array construction algorithms generally require $(2N/m) \log N$ bits of writable space with large alphabets, and the rank array must be kept intact during construction.

## 5.4 Implementation

Two variants of the construction algorithm are included in the implementation of RLCSA (see Chapter 4). Written in C++, the principal goal of the implementation is to allow indexing text collections that are too large to fit into memory. The implementation has been parallelized by using *OpenMP* and either *MCSTL*[1] or the version of MCSTL integrated into GCC, the *libstdc++ parallel mode*. Both algorithms assume that the collection is stored on disk as a set of $m$ files, each of them less than 4 gigabytes in size.

First of the algorithms, *Merge*, is an implementation of the algorithm in Section 5.2. It first builds RLCSAs for all of the subcollections, using a parallelized prefix-doubling algorithm (see below) for the task, and stores them on disk. After all partial indexes have been built, the algorithm starts merging them. Merging has also been parallelized, assuming that the current subcollection consists of multiple texts, so that multiple threads can be used to construct the rank array. The merging phase requires roughly $9n$ bytes of memory in addition to the two RLCSAs, where $n = N/m$ is the size of a subcollection. This includes $8n$ bytes for the rank array and $n$ bytes for the collection. In practice, working space can be significantly more than that, as in-place merging of bit vectors has not been implemented.

The second algorithm, *Fast*, is the algorithm from Section 5.3, except that splitting a text into two subcollections has not been implemented. It processes the subcollections one at a time, using the rank array to build a RLCSA, before merging it with the existing index. This variant uses $25n$ to $29n$ bytes of working space, where the extra $16n$ to $20n$ bytes comes from the suffix array construction algorithm.

Both algorithms use a parallelized *prefix-doubling* algorithm to build the partial indexes. Prefix-doubling-based algorithms are useful for constructing suffix arrays for collections, as we can easily afford having different lexicographic values for the end markers of different sequences. A prefix-doubling algorithm maintains an invariant that array $\mathsf{SA}_k$ contains the suffixes in lexicographic order, and array $\mathsf{RA}_k$ contains the lexicographic ranks of the suffixes in text order, when the ordering is based on $k$-character prefixes of each suffix. From these arrays, $\mathsf{SA}_{2k}$ can be determined by sorting $\mathsf{SA}_k$ with $(\mathsf{RA}_k[\mathsf{SA}_k[i]], \mathsf{RA}_k[\mathsf{SA}_k[i]+k])$ as the sort key for $\mathsf{SA}_k[i]$. From $\mathsf{SA}_{2k}$, it is then easy to determine $\mathsf{RA}_{2k}$ in linear time. If sorting is also done in linear time, the prefix-doubling algorithm uses $O(n \log n)$ time for a collection of size $n$.

[1] `http://algo2.iti.kit.edu/singler/mcstl/`

The algorithm has been influenced by the suffix array construction algorithm of Larsson and Sadakane [56]. If suffix $\mathsf{SA}_k[i]$ has already been sorted, we have $\mathsf{RA}_k[i] = \mathsf{RA}_{2k}[i]$ and $\mathsf{SA}_k[i] = \mathsf{SA}_{2k}[i]$. Hence we have to handle only unsorted ranges in subsequent iterations. To make parallelization easier, we store the unsorted ranges explicitly as pairs of 32-bit integers. In the worst case, there can be almost $n/2$ unsorted ranges in both $\mathsf{SA}_k$ and $\mathsf{SA}_{2k}$, requiring up to $4n$ bytes of space. In practice, the ranges take at most $n$ bytes of space.

As the first part of the sort key is equal for all suffixes in an unsorted range, we only have to use $\mathsf{RA}_k[\mathsf{SA}_k[i] + k]$ as the sort key for $\mathsf{SA}_k[i]$. To avoid cache misses during sorting, we sort pairs $(\mathsf{SA}_k[i], \mathsf{RA}_k[\mathsf{SA}_k[i] + k])$ of two 32-bit integers, instead of retrieving the sort key indirectly. This increases the space usage of the algorithm by $4n$ bytes to a total of $12n$ to $16n$ bytes. Initially, we use the parallel quicksort from MCSTL to sort pairs $(i, T[i, i+1])$ or $(i, T[i])$, depending on whether we can pack two characters into a single 32-bit integer. Later, we divide the unsorted ranges between a number of threads, and use the standard sequential sorting algorithm from STL to sort each of the ranges.

The suffix array construction algorithm used in Fast differs from the basic version above. As we are building the suffix array for the rank array instead of the collection, we have to use 64-bit sort keys in the initial sorting, and cannot pack two adjacent characters into the key. On the other hand, as the elements of $\mathsf{RA}_k$ are 32-bit integers, we can pack both $\mathsf{RA}_k[\mathsf{SA}_k[i]+k]$ and $\mathsf{RA}_k[\mathsf{SA}_k[i]+2k]$ into the sort key of $\mathsf{SA}_k[i]$, and obtain $\mathsf{SA}_{3k}$ instead of $\mathsf{SA}_{2k}$. This prefix-tripling algorithm uses $16n$ to $20n$ bytes of memory, and is usually somewhat faster than the prefix-doubling variant.

## 5.5 Experiments

To compare the performance of Merge and Fast, we used both algorithms to build RLCSA for two data sets from Paper II. The first of them, *fiwiki*, is the Finnish language Wikipedia with full version history (42.03 gigabytes), while the other, *enwiki*, is a snapshot of the English language Wikipedia (41.46 gigabytes). Of the two data sets, fiwiki is a highly repetitive collection. The construction was done on the same system as in Section 4.4, using 8 parallel threads.

The only other CSA construction algorithm for collections that are too large to fit into memory is the algorithm of Hon et al. [38] that is essentially an earlier variant of Merge. Some implementations [53, 59] of the algorithm exist, but they can only be used for constructing uncompressed BWT for texts over 2-bit alphabets such as DNA sequences. As the algorithms are so similar to each other, any performance differences would most likely be due to implementation choices.

Another way to construct CSAs for collections larger than the main memory is to use an external memory suffix array or BWT construction algorithm. The fastest known general-purpose algorithm is the *bwte* of Ferragina et al. [15]. Distantly related to the algorithm of Hon et al. [38], bwte builds an index for the long suffixes, streams the already indexed part of text to determine the *gap array*, and uses the gap array to merge the new index with the existing BWT. The gap array, closely related to the rank array, stores the number of short suffixes falling between two lexicographically adjacent long suffixes. If a collection of size $N$ is indexed in $m$ passes, bwte takes $O(mN)$ time, streams $O(mN \log \sigma)$ bits of data, and requires $O((N/m) \log(N/m))$ bits of working space. As our test environment uses network storage with relatively low transfer rates, it was not reasonable to use it for testing external memory algorithms. Instead, we used a system with a quad-core 2.93 GHz Intel Core i7-870 processor running OS X 10.6.8 with bwte. This system had 16 gigabytes of memory and a solid-state drive.

We split both data sets into 400-megabyte subcollections for Merge and Fast, and also used 800-megabyte subcollections with Merge. We built RLCSA with default parameters $b = 32$ bytes and $d = 128$. The final index sizes were 14.33 GB for enwiki and 4.42 GB for fiwiki. For bwte, we used 1.5-gigabyte subcollections, resulting in 28 (enwiki) and 29 (fiwiki) passes over the input collection.

In the fiwiki collection, different versions of the same document are stored consecutively. This is almost the worst possible ordering for Fast. If a number of lexicographically adjacent suffixes belong to the same subcollection, they will have the same values in the rank array, and sorting them probably requires many iterations. To test the effect of document ordering, we sorted the sequences in fiwiki by their timestamps. We then compared Merge using 400 and 800-megabyte and 1.5-gigabyte subcollections with Fast using 400-megabyte subcollections on the new collection *fiwiki-sorted*.

Construction times and memory requirements can be seen in Table 5.1. Note that while the results for bwte include just BWT construction, building RLCSA would not increase construction time significantly. In general, Fast is faster than Merge with the same subcollection size, while Merge offers better time/space trade-offs. However, when the sequences have been sorted by their timestamps, Fast becomes faster than Merge with the same amount of memory, as it can build the partial indexes much faster.

| **Algorithm** | **Time** | **Space** | **Speed** | **Build** | **Rank** | **Sort** | **Merge** | **I/O** |
|---|---|---|---|---|---|---|---|---|
| **enwiki** | | | | | | | | |
| Merge-400 | 10.3 | 24.5 | 1.14 | 3.91 | 1.52 | 0.94 | 3.28 | 0.68 |
| Merge-800 | 8.6 | 27.0 | 1.36 | 3.34 | 1.50 | 0.94 | 2.14 | 0.71 |
| Fast-400 | 8.9 | 30.0 | 1.32 | 3.24 | 1.51 | 0.92 | 2.97 | 0.28 |
| bwte-1536 | 84.3 | 12.0 | 0.14 | – | – | – | – | – |
| **fiwiki** | | | | | | | | |
| Merge-400 | 11.8 | 11.6 | 1.01 | 6.96 | 1.38 | 0.87 | 2.23 | 0.37 |
| Merge-800 | 10.1 | 14.0 | 1.18 | 5.68 | 1.34 | 0.88 | 1.64 | 0.57 |
| Fast-400 | 10.5 | 19.4 | 1.14 | 6.27 | 1.37 | 0.86 | 1.73 | 0.27 |
| bwte-1536 | 72.9 | 12.0 | 0.16 | – | – | – | – | – |
| **fiwiki-sorted** | | | | | | | | |
| Merge-400 | 12.6 | 11.5 | 0.95 | 7.48 | 1.25 | 0.94 | 2.36 | 0.60 |
| Merge-800 | 11.2 | 15.0 | 1.07 | 6.38 | 1.20 | 0.96 | 2.10 | 0.52 |
| Merge-1536 | 11.3 | 22.9 | 1.06 | 7.08 | 1.18 | 0.95 | 1.66 | 0.45 |
| Fast-400 | 10.1 | 19.7 | 1.18 | 5.51 | 1.25 | 0.94 | 2.33 | 0.11 |

Table 5.1: Indexing two 41 to 42-gigabyte collections. Construction time in hours, memory usage in gigabytes, and construction speed in megabytes per second. For Merge and Fast, the numbers also include a breakdown of time usage between partial index construction (Build), rank array construction (Rank), rank array sorting (Sort), index merging (Merge), and I/O and other overhead (I/O). The number in algorithm name indicates subcollection size in megabytes.

Both Merge and Fast are much faster than bwte. While bwte is an external memory algorithm, its performance is constrained by CPU speed in practice. Most of the time is taken by the construction of the gap arrays. This requires backward searching for a volume of data equal to roughly $m/2$ times the size of the collection — more than 500 gigabytes in these experiments. As backward searching is easy to parallelize, minor modifications should make bwte more competitive on current hardware.

Even though the prefix-doubling algorithm has superlinear time complexity, increasing subcollection size from 400 megabytes to 800 megabytes decreases the time required for constructing the partial indexes. A probable explanation is that while the total amount of work increases with block size, the algorithm parallelizes better when sorting larger sets of suffixes. Merging also takes less time with larger block sizes, as fewer merges are required. Still, the effect of subcollection size on running time remains small.

# Chapter 6

# Longest common prefix array

A *compressed suffix tree (CST)* is a compressed data structure that provides similar functionality as the concrete suffix tree. While the exact list of operations varies from proposal to proposal, the extra functionality over a suffix array (see Definition 2.1) is mostly tree navigation and determining the length of the longest common prefix of the suffixes in a given subtree. The majority of compressed suffix tree proposals are actually compressed enhanced suffix arrays, combining a CSA, a compressed representation of the LCP array, and some representation of the suffix tree topology [85, 27, 69, 76, 75].

In this chapter, we investigate compressed LCP array representations and their space-efficient construction, based on Paper III. We describe a LCP array sampling mechanism that can be used in a similar way as the suffix array samples in a compressed suffix array. For regular texts, this representation offers better time/space trade-offs than the earlier compressed representations. We also describe an algorithm for constructing the LCP array directly from a compressed suffix array.

## 6.1 LCP array representations

In a plain representation of the LCP array, each element is a $\log n$-bit integer, for a total of $n \log n$ bits. Yet as the array can be derived from the text, it should be at least as compressible as the text itself. A number of different compressed representations have been proposed for the LCP array. Most of them are based on one or more of the three main ideas: variable-length codes, storing the array in text order, and sampling the array.

**Variable-length codes.** If the text is not highly repetitive, most of the LCP values are likely to be small. An easy way to utilize this to compress the LCP array is to store small values (less than 255) as 8-bit integers [1]. Large values are marked with a 255 in the array and stored explicitly as pairs $(i, \mathsf{LCP}[i])$. If a large value is needed, it can be found efficiently with binary searching. On typical texts, where large LCP values are rare, this representation takes approximately $8n$ bits of space.

A more advanced representation [12] is based on directly addressable codes [5]. The binary representation of each LCP value is broken into $b$-bit chunks. The $i$th element of array $B_1$ contains the least significant chunk of $\mathsf{LCP}[i]$, followed by a bit indicating whether more chunks are needed to encode the value. Array $B_2$ contains the next chunks of LCP values larger than $2^b - 1$, and a *rank* index is built over the indicator bits of array $B_1$ to map the elements of $B_1$ to the corresponding elements of $B_2$. If more than $2b$ bits needed to encode the largest LCP values, arrays $B_3, B_4, \ldots$ are built in a similar way. On typical texts, this representation takes $6n$ to $8n$ bits of space, with an average access time similar to one *rank* or *select* operation on a bit vector.

**Permuted LCP array.** While encoding each LCP value individually allows fast random access, the compression results are not that good. For better compression, we have to encode the values relative to other values. The key to this is the *permuted LCP (PLCP) array* $\mathsf{PLCP}$ that stores the LCP values in text order. By using the PLCP array and the suffix array, we can retrieve any LCP value as $\mathsf{LCP}[i] = \mathsf{PLCP}[\mathsf{SA}[i]]$.

**Definition 6.1.** For text $T[1, n]$ and integer $i > 1$, the *left match* of suffix $T[\mathsf{SA}[i], n]$ is the suffix $T[\mathsf{SA}[i-1], n]$.

It follows that $\mathsf{PLCP}[j]$ is the length of the longest common prefix of suffix $T[j, n]$ and its left match $T[j', n]$. As $\mathrm{lcp}(T[j'+1, n], T[j+1, n])$ is a lower bound for $\mathsf{PLCP}[j+1]$ (unless $\mathsf{PLCP}[j] = 0$), we get the following lemma.

**Lemma 6.1** ([43, 50])**.** *For* $j \in \{1, \ldots, n-1\}$, $\mathsf{PLCP}[j+1] \geq \mathsf{PLCP}[j] - 1$.

The definition and the lemma generalize for collections of texts as well.

An immediate consequence of Lemma 6.1 is that values $\mathsf{PLCP}[j] + 2j$ form a strictly increasing sequence. Hence the PLCP array can be encoded as bit vector $B_L$ of length $2n$. Individual PLCP values can then be retrieved as $\mathsf{PLCP}[j] = \mathit{select}_1(B_L, j) - 2j$. In the original proposal of Sadakane [85], bit vector $B_L$ was a succinct bit vector with constant-time *select*, taking

$2n + o(n)$ bits of space and allowing constant-time access to individual PLCP values.

For highly repetitive texts, a run-length encoded bit vector representation of the PLCP array provides even better compression.

**Definition 6.2.** A PLCP value $\mathsf{PLCP}[j]$ is *minimal*, if $j = n$ or $\mathsf{PLCP}[j] < \mathsf{PLCP}[j+1] + 1$. Value $\mathsf{PLCP}[j]$ is *maximal*, if $j = 1$ or $\mathsf{PLCP}[j-1]$ is minimal.

In a maximal run of 1-bits in bit vector $B_L$, the first 1-bit encodes a maximal PLCP value, and the last 1-bit encodes a minimal value.

**Lemma 6.2.** *Value* $\mathsf{PLCP}[j]$ *is non-minimal if and only if* $\mathsf{PLCP}[j] = \mathsf{PLCP}[j+1] + 1$.

*Proof.* By definition, $j < n$ and $\mathsf{PLCP}[j] \geq \mathsf{PLCP}[j+1] + 1$ for non-minimal $\mathsf{PLCP}[j]$. By Lemma 6.1, $\mathsf{PLCP}[j] \leq \mathsf{PLCP}[j+1] + 1$. □

**Lemma 6.3.** *The* 1*-bit encoding* $\mathsf{PLCP}[\mathsf{SA}[i]]$ *in bit vector* $B_L$ *can be a run head only if* $\mathsf{BWT}[i]$ *is also a run head.*

*Proof.* Assume that $\mathsf{BWT}[i-1] = \mathsf{BWT}[i]$, and let $\mathsf{SA}[i-1] = j'$ and $\mathsf{SA}[i] = j$. Suffix $T[j', n]$ is then the left match of suffix $T[j, n]$. As we have $LF(i-1) = LF(i) - 1$, we also have that $T[j'-1, n] = T[\mathsf{SA}[LF(i-1)], n]$ is the left match of suffix $T[j-1, n] = T[\mathsf{SA}[LF(i)], n]$. And as $T[j'-1] = \mathsf{BWT}[i-1] = \mathsf{BWT}[i] = T[j-1]$, it follows that $\mathsf{PLCP}[j-1] = \mathsf{PLCP}[j] + 1$. As $\mathsf{PLCP}[j-1]$ is non-minimal by Lemma 6.2, $\mathsf{PLCP}[j]$ is non-maximal, and the 1-bit encoding $\mathsf{PLCP}[j]$ is not a run head. □

**Corollary 6.1** ([27])**.** *The number of runs of* 1*-bits in bit vector* $B_L$ *is at most* $R$*.*

For every run of 1-bits, there is exactly one minimal and one maximal PLCP value. Note that $\mathsf{PLCP}[\mathsf{SA}[i]]$ is not necessarily a maximal value, when $\mathsf{BWT}[i]$ is a run head. Experimental results suggest that the number of runs of 1-bits in bit vector $B_L$ is usually about $2R/3$ (see Section 6.4).

Proposed run-length encoded representations of bit vector $B_L$ require at most $2R \log(n/R) + O(R) + o(n)$ [27] or $2R \log(n/R) + O(R \log\log(n/R))$ [69] bits of space, and provide fast access to individual PLCP values. The main drawback of all PLCP-based representations is that when used with a compressed suffix array, accessing *LCP* values requires using *locate*, which is an expensive operation.

**Sampled PLCP representations.** An alternative way to compress the PLCP array is to sample one out of $d'$ values and use the text and the suffix array to derive the rest [44]. Assume that we have sampled $\mathsf{PLCP}[ad']$ and $\mathsf{PLCP}[(a+1)d']$, and we want to determine $\mathsf{PLCP}[ad'+b]$ for some $b < q$. Lemma 6.1 states that $\mathsf{PLCP}[ad'] - b \leq \mathsf{PLCP}[ad'+b] \leq \mathsf{PLCP}[(a+1)d'] + d' - b$, so at most $d' + \mathsf{PLCP}[(a+1)d'] - \mathsf{PLCP}[ad']$ character comparisons are required to determine the missing value. While the number of comparisons can be large in the worst case, the average number of character comparisons over the array is $O(d')$ [50].

With a more careful selection of the sampled positions, we get similar trade-offs even in the worst case. In particular, we can store the samples in $o(n)$ bits, while requiring only $O(\log^{\delta} n)$ character comparisons in the worst case, for any $\delta > 0$ [25]. This solution is essentially the *select* structure of bit vector $B_L$ without the bit vector itself.

While the sampled PLCP representations provide attractive trade-offs when used with a plain suffix array, they are slow with a compressed suffix array. In addition to requiring the expensive *locate* operation to access LCP values, they also use the equally expensive *extract* for character comparisons.

## 6.2 Sampling the LCP array

While increasing the number of suffix array samples increases the performance of PLCP-based representations, it also quickly eliminates the size advantage of those representations. A sampled LCP representation can offer better time/space trade-offs, as individual LCP samples tend to be smaller than suffix array samples.

Assume that we have sampled the at most $R$ *minimal* PLCP values, where $\mathsf{PLCP}[j] < \mathsf{PLCP}[j+1]+1$. If we store these samples in suffix array order in the same way as suffix array samples, we can use a similar mechanism to retrieve the rest of the values. If $\mathsf{LCP}[i]$ has not been sampled, we proceed to position $\Psi(i)$ and check if $\mathsf{LCP}[\Psi(i)]$ has been sampled. If $\mathsf{LCP}[\Psi^k(i)]$ is the first sampled position we encounter, then $\mathsf{LCP}[i] = \mathsf{LCP}[\Psi^k(i)] + k$. This follows from the fact that $\mathsf{LCP}[\Psi^{k'}(i)]$ is a non-minimal PLCP value for all $k' < k$, and hence $\mathsf{LCP}[\Psi^{k'}(i)] = \mathsf{PLCP}[\mathsf{SA}[i]+k'] = \mathsf{PLCP}[\mathsf{SA}[i]+k'+1]+1 = \mathsf{LCP}[\Psi^{k'+1}(i)] + 1$ by Lemma 6.2.

**Lemma 6.4** ([50]). *For a text of length $n$, the sum of minimal or maximal PLCP values is at most $2n \log n$.*

The worst case occurs in random texts, where most of the PLCP values are both maximal and minimal, and the average value is $\Theta(\log_{\sigma} n)$ (see the analysis in Section 4.2). For a binary de Bruijn sequence, the sum of maximal values (with a looser definition, where Lemma 6.3 holds in both directions) is $(n/2)\log n - O(n)$ [50], making the bound asymptotically tight. In highly repetitive texts, the sum of minimal values tends to be less than $n$ (see Section 6.4).

If we use the bit vector of Gupta et al. [37] to mark the sampled positions, the vector takes $(1 + O(1/\log R))R\log(n/R) + O(R\log\log(n/R))$ bits of space in the worst case. As the sum of the samples is at most $2n\log n$, we can concatenate the binary representations of the samples in at most $R\log(2n\log n/R) + R$ bits. The two-level storage scheme of Ferragina and Venturini [24] allows constant-time access to the samples by using $O(R\log\log n/\log n)$ bits of extra space. Overall, the samples require

$$\left(2 + O\left(\frac{1}{\log R}\right)\right) R\log\frac{n}{R} + O\left(R\log\log\frac{n}{R}\right)$$

bits of space, which is almost the same as for one of the run-length encoded PLCP proposals [69].

To guarantee worst-case behavior and to improve the performance with highly repetitive texts, where $R \ll n$, we also sample one out of $d' > 0$ PLCP values, when the successive minimal values are spaced more than $d'$ positions apart. With the pessimistic assumption that each of these extra samples is large, requiring $\log n$ bits, the size bound becomes

$$\left(1 + O\left(\frac{1}{\log n_S}\right)\right) n_S\log\frac{n}{n_S} + R\log\frac{n}{R} + \frac{n\log n}{d'} + O\left(n_S\log\log\frac{n}{n_S}\right)$$

bits, where $n_S \le R + n/d'$ is the number of samples. In practice, an extra sample $\mathsf{PLCP}[j]$ requires roughly $\log \mathsf{PLCP}[j]$ bits and a mark in the bit vector, while an extra suffix array sample requires $2\log(n/d)$ bits in addition to the mark, where $d$ is the suffix array sample rate. Hence with typical sample rates, adding extra LCP samples is cheaper than adding the same number of suffix array samples.

**Theorem 6.1.** *Let $T[1,n]$ be a text, and let $d' > 0$ be an integer. The LCP samples for text $T$ with sample rate $d'$ require*

$$O\left(n_S\log\frac{n}{n_S}\right) + \frac{n\log n}{d'}$$

*bits of space, where $n_S \le R + n/d'$ is the number of samples, and $R$ is the number of equal letter runs in the Burrows-Wheeler transform of text $T$. A compressed suffix array that computes $\Psi$ in $t_\Psi$ time can use the samples to access any element of the LCP array in $O(d' \cdot (t_\Psi + o(\log n)))$ time.*

## 6.3 Space-efficient LCP array construction

Kasai et al. [43] introduced the first linear-time LCP array construction algorithm. As the algorithm requires the suffix array in memory, it uses $\Theta(n \log n)$ bits of space for a text of length $n$. Yet as the LCP array can be compressed into $2n + o(n)$ bits or even less, this greatly restricts the size of the texts with which the LCP array can be used. Later developments concentrated on reducing the working space to $O(n \log \sigma)$ bits [80, 50, 30, 4] or making the construction faster in practice [50, 30, 26].

The fastest algorithm so far is the one by Fischer [26]. Based on a suffix array construction algorithm using induced sorting [73], the algorithm works in linear time, yet requires $\Theta(n \log n)$ bits of working space. Another interesting algorithm is the one by Beller et al. [4] that builds the LCP array directly from a compressed suffix array. On a conceptual level, the algorithm is based on repeating the following for $k = 0, 1, \ldots$.

1. For all patterns $P$ of length $k$, let $[sp_P, ep_P]$ be the suffix array range containing the occurrences of the pattern. Assume that set $Q_k$ contains all these ranges.

2. Use backward searching to determine set $Q_{k+1}$ from set $Q_k$.

3. For each range $[sp, ep] \in Q_{k+1}$, set $\mathsf{LCP}[ep+1] \leftarrow k$, unless $\mathsf{LCP}[ep+1]$ has already been defined.

A practical variant of the algorithm works in $O(n \log n)$ time and uses roughly $n$ bytes of working space in addition to the compressed suffix array. The LCP array is written directly to disk in several passes. While the algorithm is reasonably fast and space-efficient, it cannot be used for constructing the LCP array for texts that are too large to fit into memory. This is in contrast to the compressed suffix arrays, for which such algorithms exist (see Chapter 5).

By starting from the *irreducible LCP algorithm* of Kärkkäinen et al. [50], we can design an LCP array construction algorithm that uses negligible working space in addition to the compressed suffix array and the compressed (P)LCP representation. The irreducible LCP algorithm finds the irreducible (maximal) PLCP values, computes them directly, and uses Lemma 6.2 to derive the rest of the values. As the sum of maximal PLCP values is at most $2n \log n$, the algorithm works in $O(n \log n)$ time. The PLCP values are computed in text order, making it possible to build any compressed PLCP representation directly.

The original irreducible LCP algorithm uses the text and the suffix array to identify the maximal values. With a compressed suffix array, we want to identify the irreducible values using function $\Psi$ in one pass over the text. Additionally, as computing the irreducible values also involves scanning the text forward using $\Psi$, we can avoid redundant work by finding minimal instead of maximal PLCP values.

From Lemma 6.3, we can derive the following result by noting that $\mathsf{PLCP}[j]$ is minimal if $\mathsf{PLCP}[j+1]$ is maximal.

**Lemma 6.5.** *Let* $\mathsf{SA}[i] = j$. *Value* $\mathsf{PLCP}[j]$ *can be minimal only if* $j = n$ *or if* $\mathsf{BWT}[\Psi(i)]$ *is a run head.*

Let $c$ be a character such that $C[c] < i \leq C[c+1]$. As we compute $\Psi(i) = \mathrm{select}_c(\mathsf{BWT}, i - C[c])$, we know that $\mathsf{BWT}[\Psi(i)]$ is a run head if we use a run head in $\mathsf{BWT}$ (or in bit vector $B_c$) to compute $\Psi(i)$. Equivalently, $\mathsf{BWT}[\Psi(i)]$ is a run head if and only if $\mathrm{char}(i-1) \neq \mathrm{char}(i)$ or $\Psi(i-1) \neq \Psi(i) - 1$.

While Lemma 6.5 produces false positives, we can identify true minimal values by buffering the previous candidate, until we find the next possibly minimal value. The modified irreducible LCP algorithm can be seen in Figure 6.1.

A two-pass variant of the same algorithm can be used to sample the LCP array. In the first pass, we scan the CSA in suffix array order, and find the minimal samples. As the samples are output in suffix array order, we can compress them immediately, using negligible working space. The second pass is in text order, as in the original algorithm. We output one out of $d'$ consecutive non-minimal values as pairs $(i, \mathsf{LCP}[i])$, deriving $\mathsf{LCP}[i]$ from the next sample. Once the non-minimal samples have been determined, we sort them in suffix array order, and merge them with the minimal samples.

**Theorem 6.2.** *Given a compressed suffix array for a text of length* $n$, *the irreducible LCP algorithm computes the PLCP array in negligible working space in addition to the CSA and the PLCP array. The running time of the algorithm is equivalent to extracting* $O(n \log n)$ *characters from the text using the CSA. A two-pass version of the algorithm samples the LCP array with sample rate* $d' > 0$ *in the same time bound, while using* $O((n/d') \log n)$ *bits of additional working space.*

```
function lcp($i$)
    $(i', k) \leftarrow (i - 1, 0)$
    $c \leftarrow \mathrm{char}(i)$
    while $i' \in C_c$
        $(i', i, k) \leftarrow (\Psi(i'), \Psi(i), k + 1)$
        $c \leftarrow \mathrm{char}(i)$
    return $k$

function irreducibleLCP($\mathsf{SA}^{-1}[1]$)
    $\mathsf{PLCP}[1] \leftarrow 0$
    $(i, j, j') \leftarrow (\mathsf{SA}^{-1}[1], 1, 2)$
    while $j < n$
        $c \leftarrow \mathrm{char}(i)$
        if $i - 1 \notin C_c$ or $\Psi(i - 1) \neq \Psi(i) - 1$
            $x \leftarrow \mathrm{lcp}(\Psi(i))$
            $\mathsf{PLCP}[j', j + 1] \leftarrow (x + j + 1 - j', \ldots, x)$
            $j' \leftarrow j + 2$
        $(i, j) \leftarrow (\Psi(i), j + 1)$
    return PLCP
```

Figure 6.1: The irreducible LCP algorithm for computing the PLCP array directly from a compressed suffix array. The algorithm maintains an invariant that $\mathsf{SA}[i] = j$, while $\mathsf{PLCP}[j']$ is the maximal value in the current run.

## 6.4 Implementation and experiments

LCP samples, bit vector encodings of the PLCP array, and their construction algorithms have been implemented as a part of RLCSA (see Chapter 4). For PLCP, we use a run-length encoded bit vector with highly repetitive data sets, and a succinct bit vector with regular data sets. For marking the sampled LCP positions, we similarly use a gap encoded bit vector with highly repetitive data sets, and a succinct vector with the regular ones. The sampled values are encoded with $\delta$-codes, and stored in a stripped-down version of the gap encoded bit vector for fast access.

The succinct bit vector is a practical implementation designed for current hardware. For *rank*, we divide the vector into 256-bit blocks, and store the number of 1-bits before each block in $\log n$ bits. Solving *rank* then requires retrieving the stored value for the correct block, and counting the number of 1-bits in the block up to the queried position using the 64-bit popcount function provided in GCC. The function compiles either into a single instruction or a small subroutine, depending on architecture.

For *select*, the implementation uses two levels of indexes. The first one stores $n/256$ pointers to blocks that contain the 1-bit of rank $i \cdot 256 n_1/n + 1$, for $i = 0$ to $n/256-1$. With this index, we get a range of blocks that contains the queried 1-bit. If the range spans more than 16 blocks, we use binary search in the *rank* index to narrow it down. When the range becomes short enough, we continue with linear search in the *rank* index to find the correct block, and resort to popcount to compute the answer within the block.

To avoid redundant work in LCP sampling and PLCP construction, we interleave the computation of minimal values with the main loop. When sampling the LCP array, we make both of the passes in text order, and store all samples in an array of pairs $(i, \mathsf{LCP}[i])$ before compressing them.

For the experiments, we used the same system as in Section 4.4. Only once core was used in the experiments. We used the same four data sets as in Paper III. As regular data sets, we used human DNA sequences (dna) and English language texts (english) from the Pizza & Chili Corpus [18]. As highly repetitive data sets, we used Finnish language Wikipedia with full version history (fiwiki) and the genomes of 36 strains of *Saccharomyces paradoxus* (para) (see also Chapter 4). When the data set was much larger than 400 megabytes, a 400 MB prefix was used instead. Further information on the data sets can be found in Table 6.1.

The construction times reported in Table 6.1 are for the highest number of SA and LCP samples. Note that sample rates had no significant impact on construction times. With the regular data sets, the irreducible LCP algorithm was very slow, as its time complexity scales with the sum of

| | | | Minimal | | Construction | | |
|---|---|---|---|---|---|---|---|
| **Data set** | **Size** | **Runs** | **Number** | **Sum** | **PLCP** | **Samples** | **Induced** |
| dna | 385 | 243.49 | 158.55 | 2215 | 2402 | 2695 | 47 |
| english | 400 | 156.35 | 99.26 | 1052 | 1181 | 1471 | 76 |
| fiwiki | 400 | 1.79 | 1.15 | 117 | 210 | 365 | 46 |
| para | 409 | 15.64 | 10.05 | 299 | 374 | 590 | 55 |

Table 6.1: Data sets used in LCP experiments. Size in megabytes, millions of runs in BWT, number and sum of minimal samples in millions, and construction time in seconds. Construction times for PLCP and (LCP) Samples are from using the irreducible LCP algorithm, while Induced is for building the LCP array with the algorithm of Fischer [26], which is currently the fastest LCP construction algorithm.

minimal PLCP values. A parallel version that scans different sequences in different threads might still be as fast as direct CSA construction (see Chapter 5), as the basic CSA operations parallelize well. The algorithm was much faster with the highly repetitive data sets, while still being 5 to 10 times slower than the LCP construction algorithm of Fischer [26].

As noted in Section 6.1, the number of minimal PLCP values was roughly 2/3 times the number of equal letter runs in the Burrows-Wheeler transform of the text. As the number of minimal PCLP values is the same as the number of runs of 1-bits in the PLCP vector $B_L$, this means that the size bounds for the run-length encoded PLCP variants are significantly larger than the size in practice.

To compare the time/space trade-offs offered by PLCP bit vectors and LCP samples, we built RLCSA with sample rates $d = 8, 16, 32, 64$ for the regular data sets, and $d = 32, 64, 128, 256$ for the highly repetitive data sets. We used LCP sample rates $d' = 8, 16$ for the regular data sets, and $d' = 16, 32, 64$ for the highly repetitive ones. Instead of measuring the average time required for a random LCP query, we used a measure that is independent of the underlying CSA implementation, counting the average number of steps of $\Psi$ required to find a sampled SA or LCP position. The results can be seen in Figure 6.2. As a lower bound for the size of any PLCP-based representation, we also included the results for *locate* in the figures.

With the regular data sets and sparse SA sampling, the performance of LCP samples was superior to the PLCP-based approach, while increasing the overall index size only slightly. As 25% (english) to 40% (dna) of text positions were sampled, a sampled position was found in a couple of steps

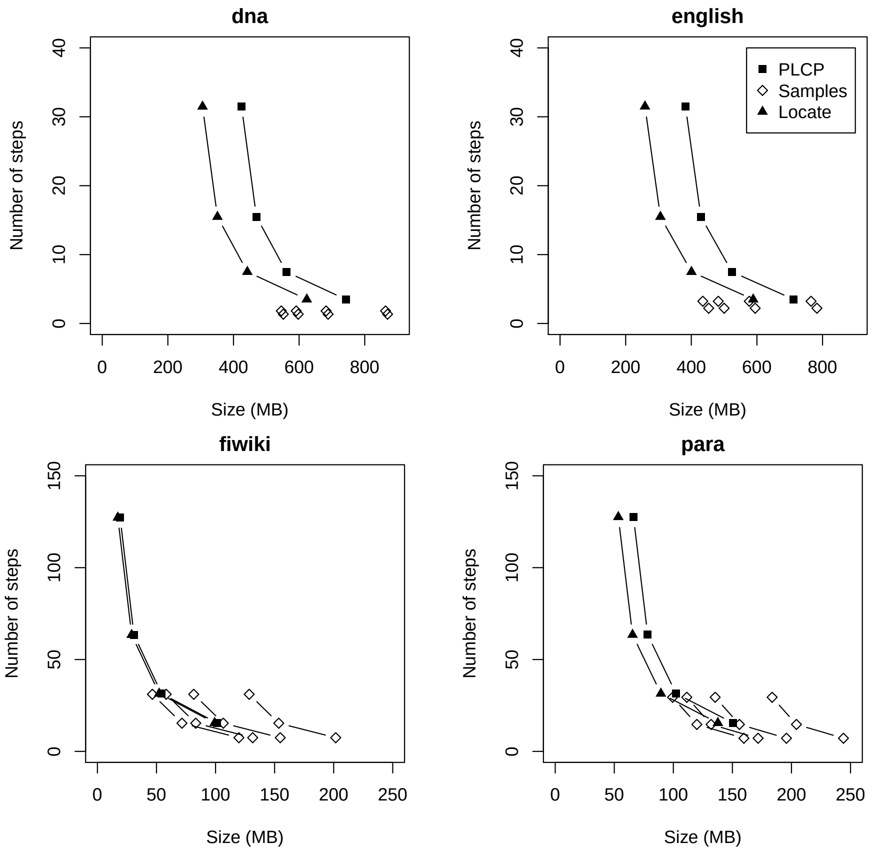


Figure 6.2: Experimental results for $10^6$ random LCP queries with LCP samples and PLCP, with $10^6$ random *locate* queries as a lower bound for any PLCP-based representation. Index size in megabytes and the average number of steps required to reach a SA/LCP sample. The results with LCP samples have been grouped by SA sample rate.

most of the time. With denser suffix array sampling, this advantage mostly disappeared, as the *locate* queries were no longer that expensive.

As there were only a few minimal samples in the highly repetitive data sets (see Table 6.1), the solution using LCP samples relied mostly on the non-minimal extra samples. This made the LCP samples significantly larger than the run-length encoded PLCP bit vector. While the LCP samples still offered better time/space trade-offs than any PLCP-based approach in some cases, the improvement was not significantly better than what could be achieved by denser suffix array sampling.

We also measured the average number of steps required for computing the LCP values directly. As there were long repetitions even in the regular data sets, the results (2424 steps in dna and 5786 steps in english) were not competitive with the other approaches. Still, as many of the minimal samples are small, it should be possible to decrease the size of the samples without sacrificing too much performance by storing only large minimal samples, and computing the small ones directly when required.

# Chapter 7

# Generalized compressed suffix array

The compressed suffix arrays discussed so far index either single sequences or collections of sequences. This is not a fundamental limitation, however. In this chapter, based on Paper IV, we generalize the compressed suffix arrays to handle a certain class finite automata. This class of automata can recognize all finite languages and some infinite languages as well.

Backward searching using BWT is based on the following property: text positions containing character $c$ are sorted in the same order as text positions preceded by character $c$. If we consider the sequence a finite automaton, we could say that nodes labeled with character $c$ are sorted in the same order as those nodes with a predecessor labeled with character $c$. To use this idea to index finite automata, we need to solve two problems: handling nodes with multiple predecessors or successors (Section 7.1), and constructing an automaton that can be sorted in the desired way (Section 7.2).

In the general case, the sorted automaton can be exponentially larger than the minimal deterministic automaton recognizing the same language. However, if only a small fraction of the nodes of the minimal automaton have multiple successors, most of the nodes can be sorted by the label of the unique path of length $k$ (for some $k > 0$) starting from the node. For these languages, the sorted automaton is not much larger than the minimal one. An important example of this kind of automata are those arising from a reference sequence and a set of substitutions, insertions, and deletions, or from a multiple alignment of sequences. In the latter case, the automaton will recognize not only the original sequences, but all their plausible recombinations as well.

## 7.1 Indexing finite languages

The *XBW transform* [19] is a generalization of the Burrows-Wheeler transform for labeled trees, where leaf nodes and internal nodes are labeled with different alphabets. Each internal node of the tree is represented as a concatenation of the labels of its children. These representations, sorted in lexicographic order according to the path labels from the node to the root, form sequence $\mathsf{BWT}$. The starting position of each internal node in $\mathsf{BWT}$ is marked by an 1-bit in bit vector $F$, so that the node with lexicographic rank $i$ can be found as $\mathsf{BWT}[\mathrm{select}_1(F, i), \mathrm{select}_1(F, i+1) - 1]$.

XBW supports tree navigation with generalizations of functions $LF$ and $\Psi$. In downward functions such as $LF$, the lexicographic ranks returned by the regular versions of the functions are converted into BWT ranges by using *select* on bit vector $F$, as above. Upward functions such as $\Psi$ work in the opposite way, converting BWT ranges into lexicographic ranks by using *rank* on bit vector $F$, before calling the regular version of the function.

**Generalization for finite automata.** Bit vector $F$, mapping lexicographic ranks into BWT ranges, allows a single node to have multiple predecessors. We can use a similar idea to allow multiple successors, extending XBW from trees to finite automata. The idea is to use another bit vector $M$ to encode the number of outgoing edges, so that the node with lexicographic rank $i$ has $\mathrm{select}_1(M, i+1) - \mathrm{select}_1(M, i)$ outgoing edges.[1] For convenience, we assume that the final node $V_{|V|}$ has a single outgoing edge to the initial node $V_1$.

Backward navigation ($LF$) first uses bit vector $F$ to convert lexicographic ranks into a BWT range, then calls the regular version of the function, and finally uses bit vector $M$ to convert the edge range into lexicographic ranks. Forward navigation ($\Psi$) uses bit vectors $M$ and $F$ in the opposite way. See Figure 7.1 for pseudocode for basic navigation functions, and below for exact definitions of the functions.

**Definitions.** As mentioned in the beginning of the chapter, the automaton must have a certain property in order for functions $LF$ and $\Psi$ to work. We call this property prefix-range-sortedness.

**Definition 7.1.** Let $A = (V, E)$ be a finite automaton, and let $v \in V$ be a node. Let $rng(v)$ be the smallest (open, semiopen, or closed) lexicographic range containing all suffixes that can be recognized from node $v$. Node $v$

[1] This definition of bit vector $M$ makes the generalization simpler than the original definition.

```
**function** LF($[sp, ep], c$)
    $[sp, ep] \leftarrow [\mathrm{select}_1(F, sp), \mathrm{select}_1(F, ep + 1) - 1]$
    $[sp, ep] \leftarrow [C[c] + \mathrm{rank}_c(\mathsf{BWT}, sp - 1) + 1, C[c] + \mathrm{rank}_c(\mathsf{BWT}, ep)]$
    $[sp, ep] \leftarrow [\mathrm{rank}_1(M, sp), \mathrm{rank}_1(M, ep)]$
    **return** $[sp, ep]$

**function** $\Psi(i, j)$
    $c \leftarrow \mathrm{char}(i)$
    $i \leftarrow \mathrm{select}_1(M, i) + j - 1$
    $i \leftarrow \mathrm{select}_c(\mathsf{BWT}, i - C[c])$
    $i \leftarrow \mathrm{rank}_1(F, i)$
    **return** $i$
```

Figure 7.1: Pseudocode for the basic navigation functions $LF$ and $\Psi$.

is *prefix-range-sorted*, if no suffix $S \in rng(v)$ is recognized from any other node $v' \neq v$. Automaton $A$ is prefix-range-sorted, if all nodes are prefix-range-sorted.

In the following, we use a stronger definition to simplify the discussion. The results for prefix-sorted automata generalize for prefix-range-sorted automata as well.

**Definition 7.2.** Let $A$ be a finite automaton, and let $v \in V$ be a node. Node $v$ is *prefix-sorted* by prefix $p(v)$, if the labels of all paths from $v$ to $v_{|V|}$ share a common prefix $p(v)$, and no path from any other node $u \neq v$ to $v_{|V|}$ has $p(v)$ as a prefix of its label. Automaton $A$ is prefix-sorted, if all nodes are prefix-sorted.

We can use the prefixes $p(v)$ to sort the nodes of a prefix-sorted automaton in lexicographic order. If we do so, then we have also sorted the outgoing edges $(u, v)$ using sequences $\ell(u)p(v)$ as sort keys, and the edge encoded by bit $M[i]$ has lexicographic rank $i$. This is the key for functions $LF$ and $\Psi$ to work properly. For any given character $c$, all outgoing edges from nodes with label $c$ are lexicographically adjacent, and they are sorted by prefix $p(v)$ of the destination node. Similarly, all occurrences of character $c$ in $\mathsf{BWT}$ encode an incoming edge from a node with label $c$, and these edges are also sorted by prefix $p(v)$ of the destination node. Hence the incoming edge labeled by the $j$th occurrence of character $c$ is the same edge as the outgoing edge of rank $C[c] + j$, and the other way around. Note that $C[c]$ stores the number of occurrences of characters smaller than $c$ in $\mathsf{BWT}$, not the number of nodes with label smaller than $c$.

**Operations.** We define the basic navigation functions in the following way.

- $LF([sp, ep], c)$ is the lexicographic range of nodes with label $c$ that have a successor in the lexicographic range $[sp, ep]$. This is essentially a step of backward searching with character $c$.
- $\Psi(i, j)$ is the lexicographic rank of the $j$th successor of the node with lexicographic rank $i$.
- char($i$) is the label of the node with lexicographic rank $i$.

These operations can be used to support the following generalization of suffix array functionality (see Definition 2.1).

- *find*$(P)$ returns the lexicographic range $[sp, ep]$ of nodes recognizing any suffix that has pattern $P$ as its prefix.
- *locate*$(i)$ returns a numerical value corresponding to the node with lexicographic rank $i$.
- *extract*$(i, \mathsf{P})$ returns the label of path $\mathsf{P}$ starting from the node with lexicographic rank $i$.

We can support *find* by replacing the first two lines of the loop body in Figure 2.3 with function $LF$ from Figure 7.1.

For *locate*, we assume that there is a (not necessarily unique) numerical value $id(v)$ attached to each node $v \in V$. Examples of these values include node ids (so that $id(v_i) = i$) and positions in a multiple alignment. To avoid excessive sampling of node values, $id(v)$ should be $id(u) + 1$ whenever $(u, v)$ is the only outgoing edge from $u$ and the only incoming edge to $v$.

We sample $id(u)$, if there are multiple outgoing edges from node $u$, or if $id(v) \neq id(u) + 1$ for the only outgoing edge $(u, v)$. We also sample one out of $d$ node values, given sample rate $d > 0$, on paths of at least $d$ nodes without any samples. The sampled values are stored in the same order as the nodes, and their positions are marked in bit vector $B_s$.

As we have sampled all nodes with multiple successors, we can use the *locate* algorithm of the CSA family directly with our new function $\Psi$. To retrieve $id(u)$ for node $u$ of lexicographic rank $i$, we first check if $B_s[i] = 1$, and return sample $rank_1(B_s, i)$, if this is the case. Otherwise we follow the only outgoing edge $(u, v)$ by using function $\Psi$, and continue from node $v$. When we find a sampled node $w$, we return $id(w) - k$, where $k$ is the number of steps taken by using $\Psi$.

In *extract*, we assume that the description of path $\mathsf{P}$ allows us to determine in constant time, which outgoing edge we should take, and have we already finished the path. With such description, we can use function $\Psi$ to move forward on the path, and function $\mathrm{char}(\cdot)$ to read the next character of the path label. The algorithm is similar to the *extract* algorithm of the CSA family, with the exception that we already know the lexicographic rank of the initial node. This is because we might be using a node value scheme that does not allow mapping node values to lexicographic ranks.

We call a compressed self-index based on this generalization of the XBW transform a *generalized compressed suffix array (GCSA)*. As each step of $LF$ and $\Psi$ requires both *rank* and *select*, we get the following generalization of Theorem 3.1.

**Theorem 7.1.** *Let* rank *and* select *on binary sequences take* $t_R$ *and* $t_S$ *time, respectively. Then a generalized compressed suffix array with sample rate* $d$ *supports* $\mathrm{find}(P)$ *in* $O(|P| \cdot (t_R + t_S))$ *time,* locate *in* $O(d \cdot (t_R + t_S))$ *time, and* $\mathrm{extract}(i, \mathsf{P})$ *in* $O(|\mathsf{P}|(t_R + t_S))$ *time.*

**A better encoding.** Recall that the size bound of RLCSA (Theorem 4.1) is worse than for plain run-length encoding of the BWT (Section 2.3). This is because RLCSA has to encode each character of $\mathsf{BWT}$ with either a 0-bit or an 1-bit in every bit vector $B_c$, while the run-length encoding encodes each character only once. In GCSA, we can use this redundancy for our advantage.

As a prefix-range sorted automaton is reverse deterministic, each node can have at most one predecessor with a certain label. Hence the section of $\mathsf{BWT}$ corresponding to a node can have at most one occurrence of each character, meaning that we can put all these predecessor labels into the same position in bit vectors $B_c$. As the bit $B_c[i]$ now determines, whether the node with lexicographic rank $i$ has a predecessor with label $c$, we no longer have to use bit vector $F$ to map between lexicographic ranks and BWT ranges. This is a major speedup in practice, as we get rid of one third of bit vector operations.

## 7.2 Construction algorithm

A straightforward way of constructing a prefix-sorted automaton from a finite automaton recognizing a finite language is via a prefix-doubling algorithm (see Section 5.4). The algorithm consists mostly of sorting, scanning, and database joins. Hence it can be efficiently implemented in parallel, distributed, and external memory settings.

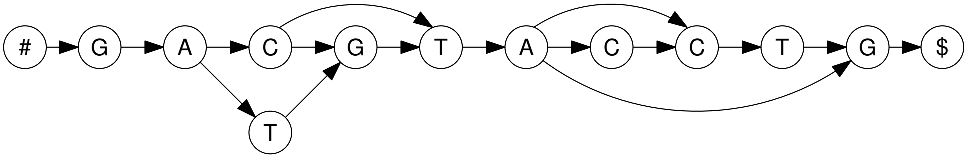


Figure 7.2: A reverse deterministic automaton corresponding to the first 10 positions of the multiple alignment in Figure 2.1.

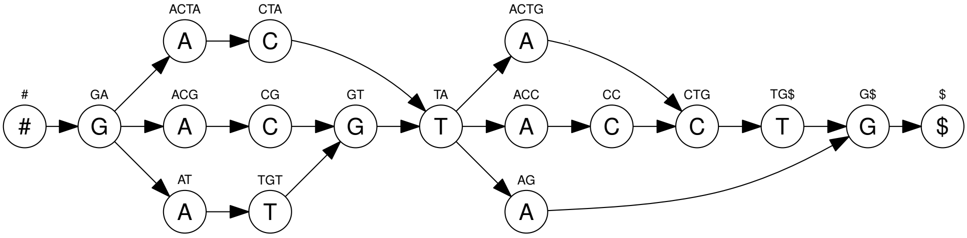


Figure 7.3: A prefix-sorted automaton built for the automaton in Figure 7.2. The strings above nodes are prefixes $p(v)$.

**Theorem 7.2.** *Assume we have a length $n$ multiple alignment of $r$ sequences over alphabet of size $\sigma$. We can build a prefix-range-sorted automaton recognizing all paths through the alignment in $O(nr + |V'| \log n + |E'|)$ time and $O(nr \log \sigma + |V'| \log|V'| + |E'| \log|E'|)$ bits of space, where $V'$ and $E'$ are the largest intermediate sets of nodes and edges, respectively.*

*Proof.* From Lemmas 7.1, 7.2, and 7.3 below. □

The sizes of the largest intermediate sets of nodes and edges are analyzed in a restricted model in Section 7.3. An example of construction can be seen in Figures 7.2 and 7.3 and Table 7.1.

| | $ | ACC | ACG | ACTA | ACTG | AG | AT | CC | CG | CTA | CTG | G$ | GA | GT | TA | TG$ | TGT | # |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BWT | G | T | G | G | T | T | G | A | A | A | AC | AT | # | CT | CG | C | A | $ |
| $M$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 100 | 1 | 100 | 1 | 1 | 1 |

Table 7.1: GCSA for the automaton in Figure 7.3. Nodes are identified by prefixes $p(v)$.

**Building a reverse deterministic automaton.** With the following algorithm, we can build a reverse deterministic automaton that recognizes all paths through a multiple alignment of sequences. The same approach, when used with a reference sequence and a set of edit operations, is essentially a variant of the textbook algorithm for determinizing finite automata.

In the following, we assume that the alignment consists of sequences $S_1, \dots, S_r$ of length $n$, possibly containing gap characters $-$. Sequences $S_i$ and $S_{i'}$ are considered to be equivalent at position $j$, if $S_i[j] = S_{i'}[j] \neq -$. We can allow edit operations longer than one character by using a context to determine the equivalence of two positions. With context length $k \geq 0$, sequences $S_i$ and $S_{i'}$ are equivalent at position $j$, if $S_i[j] = S_{i'}[j] \neq -$ and the next $k$ non-gap characters in the sequences are also equal.

The algorithm works in one pass from right to left. Assume that we have already processed positions $j+1$ to $n$ and created the corresponding part of the automaton. For each sequence $S_i$ with a non-gap character in column $j$, we first create a temporary node $v_{i,j}$ and an edge from $v_{i,j}$ to the node corresponding to the next non-gap character in sequence $S_i$. Next, we merge the temporary nodes for those sequences that are equivalent at position $j$.

Finally, we find the preceding non-gap characters for all sequences with a non-gap character at position $j$. Assume that two or more sequences that are equivalent at position $j$ have $c$ as the preceding non-gap character. If these characters $c$ occur at different positions, we move them all to the rightmost of these positions. This way, the node $v_{i,j}$ corresponding to the equivalent sequences will only have one predecessor with label $c$.

**Lemma 7.1.** *Let $n$ be the length of the multiple alignment, $r$ the number of sequences, and $\sigma$ the size of the alphabet. Building a reverse deterministic automaton takes $O(nr)$ time and requires $O(nr \log \sigma + |E| \log |E|)$ bits of space, where $E$ is the set of edges of the automaton.*

Note that each position can be processed in $O(r)$ amortized time, regardless of context length, by keeping the suffixes $S_i[j]$ in sorted order and maintaining the lengths of the longest common prefixes of lexicographically adjacent suffixes.

**Creating a prefix-sorted automaton.**

**Definition 7.3.** Let $A$ be a finite automaton recognizing a finite language, and let $k > 0$ be an integer. Automaton $A$ is *$k$-sorted* if, for every node $v$, the labels of all paths from $v$ to $v_{|V|}$ share a common prefix $p(v, k)$ of length $k$, or if node $v$ is prefix-sorted by prefix $p(v, k)$ of length at most $k$.

Every automaton is 1-sorted. Automaton $A$ is prefix-sorted if and only if it is $n$-sorted, where $n$ is the length of the longest string in $L(A)$.

Starting from a reverse deterministic automaton $A = A_0$, we create the nodes of automata $A_i = (V_i, E_i)$ for $i = 1, 2, \dots$ that are $2^i$-sorted, until we get an automaton that is prefix-sorted. For every node $v \in V_i$, let $P(v)$ be the path of $A$ corresponding to prefix $p(v, 2^i)$. We store the first and the last nodes of this path as $\mathit{from}(v)$ and $\mathit{to}(v)$, and set $\mathit{rank}(v)$ to be the lexicographic rank of prefix $p(v, 2^i)$ among all distinct prefixes $p(u, 2^i)$ of nodes $u \in V_i$. If node $v$ has unique $\mathit{rank}(v)$ value, then it is prefix-sorted.

The basic step of the algorithm is the *doubling* step from $A_i$ to $A_{i+1}$. If node $u \in V_i$ is prefix-sorted, we *duplicate* it as $w \in V_{i+1}$, and set $\mathit{rank}(w) = (\mathit{rank}(u), 0)$. Otherwise we create a *joined* node $uv \in V_{i+1}$ for every node $v \in V_i$ such that $P(uv) = P(u)P(v)$ is a path in $A$, and set $\ell(uv) = \ell(u)$ and $\mathit{rank}(uv) = (\mathit{rank}(u), \mathit{rank}(v))$. As path $P(uv)$ exists if and only if there is an edge $(\mathit{to}(u), \mathit{from}(v)) \in E_0$, this essentially requires two database joins.[2] When the nodes of $A_{i+1}$ have been created, we sort them by their ranks, and replace the pairs of integers with integer ranks.

The doubling step is followed by the *pruning* step, where we merge equivalent nodes. The nodes in $V_{i+1}$ are sorted by their $\mathit{rank}(\cdot)$ values. If all nodes sharing a certain $\mathit{rank}(\cdot)$ value also share their $\mathit{from}(\cdot)$ node, these nodes are equivalent, and can be merged. Merging makes the resulting node prefix-sorted.

**Lemma 7.2.** *Prefix-doubling algorithm creates the nodes of a prefix-sorted automaton equivalent to $A$ in $O(|V'| \log n)$ time and $O(|V'| \log |V'|)$ bits of space in addition to automaton $A$, where $V'$ is the largest set of nodes during construction, and $n$ is the length of the longest string in $L(A)$.*

The lemma assumes using a linear-time integer sorting algorithm.

**Creating the edges.** Let $A = (V, E)$ be a reverse deterministic automaton recognizing a finite language, and let $W$ be the set of nodes of an equivalent prefix-sorted automaton. To create the edges, we first merge nodes with adjacent $\mathit{rank}(\cdot)$ values, if they share their $\mathit{from}(\cdot)$ node. The resulting set $V'$ is the set of nodes of a prefix-range-sorted automaton $A' = (V', E')$ equivalent to automaton $A$. The set of edges $E'$ can be constructed efficiently from automaton $A$ and the set of nodes $V'$.

The key to edge construction is that for each node $v \in V'$, the set of $\mathit{from}(u)$ nodes for the predecessors $u$ of node $v$ is the same as the set of

[2] Juha Kärkkäinen noted that one join is enough, if we replace $\mathit{to}(u)$ with the destination nodes $w$ for all edges $(\mathit{to}(u), w) \in E_0$.

predecessors of node *from*$(v)$. With automaton $A$ and the set of nodes $V'$, we can output the edges $(u, v) \in E'$ initially as pairs $(\mathit{from}(u), v)$, sorted by $(\ell(\mathit{from}(u)), \mathit{rank}(v))$. Note that this is the same order as sorting the edges by $\mathit{rank}(u)$.

We can map nodes *from*$(u)$ to nodes $u$ by scanning the sorted lists of nodes and edges. As every node has at least one outgoing edge, and no adjacent nodes share their *from*$(\cdot)$ value, all adjacent edges with the same *from*$(\cdot)$ values start from the current node. When the *from*$(\cdot)$ value changes in the list of edges, we advance to the next node.

**Lemma 7.3.** *Creating the edges of prefix-range-sorted automaton $A'$ takes $O(|W| + |E'|)$ time and requires $O(|W| \log|W| + |E'| \log|E'|)$ bits of space, where $W$ is the set of nodes of an equivalent prefix-sorted automaton.*

## 7.3 Analysis

**Languages recognized by prefix-range-sorted automata.** As shown in Section 7.2, every finite language can be recognized by a prefix-range-sorted automaton. There are also some infinite languages that can be recognized by such automaton. For example, consider the regular language $\{\#x\$ \mid x \in \{a, b\}^*\}$. The minimal automaton recognizing this language is prefix-range-sorted, as each node has a distinct label.

Not all regular languages have prefix-range-sorted automata, however. Consider, for example, the language $L = \{\#x\$ \mid x \in \{a, b\}^* \cup \{a, c\}^*\}$. Assume that there is a prefix-range-sorted automaton that recognizes the language. Suffixes $B_n = a^n b\$$ and $C_n = a^n c\$$ must be recognized from different nodes, as $bB_n$ is a suffix of language $L$, while $bC_n$ is not. Because $B_{n+1} < C_{n+1} < B_n$, suffixes $B_n$ and $B_{n+1}$ must also be recognized from different nodes. As the automaton must have an infinite number of nodes, it cannot be a finite automaton.

**Size of the automaton.** We analyze the size of the automata created by the doubling algorithm in the following model. Let $S[1, n]$ be a reference sequence, and let $p$ be the mutation rate. For each position $i = 1, \dots, n$, the initial automaton $A$ has a node $u_i$ with label $\ell(u_i) = S[i]$, randomly chosen from alphabet $\Sigma$. With probability $p$, there is also another node $w_i$ with a random label $\ell(w_i) \in \Sigma \setminus \{S[i]\}$. The automaton has edges from all nodes at position $i$ to all nodes at position $i + 1$.

**Definition 7.4.** Let $k > 0$ be an integer. A *k-path* in an automaton is a path of length $k$, or a shorter path ending at the final node.

Let $k > 0$ be an integer. For any position $i$, let $X_{i,k}$ be the number $k$-paths starting from position $i$. If there are $j$ mutated positions covered by these paths, then $X_{i,k} = 2^j$, and each of the paths has a different label. The number of mutations is binomially distributed, with path length and mutation probability as the parameters. From the moment-generating function for binomial distribution, we get

$$\mathrm{E}\left[X_{i,k}\right] = \sum_{j=0}^{k} \Pr(X_{i,k} = 2^j) 2^j \leq (1+p)^k. \tag{7.1}$$

For positions $i = 1, \ldots, n-k+1$, this is an equality.

Let $A_h$ be the $2^h$-sorted automaton created by the prefix-doubling algorithm. By summing Equation 7.1 for all positions in the reference sequence, and including the initial and the final nodes, we get $N(2^h) = n(1+p)^{2^h} + 2$ as an upper bound for the expected number of nodes in $A_h$. As the expected number of predecessors for any node at position $i > 1$ is $(1+p)$, we get $N(2^h)(1+p)$ as an upper bound for the expected number of edges.

Consider the expectation $\mathrm{E}\left[X_{i,k} X_{i',k}\right]$ for a pair of text positions $i < i'$. If $i' \geq i+k$, then the random variables are independent, and the expectation becomes

$$\mathrm{E}\left[X_{i,k} X_{i',k}\right] = \mathrm{E}\left[X_{i,k}\right] \mathrm{E}\left[X_{i',k}\right] \leq (1+p)^{2k}. \tag{7.2}$$

Otherwise assume that the paths starting from positions $i$ and $i'$ overlap in $k' < k$ positions. Then the expectation is a product of the expectations of three independent random variables $X_{i,k-k'}$, $X^2_{i',k'}$, and $X_{i'+k',k-k'}$. By using the moment-generating function, we get

$$\mathrm{E}\left[X_{i,k} X_{i',k}\right] \leq (1+p)^{2(k-k')}(1+3p)^{k'} \leq (1+p)^{3k}. \tag{7.3}$$

**Definition 7.5.** A pair of nodes of automaton $A_h$ *collides*, if the corresponding $2^h$-paths have identical labels.

Automaton $A_h$ is prefix-sorted, if it has no colliding pairs. Two nodes can collide only, if the $2^h$-paths are of length $2^h$ and start from different positions in the reference sequence. By Equations 7.2 and 7.3, the expected number of colliding pairs is at most

$$\sum_{i<i'} \mathrm{E}\left[X_{i,2^h} X_{i',2^h} / \sigma^{2^h}\right] \leq n^2 (1+p)^{3 \cdot 2^h} / \sigma^{2^h}. \tag{7.4}$$

**Lemma 7.4.** *Let $n$ be the length of the reference sequence, $\sigma$ the size of the alphabet, and $p < \sigma^{1/3} - 1$ the mutation rate. For any $\varepsilon > 0$, the largest automaton created by the prefix-doubling algorithm has at most $n(1+p)^k + 2$ nodes with probability $1 - \varepsilon$, where $k = 2 \log_\sigma \frac{n^2}{\varepsilon} / (1 - 3 \log_\sigma (1+p))$.*

*Proof.* We want to find $k = 2^h$, for an integer $h$, such that the expected number of colliding pairs in automaton $A_h$ is at most $\varepsilon$. Then, by Markov's inequality, the probability of having a colliding pair is at most $\varepsilon$. If this happens after $h$ doubling and pruning phases, the expected number of nodes in the largest automaton created is at most $N(k) = n(1+p)^k + 2$.

By using the bound for the expected number of colliding pairs from Equation 7.4, we get

$$\frac{n^2(1+p)^{3k}}{\sigma^k} \leq \varepsilon \iff \frac{\log_\sigma \frac{n^2}{\varepsilon}}{1 - 3\log_\sigma(1+p)} \leq k.$$

As $k$ has to be a power of two, $2\log_\sigma \frac{n^2}{\varepsilon}/(1 - 3\log_\sigma(1+p))$ is an upper bound for the smallest suitable $k$. □

With reasonable mutation rates, the expected number of nodes and edges is at most $n(1+p)^{O(\log_\sigma n)} + O(1)$.

**Theorem 7.3.** *Let $n$ be the length of the reference sequence, $\sigma$ the size of the alphabet, and $p$ the mutation rate. If $1/p = \Omega(\log_\sigma n)$, then the expected number of nodes and edges in the largest automaton created by the prefix-doubling algorithm is $O(n)$.*

## 7.4 Implementation and experiments

We have implemented GCSA in C++, using the components from the implementation of RLCSA (see Section 4.3). For each character $c \in \Sigma \cup \{\#\}$, we use a gap encoded bit vector to mark the occurrences of $c$ in $\mathsf{BWT}$. Bit vector $M$ is run-length encoded, as it usually consists of long runs of 1-bits. Bit vector $B$ marking the sampled positions is gap encoded, while the samples are stored using $\lceil \log(id_{\max} + 1) \rceil$ bits each, where $id_{\max}$ is the largest sampled value. Block size is set to 32 bytes in all bit vectors.

For our experiments, we used the same system as in Section 4.4. The construction algorithms were parallelized, while the rest of the experiments used only one core. As our test data, we used a multiple alignment of four different assemblies of the human chromosome 18 (about 76 million base pairs each).[3] We built a GCSA with sample rate $d' = 16$ for the alignment, as well as RLCSA (sample rate $d = 32$) for the four sequences. We searched for exact matches of 10 million Illumina/Solexa reads of length 56, sequenced from the whole genome, as both regular patterns and reverse complements. Table 7.2 lists the results of these experiments.

[3] See Paper IV for a description of the sequences and the alignment.

| | | Construction | | Matching | | |
|---|---|---|---|---|---|---|
| **Index** | **Size** | **Time** | **Space** | **Matches** | **Find** | **Locate** |
| GCSA-2 | 67.7 MB | 11 min | 5.9 GB | 388,963 | 12 min | 16 min |
| GCSA-4 | 66.0 MB | 11 min | 5.7 GB | 388,134 | 12 min | 14 min |
| GCSA-8 | 64.7 MB | 11 min | 3.6 GB | 387,696 | 12 min | 13 min |
| RLCSA | 165.0 MB | 4 min | 1.0 GB | 384,400 | 6 min | 7 min |

Table 7.2: Index construction and exact matching with GCSA (sample rate 16) and RLCSA (sample rate 32) for four sequences of human chromosome 18. The number of matches is the number of matching patterns out of 10 million. Times for *locate* include the time used by *find*. GCSA-$k$ denotes GCSA with context length $k$.

As there were relatively few occurrences inside the selected chromosome, most of the time was spent doing *find*. Hence the sample rate that only affects *locate* had little effect on the overall performance. GCSA was 2.0–2.3 times slower than RLCSA. About 1% of the reads matched by GCSA were not matched by RLCSA. Memory requirements for building GCSA were significantly higher than for RLCSA. The differences in query performance between GCSA and RLCSA reflect the fundamental techniques, as the implementations share most of their basic components and design choices. Theoretically GCSA should be about two times slower, as it requires four bit vector operations per character in *find*, while RLCSA uses just two.

To test GCSA in a more complicated algorithm, we implemented BWA-like approximate searching [59] for both GCSA and RLCSA. There are some differences to BWA: i) we return all best matches; ii) we do not use a seed sequence; iii) we have no limits on gaps; and iv) we have to match $O(|P| \log |P|)$ instead of $O(|P|)$ characters to build the lower bound array for pattern $P$, as we have not indexed the reverse sequence. We used context length 4 for GCSA. The results can be seen in Table 7.3. GCSA was consistently about two times slower than RLCSA, while finding from 1.0% (exact matching) to 2.4% (edit distance 3) more matches in addition to those found by RLCSA.

| | **GCSA-4** | | **RLCSA** | |
|---|---|---|---|---|
| **k** | **Matches** | **Time** | **Matches** | **Time** |
| 0 | 388,134 | 14 min | 384,400 | 7 min |
| 1 | 619,927 | 78 min | 609,320 | 39 min |
| 2 | 875,183 | 220 min | 856,373 | 111 min |
| 3 | 1,145,895 | 1,356 min | 1,118,719 | 703 min |

Table 7.3: Approximate matching with GCSA and RLCSA. The reported numbers of matching patterns for a given edit distance $k$ include those found with smaller edit distances.

# Chapter 8

# Conclusions

**Chapter 4.** We showed that compressed suffix arrays can achieve much better compression than predicted by empirical entropy, if the text is highly repetitive. This improved compression does not degrade the query performance of the index significantly. A partial exception is the *locate* query, whose performance depends on the number of suffix array samples. While there has been some success in compressing the samples when a good multiple alignment of a collection of texts is known [69, 40], they are still incompressible in the general case.

When analyzing the proposed index, we used the number of equal letter runs in the Burrows-Wheeler transform as our complexity metric, instead of the usual empirical entropy. We provided evidence that the number of runs scales with the complexity of the text, and hence is a good complexity metric, at least for highly repetitive texts. An interesting open question is, how does the number of runs relate to the number of phrases in a Lempel-Ziv parsing of the text. We get an expected case bound for the number of runs from Theorem 4.2 by treating each phrase as a constant number of edit operations. No worst-case bounds or bounds in the other direction are known, however.

**Chapter 5.** In many applications, the size of data sets increases with the amount of computing power available. One consequence of this is that when using compressed data structures, the space requirements of the construction algorithm often become the bottleneck that determines how large data sets we can process. We showed how to construct a compressed suffix array for a collection of length $N$ in $O(Nm)$ time and $O(N/m \log N)$ bits of extra working space by indexing the collection in $m$ parts. In many cases, this means that we can build the compressed suffix array in $O(n \log n)$ time while using less space than original size of the collection. As the algorithm parallelizes well, it can be used to index data sets of up to tens of gigabytes in size on current hardware.

One variant of the construction algorithm sorts the suffixes of a new subcollection by their lexicographic ranks among the suffixes of already indexed subcollections. This generalizes the well-known technique of packing several consecutive characters into a single machine word to speed up suffix sorting. While a straightforward application of the idea did not yield significant improvements in construction speed, there may be other ways to use the rank information to construct the suffix array faster.

**Chapter 6.** We proposed an extremely space-efficient algorithm for constructing various compressed representations of the longest common prefix array directly from a compressed suffix array. While the algorithm is much slower than the alternatives with larger space requirements, it should parallelize well, making it competitive with space-efficient CSA construction. Yet if we have to construct the LCP array from scratch every time we insert new sequences into the collection, LCP construction will be the bottleneck in maintaining the index. An open question is, can we merge the LCP arrays of two text collections efficiently in a similar way as we merge the CSAs in the space-efficient construction algorithm.

We also showed how to derive any LCP value from a set of sampled values in a similar way as deriving suffix array values from samples in a *locate* query. With regular (not highly repetitive) texts, this new sampled LCP array representation occupies the middle ground in time/space trade-offs, combining reasonable compression and relatively fast access to the LCP values. Yet if most of the minimal LCP values are small, it might be possible to achieve better compression without sacrificing too much performance by computing the small values directly. It remains to be seen, if some combination of sampling and direct LCP computation can provide improved time/space trade-offs.

Many compressed suffix tree proposals are based on combining a compressed suffix array, a compressed representation of the LCP array, and a representation of suffix tree topology. For the first two components, there exist solutions whose size depends on the number of equal letter runs in the Burrows-Wheeler transform, making them attractive for indexing highly repetitive collections. Unfortunately, no known solution for compressing the suffix tree topology combines such compression performance with fast queries. Finding such solution would make compressed suffix trees much more attractive for indexing highly repetitive collections.

**Chapter 7.** We generalized compressed suffix arrays for indexing finite automata. While the index can be exponentially larger than the automaton in the worst case, the size increase will remain small for highly repetitive languages such as those arising from a set of individual genomes. The generalized index is slower than a regular compressed suffix array by a factor of two, due to the use of an additional bit vector to map outgoing edges to nodes. Index construction still requires much more memory than with regular CSAs. It should be possible to adapt most of the algorithms using a CSA to use the generalized index instead.

A major open problem is, whether a similar generalization is possible for grammar-based indexes. Context-free grammars have many advantages over finite automata in describing finite languages. One of the advantages is the ability to express rearrangements: substrings that occur in different positions in different strings of the language.

# References

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[2] Jérémy Barbay, Travis Gagie, Gonzalo Navarro, and Yakov Nekrich. Alphabet partitioning for compressed rank/select and applications. In *Proceedings of the 21st International Symposium on Algorithms and Computation, Part II (ISAAC 2010)*, volume 6507 of *LNCS*, pages 315–326. Springer, 2010.

[3] Markus J. Bauer, Anthony J. Cox, and Giovanna Rosone. Lightweight BWT construction for very large string collections. In *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM 2011)*, volume 6661 of *LNCS*. Springer, 2011.

[4] Timo Beller, Simon Gog, Enno Ohlebusch, and Thomas Schnattinger. Computing the longest common prefix array based on the Burrows-Wheeler transform. In *Proceedings of the 18th Symposium on String Processing and Information Retrieval (SPIRE 2011)*, volume 7024 of *LNCS*, pages 197–208. Springer, 2011.

[5] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. Directly adressable variable-length codes. In *Proceedings of the 16th Symposium on String Processing and Information Retrieval (SPIRE 2009)*, volume 5721 of *LNCS*, pages 122–130. Springer, 2009.

[6] Michael Burrows and David J. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[7] Ho-Leung Chan, Wing-Kai Hon, Tak-Wah Lam, and Kunihiko Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2):21, 2007.

[8] David R. Clark and J. Ian Munro. Efficient suffix trees on secondary storage (extended abstract). In *Proceedings of the seventh annual ACM-SIAM symposium on Discrete algorithms (SODA 1996)*, pages 383–391. SIAM, 1996.

[9] Francisco Claude, Antonio Fariña, Miguel Martínez-Prieto, and Gonzalo Navarro. Compressed q-gram indexing for highly repetitive biological sequences. In *Proceedings of the 10th IEEE International Conference on Bioinformatics and Bioengineering (BIBE 2010)*, pages 86–91. IEEE, 2010.

[10] Francisco Claude and Gonzalo Navarro. Practical rank/select queries over arbitrary sequences. In *Proceedings of the 15th International Symposium on String Processing and Information Retrieval (SPIRE 2008)*, volume 5280 of *LNCS*, pages 176–187. Springer, 2008.

[11] Francisco Claude and Gonzalo Navarro. Self-indexed text compression using straight-line programs. In *Proceedings of the 34th International Symposium on Mathematical Foundations of Computer Science (MFCS 2009)*, volume 5734 of *LNCS*, pages 235–246. Springer, 2009.

[12] Rodrigo Cánovas and Gonzalo Navarro. Practical compressed suffix trees. In *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA 2010)*, volume 6049 of *LNCS*, pages 94–105. Springer, 2010.

[13] Roman Dementiev, Juha Kärkkäinen, Jens Mehnert, and Peter Sanders. Better external memory suffix array construction. *Journal of Experimental Algorithmics*, 12:article 3.4, 2008.

[14] Peter Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[15] Paolo Ferragina, Travis Gagie, and Giovanni Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

[16] Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The engineering of a compression boosting library: theory vs practice in BWT compression. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA 2006)*, volume 4168 of *LNCS*, pages 756–767. Springer, 2006.

[17] Paolo Ferragina, Raffaele Giancarlo, Giovanni Manzini, and Marinella Sciortino. Boosting textual compression in optimal linear time. *Journal of the ACM*, 52(4):688–713, 2005.

[18] Paolo Ferragina, Rodrigo González, Gonzalo Navarro, and Rossano Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithms*, 13:article 1.12, 2009.

[19] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and S. Muthukrishnan. Compressing and indexing labeled trees, with applications. *Journal of the ACM*, 57(1):article 4, 2009.

[20] Paolo Ferragina and Giovanni Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[21] Paolo Ferragina and Giovanni Manzini. On compressing the textual web. In *Proceedings of the third ACM international conference on Web search and data mining (WSDM 2010)*, pages 391–400. ACM, 2010.

[22] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms*, 3(2):20, 2007.

[23] Paolo Ferragina, Jouni Sirén, and Rossano Venturini. Distribution-aware compressed full-text indexes. In *Proceedings of the 19th Annual European Symposium on Algorithms (ESA 2011)*, volume 6942 of *LNCS*, pages 760–771. Springer, 2011.

[24] Paolo Ferragina and Rossano Venturini. A simple storage scheme for strings achieving entropy bounds. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms (SODA 2007)*, pages 690–696. SIAM, 2007.

[25] Johannes Fischer. Wee LCP. *Information Processing Letters*, 110(8–9):317–320, 2010.

[26] Johannes Fischer. Inducing the LCP array. In *Proceedings of the 12th International Symposium on Algorithms and Data Structures (WADS 2011)*, volume 6844 of *LNCS*, pages 374–385. Springer, 2011.

[27] Johannes Fischer, Veli Mäkinen, and Gonzalo Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.

[28] Luca Foschini, Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. When indexing equals compression: Experiments with compressing suffix arrays and applications. *ACM Transactions on Algorithms*, 2(4):611–639, 2006.

[29] Wolfgang Gerlach. Dynamic FM-index for a collection of texts with application to space-efficient construction of the compressed suffix array. Master's thesis, Bielefeld University, 2007.

[30] Simon Gog and Enno Ohlebusch. Fast and lightweight LCP-array construction algorithms. In *Proceedings of the Thirteenth Workshop on Algorithm Engineering and Experiments (ALENEX 2011)*, pages 25–34. SIAM, 2011.

[31] Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithms (SODA 2006)*, pages 368–373. SIAM, 2006.

[32] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: PAT trees and PAT arrays. In William B. Frakes and Ricardo Baeza-Yates, editors, *Information retrieval: data structures and algorithms*, pages 66–82. Prentice-Hall, 1992.

[33] Rodrigo González and Gonzalo Navarro. Rank/select on dynamic compressed sequences and applications. *Theoretical Computer Science*, 410(43):4414–4422, 2009.

[34] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter. High-order entropy-compressed text indexes. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms (SODA 2003)*, pages 841–850. SIAM, 2003.

[35] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *SIAM Journal on Computing*, 35(2):378–407, 2005.

[36] Roberto Grossi, Jeffrey Scott Vitter, and Bojian Xu. Wavelet trees: from theory to practice. In *Proceedings of the First International Conference on Data Compression, Communication and Processing*, pages 210–231. IEEE, 2011.

[37] Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Compressed data structures: dictionaries and data-aware measures. *Theoretical Computer Science*, 387(3):313–331, 2007.

[38] Wing-Kai Hon, Tak-Wah Lam, Kunihiko Sadakane, Wing-Kin Sung, and Siu-Ming Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. *Algorithmica*, 48(1):23–36, 2007.

[39] Wing-Kai Hon, Kunihiko Sadakane, and Wing-Kin Sung. Breaking a time-and-space barrier in constructing full-text indices. *SIAM Journal on Computing*, 38(6):2162–2178, 2009.

[40] Songbo Huang, T.W. Lam, W.K. Sung, S.L. Tam, and S.M. Yiu. Indexing similar DNA sequences. In *Proceedings of the The Sixth International Conference on Algorithmic Aspects in Information and Management (AAIM 2010)*, volume 6124 of *LNCS*, pages 180–190. Springer, 2010.

[41] David A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[42] Guy Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual IEEE Symposium on Foundations of Computer Science (FOCS 1989)*, pages 549–554. IEEE, 1989.

[43] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching (CPM 2001)*, volume 2089 of *LNCS*, pages 181–192. Springer, 2001.

[44] Dmitry Khmelev. Program lcp version 0.1.9. `http://www.math.toronto.edu/dkhmelev/PROGS/misc/lcp-eng.html`, 2004.

[45] Sebastian Kreft and Gonzalo Navarro. LZ77-like compression with fast random access. In *Proceedings of the 2010 IEEE Data Compression Conference (DCC 2010)*, pages 239–248. IEEE, 2010.

[46] Sebastian Kreft and Gonzalo Navarro. Self-indexing based on LZ77. In *Proceedings of the 22nd Annual Symposium on Combinatorial Pattern Matching (CPM 2011)*, volume 6661 of *LNCS*, pages 41–54. Spr, 2011.

[47] Stefan Kurtz. Reducing the space requirement of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.

[48] Shanika Kuruppu, Simon J. Puglisi, and Justin Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In

*Proceedings of the 17th Symposium on String Processing and Information Retrieval (SPIRE 2010)*, volume 6393 of *LNCS*, pages 201–206. Springer, 2010.

[49] Juha Kärkkäinen. Fast BWT in small space by blockwise suffix sorting. *Theoretical Computer Science*, 387(3):249–257, 2007.

[50] Juha Kärkkäinen, Giovanni Manzini, and Simon Puglisi. Permuted longest-common-prefix array. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM 2009)*, volume 5577 of *LNCS*, pages 181–192. Springer, 2009.

[51] Juha Kärkkäinen and Simon J. Puglisi. Fixed block compression boosting in FM-indexes. In *Proceedings of the 18th Symposium on String Processing and Information Retrieval (SPIRE 2011)*, volume 7024 of *LNCS*, pages 174–184. Springer, 2011.

[52] Juha Kärkkäinen and S. Srinivasa Rao. Full-text indexes in external memory. In *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, chapter 7, pages 149–170. Springer, 2003.

[53] T. W. Lam, W. K. Sung, S. L. Tam, C. K. Wong, and S. M. Yiu. Compressed indexing and local alignment of DNA. *Bioinformatics*, 24(6):791–797, 2008.

[54] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

[55] N. Jesper Larsson and Alistair Moffat. Offline dictionary-based compression. In *Proceedings of the 1999 IEEE Data Compression Conference (DCC 1999)*, pages 296–305. IEEE, 1999.

[56] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. *Theoretical Computer Science*, 387(3):258–272, 2007.

[57] Sunho Lee and Kunsoo Park. Dynamic compressed representation of texts with rank/select. *Journal of Computing Science and Engineering*, 3(1):15–26, 2009.

[58] Sunho Lee and Kunsoo Park. Dynamic rank/select structures with applications to run-length encoded texts. *Theoretical Computer Science*, 410(43):4402–4413, 2009.

[59] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[60] Heng Li and Richard Durbin. Fast and accurate long-read alignment with Burrows-Wheeler transform. *Bioinformatics*, 26(5):589–595, 2010.

[61] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009.

[62] Chi-Ming Liu, Tak-Wah Lam, Thomas Wong, Edward Wu, Siu-Ming Yiu, Zhiheng Li, Ruibang Luo, Bingqiang Wang, Chang Yu, Xiaowen Chu, Kaiyong Zhao, and Rui. SOAP3: GPU-based compressed indexing and ultra-fast parallel alignment of short reads. In *Third Workshop on Massive Data Algorithms (MASSIVE 2011)*, 2011.

[63] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[64] Giovanni Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[65] Edward M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[66] J. Ian Munro. Tables. In *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 1996)*, volume 1180 of *LNCS*. Springer, 1996.

[67] Veli Mäkinen and Gonzalo Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

[68] Veli Mäkinen and Gonzalo Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):32, 2008.

[69] Veli Mäkinen, Gonzalo Navarro, Jouni Sirén, and Niko Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

[70] Joong Chae Na and Kunsoo Park. Alphabet-independent linear-time construction of compressed suffix arrays using o(nlogn)-bit working space. *Theoretical Computer Science*, 385(1-3):127–136, 2007.

[71] Gonzalo Navarro. Indexing text using the Ziv-Lempel trie. *Journal of Discrete Algorithms*, 2(1):87–114, 2004.

[72] Gonzalo Navarro and Veli Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):2, 2007.

[73] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Proceedings of the 2009 IEEE Data Compression Conference (DCC 2009)*, pages 193–202. IEEE, 2009.

[74] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear time suffix array construction using D-critical substrings. In *Proceedings of the 20th Annual Symposium on Combinatorial Pattern Matching (CPM 2009)*, volume 5577 of *LNCS*, pages 54–67. Springer, 2009.

[75] Enno Ohlebusch, Johannes Fischer, and Simon Gog. CST++. In *Proceedings of the 17th Symposium on String Processing and Information Retrieval (SPIRE 2010)*, volume 6393 of *LNCS*, pages 322–333. Springer, 2010.

[76] Enno Ohlebusch and Simon Gog. A compressed enhanced suffix array supporting fast string matching. In *Proceedings of the 16th Symposium on String Processing and Information Retrieval (SPIRE 2009)*, volume 5721 of *LNCS*, pages 51–62. Springer, 2009.

[77] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*, pages 60–70. SIAM, 2007.

[78] Daisuke Okanohara and Kunihiko Sadakane. A linear-time Burrows-Wheeler transform using induced sorting. In *Proceedings of the 16th Symposium on String Processing and Information Retrieval (SPIRE 2009)*, volume 5721 of *LNCS*, pages 90–101. Springer, 2009.

[79] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):4, 2007.

[80] Simon J. Puglisi and Andrew Turpin. Space-time tradeoffs for longest-common-prefix array computation. In *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC 2008)*, volume 5369 of *LNCS*, pages 124–135. Springer, 2008.

[81] Rajeev Raman, Venkatesh Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding $k$-ary trees and multisets. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA 2002)*, pages 233–242. SIAM, 2002.

[82] Luís M. S. Russo and Arlindo L. Oliveira. A compressed self-index using a Ziv-Lempel dictionary. *Information Retrieval*, 11(4):359–388, 2008.

[83] Kunihiko Sadakane. Succinct representations of lcp information and improvements in the compressed suffix arrays. In *Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms (SODA 2002)*, pages 225–232. SIAM, 2002.

[84] Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48:294–313, 2003.

[85] Kunihiko Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[86] Mikaël Salson, Thierry Lecroq, Martine Léonard, and Laurent Mouchard. A four-stage algorithm for updating a Burrows-Wheeler transform. *Theoretical Computer Science*, 410(43):4350–4359, 2009.

[87] Mikaël Salson, Thierry Lecroq, Martine Léonard, and Laurent Mouchard. Dynamic extended suffix arrays. *Journal of Discrete Algorithms*, 8(2):241–257, 2010.

[88] Jouni Sirén. Compressed suffix arrays for massive data. In *Proceedings of the 16th Symposium on String Processing and Information Retrieval (SPIRE 2009)*, volume 5721 of *LNCS*, pages 63–74. Springer, 2009.

[89] Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing finite language representation of population genotypes, 2011. arXiv:1010.2656v4.

[90] Jouni Sirén, Niko Välimäki, Veli Mäkinen, and Gonzalo Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proceedings of the 15th Symposium on String Processing and Information Retrieval (SPIRE 2008)*, volume 5280 of *LNCS*, pages 164–175. Springer, 2008.

[91] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[92] Peter Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory (FOCS 1973)*, pages 1–11. IEEE, 1973.

# Glossary

## Abbreviations

| | |
|---|---|
| AFFM | Alphabet-friendly FM-index |
| BWT | Burrows-Wheeler transform |
| CSA | Compressed suffix array |
| CST | Compressed suffix tree |
| LCP | Longest common prefix (array) |
| GCSA | Generalized compressed suffix array |
| PLCP | Permuted longest common prefix (array) |
| RLCSA | Run-length compressed suffix array |
| RLE | Run-length encoding |
| RLFM | Run-length FM-index |
| SA | Suffix array |
| Sad-CSA | Compressed suffix array of Sadakane |
| SSA | Succinct suffix array |
| SSA-RRR | A CSA with implicit compression boosting |
| ST | Suffix tree |
| WT | Wavelet tree |
| XBW | Extended Burrows-Wheeler transform for trees |

## Functions

| | |
|---|---|
| $\mathrm{char}(i)$ | Character $T[\mathsf{SA}[i]]$ |
| $\mathrm{gap}(B)$ | Complexity metric for gap encoding of $B$ |
| $\mathrm{lcp}(A, B)$ | Length of the longest common prefix of $A$ and $B$ |
| $\mathrm{popcount}(B)$ | Number of 1-bits in binary sequence $B$ |
| $\mathrm{rank}_c(S, i)$ | Number of occurrences of character $c$ in $S[1, i]$ |
| $\mathrm{rank}(T, S)$ | Lexicographic rank of $S$ among the suffixes of $T$ |
| $\mathrm{run}(B)$ | Complexity metric for run-length encoding of $B$ |
| $\mathrm{select}_c(S, i)$ | Position of the $i$th occurrence character $c$ in $S$ |

## Notation

| | |
|---|---|
| $A$ | Finite automaton; $A = (V, E)$ |
| $\mathcal{A}$ | Multiple alignment of sequences |
| $B$ | Binary string, bit vector |
| $B_s$ | Bit vector marking sampled suffix array positions |
| $B_L$ | Bit vector representation of the PLCP array |
| $\mathsf{BWT}$ | Burrows-Wheeler transform of a text |
| $C[c]$ | Number of occurrences of characters $c' < c$ in the text |
| $C_c$ | Range $[C[c]+1, C[c+1]]$, typically of $\mathsf{SA}$ or $\mathsf{BWT}$ |
| $\mathcal{C}$ | Collection of texts |
| $E$ | Set of edges of a graph |
| $F$ | Bit vector delimiting nodes in GCSA and XBW |
| $G$ | Graph; $G = (V, E)$ |
| $H$ | Complexity metric |
| $H_k$ | Order-$k$ empirical entropy |
| $L$ | Formal language |
| $L(A)$ | Language recognized by automaton $A$ |
| $\mathsf{LCP}$ | Longest common prefix array of a text |
| $LF$ | A function such that $\mathsf{SA}[LF(i)] = \mathsf{SA}[i] - 1$; the inverse of $\Psi$ |
| $N$ | Total length of a collection of texts |
| $M$ | Bit vector encoding the outgoing edges in GCSA |
| $P$ | Pattern |
| $\mathsf{P}$ | Path in a graph |
| $\mathsf{PLCP}$ | LCP array in text order |
| $R$ | Number or equal letter runs in $\mathsf{BWT}$ |
| $\mathsf{RA}$ | Rank array of a text relative to another text |
| $S$ | String / sequence |
| $\mathsf{SA}$ | Suffix array of a text |
| $T$ | Text string terminated by end marker \$ |
| $V$ | Set of nodes of a graph |
| $b$ | Block size in bits |
| $b(i)$ | Binary representation of integer $i$ |
| $c$ | Character |
| $d$ | Suffix array sample rate |
| $d'$ | LCP or PLCP array sample rate |
| $e$ | Edge of a graph |
| $f, g$ | Functions |
| $i, j, k, l$ | Non-negative integers |
| $\ell$ | Label |

| | |
|---|---|
| $m$ | Number of collections |
| $n$ | Length of input string |
| $n_1$ | Number of 1-bits |
| $p$ | Probability |
| $r$ | Number of texts in a collection |
| $s$ | Number of mutations |
| $u, v, w$ | Nodes of a graph |
| $sp, ep$ | Starting and ending points of a suffix array range |
| $t_B$ | Time complexity of one step of backward searching |
| $t_{LF}$ | Time complexity of computing $LF$ |
| $t_R$ | Time complexity of computing *rank* |
| $t_S$ | Time complexity of computing *select* |
| $t_U$ | Time complexity updating a bit vector |
| $t_\Psi$ | Time complexity of computing $\Psi$ |
| $\lambda$ | Empty string of length 0 |
| $\Sigma$ | Alphabet |
| $\sigma$ | Size of the alphabet |
| $\Psi$ | A function such that $\mathsf{SA}[\Psi(i)] = \mathsf{SA}[i] + 1$; the inverse of $LF$ |
| $\$$ | End marker of a text string; lexicographic value 0 |
| $\#$ | First character of a formal language; lexicographic value $\sigma + 1$ |