Università degli Studi di Udine

Dipartimento di Scienze Matematiche, Informatiche e Fisiche

Dottorato di Ricerca in Informatica e Scienze Matematiche e Fisiche

Ph.D. Thesis

Compressed Computation for Text Indexing

Candidate:
Nicola Prezza

Supervisor:
Prof. Alberto Policriti

Cycle XXIX — Academic Year 2016

Author's e-mail: prezza.nicola@spes.uniud.it

Author's address:

Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine
Italia

Acknowledgments

First of all, I would like to thank my parents for supporting me through all the long path that went from by bachelor to this PhD. Thanks for agreeing with my choices without questioning (well, not too much) why I was deciding to spend three more years at the university!

Special thanks go also to my supervisor Alberto Policriti for many stimulating discussions and for believing in me, starting from that time I asked him to assign me a bachelor thesis on Java graphical interfaces because I was running out of time for graduating (by the way, he did not give me that Java thesis. He instead assigned me an interesting problem related to hashing, which resulted—after five years—in *this* thesis).

Thanks to many colleagues for extremely stimulating discussions on algorithms, data structures, and compression: Simon Puglisi, Travis Gagie, Simon Gog, Djamal Belazzougui, Gabriele Fici, Nicola Gigante, Fabio Cunial, Jouni Sirén, and many others (sorry for not reporting all the names here, this thesis was already too long).

To conclude, thanks to all my friends here in Udine (and surroundings) for all the great times we had together in these years. Without you this thesis would have been two times longer, and nobody would have read (and accepted to review) it.

Abstract

This thesis deals with space-efficient algorithms to *compress* and *index* texts. The aim of compression is to reduce the size of a text by exploiting regularities such as repetitions or skewed character distributions. Indexing, on the other hand, aims at accelerating pattern matching queries (such as locating all occurrences of a pattern) with the help of data structures (indexes) on the text. Despite being apparently distant concepts, compression and indexing are deeply interconnected: both exploit the inner structure of the text to, respectively, reduce its size and speed up pattern matching queries. It should not be surprising, therefore, that compression and indexing can be "forced" (actually, quite naturally) to coincide: *compressed full-text indexes* support fast pattern matching queries while taking almost the same size of the compressed text.

In the last two decades, several compressed text indexes based on the most efficient text compressors have been designed. These indexes can be roughly classified in two main categories: those based on suffix-sorting (Burrows-Wheeler transform indexes, compressed suffix arrays/trees) and those based on the replacement of repetitions by pointers (Lempel-Ziv indexes, grammar indexes, block trees). Indexes based on a combination of the two techniques have also been proposed. In general, suffix sorting-based methods support very fast queries at the expense of space usage. This is due to several factors, ranging from weak compression methods (e.g. entropy compression, used in early FM-indexes, is not able to capture long repetitions), to heavy structures (e.g. suffix array sampling) flanked to the compressed text representation to speed up queries. The second class of indexes, on the other hand, offers strong compression rates, achieving up to *exponential* compression on very repetitive texts at the expense of query times—often quadratic in the pattern length or—in the worst case—linear in the text length.

Among the most used compression techniques, run-length compression of the Burrows-Wheeler transform and Lempel-Ziv parsing (LZ77) have been proved to be superior in the compression of very repetitive datasets. In this thesis we show that these two tools can be combined in a single index gathering the best features of the above-discussed indexes: fast queries (linear in the pattern length and logarithmic in the text length), and strong compression rates (up to exponential compression can be achieved). We describe an efficient implementation of our index and compare it with state of the art alternatives. Our solution turns out to be as space-efficient as the lightest index described in the literature while supporting queries up to three orders of magnitude faster.

Apart from index definition and design, a third concern regards index construction complexity. Often, the input text is too big to be fully loaded into main memory. Even when this is feasible, classic compression/indexing algorithms use heavy data structures such as suffix trees/arrays which can easily take several times the space of the text. This is unsatisfactory, especially in cases where (i) the text is streamed and not stored anywhere (e.g. because of its size) and (ii) the compressed text is small enough to fit into main memory. A further contribution of this thesis consists in five algorithms compressing text within compressed working space and in two *recompression* techniques (i.e. algorithms to

convert between different compression formats without full decompression). The complete picture we offer consists of a set of algorithms to space-efficiently convert among:

- the plain text
- two compressed self-indexes, and
- three compressed-file formats (entropy, LZ77, and run-length BWT)

The general idea behind all our compression algorithms is to read text characters from left to right and build a compressed dynamic Burrows-Wheeler transform of the reversed text. This structure is augmented with a dynamic suffix array sampling to support fast locate of text substrings. We employ three types of suffix array sampling: (i) evenly-spaced (ii) based on Burrows-Wheeler transform equal-letter runs, and (iii) based on Lempel-Ziv factors. Strategy (i) allows us to achieve entropy-compressed working space. Strategies (ii) and (iii) are novel and allow achieving a space usage proportional to the output size (i.e. the compressed file/index).

As a last contribution of this thesis, we turn our attention to a practical and usable implementation of our suite of algorithmic tools. We introduce `DYNAMIC`, an open-source C++11 library implementing dynamic compressed data-structures. We prove almost-optimal theoretical bounds for the resources used by our structures, and show that our theoretical predictions are empirically tightly verified in practice. The implementation of the compression algorithms described in this thesis using `DYNAMIC` meets our expectations: on repetitive datasets our solutions turn out to be up to three orders of magnitude more space-efficient than state-of-the art algorithms performing the same tasks.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The term *big data* is increasingly being used in many technological areas at an ever-accelerating pace. With this term one usually indicates the production and analysis of datasets so big that even their storage is a challenging task with current technologies. Numerous examples come from different domains: massive DNA sequencing projects (population genomics), particle accelerators (e.g. CERN), and web databases such as Wikipedia or GitHub face the problem of manipulating, filtering and storing huge amounts of data whose sizes quickly increases at a daily rate.

In many cases, such datasets are extremely repetitive and can therefore be considerably reduced in size through compression. At the same time, it is crucial to pre-process this data in order to speed up subsequent queries—such as searches—on it. The latter requirement can be met by using so-called *full-text indexes*, i.e. data structures supporting search queries in much less time than that required to scan the whole dataset. Although standard full-text indexes are able to support search queries on the text in optimal time, they are too big in size and fail in exploiting the compressibility of the dataset. Both requirements—compression and indexing—can however be satisfied by means of recent breakthroughs in the field of *compressed full-text indexing*. A compressed full-text index is a data structure whose size is close to that of the compressed text and that supports fast search functionalities (e.g. exact/approximate pattern matching) on it.

Given the importance of compression in domains where big data is predominant, much research is lately focusing towards the study of methods for efficiently compressing data and working directly on the compressed files. The term *compressed computation* indicates any technique that operates on the compressed data without first (fully) decompressing it. Results in this field are extremely powerful as they often show that computation and compression are deeply interconnected: in a sense, by getting rid of redundancy in our data, we can perform much more efficiently operations on it—compressed text indexes are an illuminating example of this phenomena. The ability of operating directly on the compressed data carries enormous advantages. First of all, we can manipulate much larger datasets: as we will empirically show, repetitive text collections such as software repositories and sets of same-species genomes, can often be reduced in size by thousands of times with techniques such as Lempel-Ziv factorization (LZ77) and run-length encoded Burrows-Wheeler transform (RLBWT). Second, the possibility of fitting the compressed dataset into main memory translates to much higher processing speeds with respect to external-memory algorithms.

Compression and indexing, however, come at a significant computational cost. Standard methods to compress and index data often require to load the full text into main memory or, even worse, to build heavy data structures such as suffix trees or suffix arrays. Due to the huge amounts of data that need to be processed, this is often unfeasible. One solution to this problem is to use offline (semi-) external algorithms that save space in RAM by increasing the number of accesses to disk. While this is a good solution in most domains (being fast also in practice), it requires the data to be completely available on disk. This is not always the case: in applications such as particle-collision experiments or massive DNA sequencing projects, data is streamed from the source and heavily filtered prior to storage (it is not even feasible to store all of it on disk). A second drawback of off-line solutions is that they are inherently *static*: in the worst case, adding new data to the archive could require scanning the whole dataset and re-compressing it. This is often unfeasible in terms of time and space requirements.

A second solution is that of compressing data on-the-fly (on-line / real-time compression). This concept is equivalent to the idea of maintaining a dynamic *and* compressed archive, i.e. a compressed file that can be efficiently updated by appending new data without first (fully) decompressing it. In this domain, the term *efficiently* means that single updates should be supported in time polylogarithmic with respect to the dataset size. Crucially, the file should never be decompressed during updates: full decompression would increase update times to linear (w.r.t. the dataset size) and could be prohibitive in space.

The first contribution of this thesis (Chapters 3 and 4) is to show that data can be compressed in the streaming model: characters are read left-to-right and previous parts of the plain text are no longer accessible. We provide algorithms that compress text using the Burrows-Wheeler transform (Chapter 3) and Lempel-Ziv (Chapter 4) compression schemes within *compressed working space* under different measures of compressibility: zero-order and high-order entropy, the number of runs in the Burrows-Wheeler transform, and the size of the Lempel-Ziv parsing. Our solutions are based on dynamic compressed data structures; such structures can be seen as dynamic compressed archives that can be efficiently updated by single-character extensions.

All the theoretical results discussed have been implemented and extensively tested on real datasets. The implementations have been carried out using a tool introduced in Chapter 6 and called `DYNAMIC` [97]: a C++ library offering efficient implementations of several *dynamic* compressed data structures. While several excellent *static* data structures libraries are available on the web (see, for example, [15, 43, 95, 96, 120]), little work has been done on the dynamic side. We believe our library represents a significant first step in this direction.

After compression, one is usually interested in accessing and manipulating the compressed data. Also in this step we are faced with the same problems discussed above: we cannot afford decompressing the whole dataset in order to perform queries on it or to build a data structure, as this would be too space- and time-consuming. Even though algorithms working directly on compressed representations do exist (e.g. indexing algorithms), they often take as input only one type of compressed representation (e.g. only LZ77 *or* RLBWT). The second problem we tackle is therefore that of *recompression*, i.e. converting between different compression formats within compressed working space. In Section 5.2 we show that it is possible to convert between LZ77 and RLBWT formats in a working space proportional to the sizes of the input and the output.

The last problem we tackle is that of indexing repetitive text collections. Existing indexes for this problem are based on three main compressors: run-length compressed suffix arrays [77,111] (RLCSAs), LZ77 [65,66,67], and straight-line programs [18,19,20,115] (SLPs: context-free grammars generating only the text). RLCSAs suffer from the problem of the suffix array sampling, which introduces a space-time tradeoff: by sampling the suffix array every k positions, the space of this component is n/k words but locate times are multiplied by a factor of k. LZ77 indexes are very space-efficient, but suffer from slow search times (extracting a character from LZ77 could require—in the worst case—to scan the whole text). Grammar indexes can achieve better compression rates than RLCSAs at the expenses of query times; they are however inferior to LZ77 compression, and hard to optimize (computing the smallest grammar is an NP-hard problem, see Section 5.1). See [16] for a comprehensive experimental evaluation of all these classes of indexes.

In this thesis we propose two compressed full-text indexes for highly repetitive texts solving some of the problems raised above. The first index, dubbed `s-rlbwt` (*sparse RLBWT*: Section 5.3.2), is an optimal-space run-length FM index significantly reducing the space of the state-of-the-art `rlcsa` index [111] (run-length compressed suffix array). The optimal space usage is achieved by employing a novel sparsification technique in the representation of the run-length encoded Burrows-Wheeler transform: this is the main difference between `s-rlbwt` and the RLFMI index described by Mäkinen et al. [77]. As for the `rlcsa`, the weak point of this index is the suffix array sampling. Our second index, dubbed `slz-rlbwt` (*sparse Lempel-Ziv RLBWT*: Section 5.3.3), solves this problem by combining (a sparse version of) LZ77 and run-length encoding of the BWT. We show that we can sample the suffix array only at the end of LZ77 phrases while still being able to quickly locate pattern occurrences. By using the results discussed in Chapters 3 and 4 and in Section 5.2, we moreover show how to build our indexes using (overall) compressed working space during construction. Efficient implementations of our indexes are available at [101, 102]. Our implementations have been tested on highly repetitive datasets generated by downloading versioned archives from different sources: Wikipedia web pages, GitHub software repositories, and repetitive genomic collections. The scripts used to generate such datasets are available at [99, 103].

The diagram in Figure 1.1 shows the relationships among all the results presented in this thesis. We will re-propose this diagram throughout the thesis to contextualize the presented results in our framework of algorithmic tools.

1.2 Contributions

This thesis contains material from six papers, in addition to several unpublished ideas. The papers will be referred to as papers `(i)-(vi)` throughout the thesis. Five out of six papers (`(i)-(iii)`,`(v)`,`(vi)`) have been published in leading conferences in the field. Paper `(iv)` has just been accepted for publication in the journal *Algorithmica*.

1.2.1 Papers

(i) Alberto Policriti, Nicola Gigante, and Nicola Prezza. **Average linear time and compressed space construction of the Burrows-Wheeler transform**. In International Conference on Language and Automata Theory and Applications (LATA 2015), pp. 587-598. Springer International Publishing, 2015.

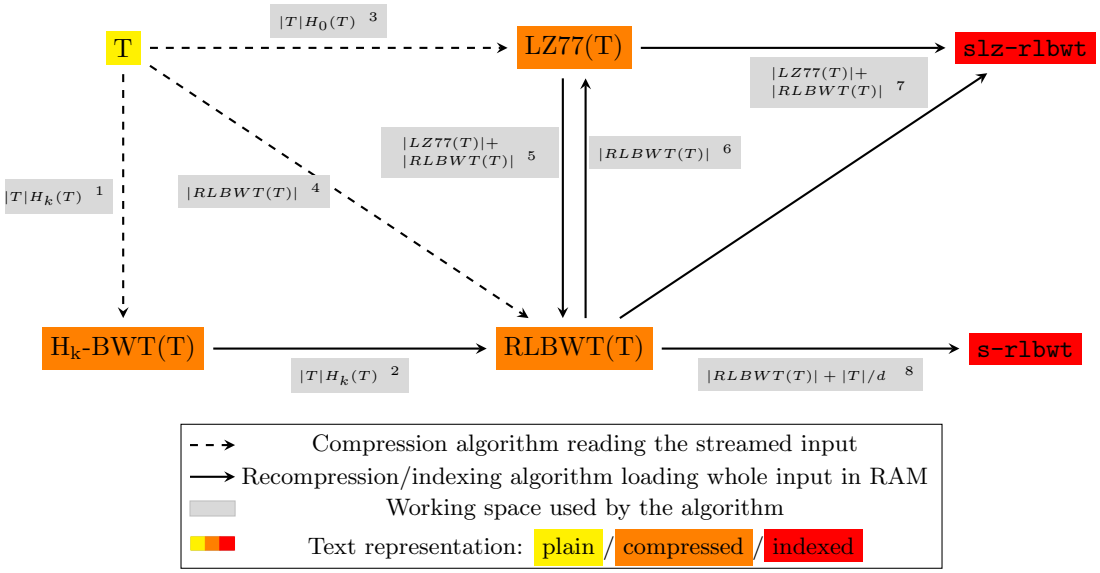


Figure 1.1: The diagram links together all the contributions of this thesis. Vertexes are labeled with file representations: T is the original text, H_k-BWT(T) is a high-order compressed BWT of T, LZ77(T) is the LZ77 parsing of T, RLBWT(T) is a run-length compressed BWT of T, and `slz-rlbwt` and `s-rlbwt` are our indexes for highly repetitive text collections. Edges represent our algorithms converting between different file representations, and are oriented from the input to the output. Dashed edges indicate that the input is streamed character-by-character from the source (e.g. disk); in particular, the input is not fully loaded in RAM. Solid edges indicate that the input needs to be fully available (and loaded in RAM) in order to be processed. Edges are labeled with the RAM working space used by our construction algorithms. Some more detail:

1- `cw-bwt` algorithm (Section 3.2) builds a high-order compressed BWT
2- We can extract characters left-to-right from our high-order compressed BWT representation and easily stream to the output a run-length BWT
3- `h0-lz77` algorithm (Section 4.2) builds online the LZ77 factorization of the text in zero-order compressed space
4- `rle-bwt` algorithm (Section 3.3) builds online a run-length compressed BWT (of the reversed text) in a working space proportional to the output
5. In Section 5.2.3 we show how to convert LZ77 to RLBWT using a working space proportional to the input and output
6. In Section 5.2.2 we show how to convert RLBWT to LZ77 using a working space proportional to the input (output is streamed to disk)
7- `slz-rlbwt` index (Section 5.3.3) combines RLBWT and LZ77 and can be built in asymptotically optimal working space taking as input both these representations
8- `s-rlbwt` index (Section 5.3.2) combines RLBWT and a sparse suffix array sampled every d text positions. The index can be built in asymptotically optimal working space taking as input $RLBWT(T)$

Note that the diagram implies that both our indexes can be built in asymptotically optimal working space by processing the streamed text with the pipeline $T \xrightarrow{4} RLBWT(T) \xrightarrow{6} LZ77(T)$ and using these two compressed representations to build the indexes.

(ii) Alberto Policriti and Nicola Prezza. **Fast online Lempel-Ziv factorization in compressed space**. In International Symposium on String Processing and Information Retrieval (SPIRE 2015), pp. 13-20. Springer International Publishing, 2015.

(iii) Alberto Policriti and Nicola Prezza. **Computing LZ77 in Run-Compressed Space**. In Data Compression Conference (DCC 2016). IEEE, 2016.

(iv) Alberto Policriti and Nicola Prezza. **LZ77 Computation Based on the Run-Length Encoded BWT**. Algorithmica, 2017 (accepted).

(v) Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. **Composite repetition-aware data structures**. In Annual Symposium on Combinatorial Pattern Matching (CPM 2015), pp. 26-39. Springer International Publishing, 2015.

(vi) Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. **Flexible Indexing of Repetitive Collections**. In Computability in Europe (CiE 2017). Springer International Publishing, 2017 (to appear).

1.2.2 Unpublished Results

The thesis contains also original results that have not been published elsewhere:

- The algorithms to convert between LZ77 and RLBWT formats in compressed working space (Section 5.2)
- Detailed description, theoretical analysis, and extensive benchmarks of the dynamic compressed data structures implemented in the `DYNAMIC` C++ library. This material is contained in Chapters 6 and 7.

1.3 Outline

The first chapter of the thesis (Chapter 2) starts with a zero-knowledge introduction to the field of compressed text indexing. The following sections are devoted to definitions and the introduction of state-of-the-art tools used in the next chapters. In writing this manuscript, one of the goals was to make it as self-contained as possible. For this reason, Chapter 2 introduces the notion of compression and describes data structures such as succinct/compressed bitvectors, wavelet trees, and geometric data structures. These results are used at the end of the chapter to introduce several compressed text indexes based on the Burrows-Wheeler Transform and on Lempel-Ziv parsing, and to describe the online BWT construction algorithm standing at the core of most of the results discussed in this thesis.

The original contributions of this thesis start with Chapter 3. This chapter and Chapter 4 describe five original algorithms (papers `(i)-(iv)`) to compute the Burrows-Wheeler transform and the LZ77 factorization in compressed working space. Both chapters are divided in three main sections: illustration of state of the art, algorithms working in entropy-compressed working space, and algorithms working in run-length compressed working space. Algorithms from the first class are suited to compress texts with skewed character

distributions, while algorithms from the second class perform very well on highly repetitive datasets.

While Chapters 3 and 4 deal with the problem of turning an uncompressed text representation into a compressed one, Chapter 5 is devoted to the problem of manipulating compressed text representations only. This field of research takes the name of *compressed computation* (i.e. computation on compressed data without fully decompressing it). In this chapter we show how to *recompress* (i.e. changing compression format) compressed files, and how to use such representations to build compressed full-text indexes. All computations require compressed working space. In detail, we show how to convert between LZ77 and RLBWT formats (which can be computed with the algorithms of Chapters 3 and 4), and how to build two self-indexes based on these compression techniques. These indexes can be built in compressed working space by using our compression algorithms and/or our recompression techniques.

Chapter 6 describes our `DYNAMIC` C++ library. The library implements several results discussed in this thesis, including succinct/compressed dynamic strings, searchable partial sums with inserts, dynamic indexes, and all compression algorithms described in Chapters 3 and 4. We include a theoretical analysis of all data structures implemented in the library.

In Chapter 7 we describe how we compared our findings with state of the art alternatives and present experimental results on highly compressible datasets. The chapter is divided in a first section dedicated to the empirical evaluation of our `DYNAMIC` library and in a second section reporting results of the experiments involving compression algorithms and compressed indexes.

1.4 Notation

Strings

We work with strings $S \in \Sigma^n$ of length $|S| = n$ on the alphabet $\Sigma = \{0, \ldots, \sigma - 1\}$. The only restriction we impose on the alphabet size is $\sigma \leq n$ (which is always true after a re-mapping of the symbols). If not specified otherwise, space of the data structures will be given in bits. Concatenation of strings $u, v \in \Sigma^*$ will be denoted by uv. By c^k we will denote the string "$cc...c$" (k times). By $S_i,\ i \leq n$, we will denote $S[i, ..., n-1]$, i.e. the i-th suffix of S. With the term *k-context* we will denote a k-mer on Σ, i.e. a string in Σ^k. The notation $\overleftarrow{S}$ indicates the reverse of the string $S \in \Sigma^*$. An equal-letter run in a string S is a maximal substring a^k, with $k > 0$ and $a \in \Sigma$. With the notation r_S we will indicate the number of equal-letter runs in the string S. A substring V of a string $S \in \Sigma^*$ is *right-maximal* if there exist two distinct characters $a \neq b,\ a, b \in \Sigma$ such that both Va and Vb are substrings of S. $S[i]$ is the i-th character of S. When dealing with dynamic strings, by $S[i] \leftarrow c$ we will denote character substitution in position i of the string. The operation $S.insert(c, i)$ will denote character insertion, turning S into $S[0, ..., i-1]cS[i, ..., |S|-1]$.

Logarithms are in base 2 if not otherwise specified.

Burrows-Wheeler Transform

We introduce some notation that will be used when dealing with the *Burrows-Wheeler transform* (BWT). See Section 2.2.2 for a formal definition of BWT.

When dealing with the Burrows-Wheeler transform of a string T, we will assume that T ends with a character \$ lexicographically smaller than all other alphabet characters and not appearing elsewhere in T. $BWT(T)$ will denote the Burrows-Wheeler transform of string T, and, when clear from the context, we will refer to it simply as BWT. With T-, L- and F-*positions* we will denote positions on the text T and on the L (last) and F (first) column of the BWT matrix, respectively. All BWT intervals are inclusive, and we denote them as $[l, r]$ (left-right positions on the BWT). $BWT.F(c),\ c \in \Sigma$, will denote the starting F-position of the block corresponding to character c in the BWT matrix. Letting $W \in \Sigma^*$, the *interval of* W will be the interval $[l, r]$ of rows prefixed by W in the BWT matrix ($r < l$ if W does not occur in T). $BWT.LF(i), 0 \leq i < |BWT|$ denotes LF function applied to L-position i. With $BWT.LF([l, r], c), 0 \leq l, r < |BWT|,\ c \in \Sigma \cup \{\$\}$, we will denote function LF applied to BWT intervals: if $[l, r]$ is the interval of a string $W \in \Sigma^*$ in BWT, then $BWT.LF([l, r], c)$ returns the interval $[l', r']$ of cW in BWT. Using the notation above introduced, $r_{BWT(T)}$ denotes the number of runs in the Burrows-Wheeler transform of T. In this work we will deal with both measures $r_{BWT(T)}$ and $r_{BWT(\overleftarrow{T})}$. In order to simplify notation, we will use the symbol r to indicate the quantity

$$r = \max\{r_{BWT(T)}, r_{BWT(\overleftarrow{T})}\}$$

In real-case texts, $r_{BWT(T)} \approx r_{BWT(\overleftarrow{T})}$ holds, so this simplification has also a practical justification.

A run-length encoded representation of $BWT(T)$—to be denoted as $RLBWT(T)$ or simply $RLBWT$—is defined as follows:

Definition 1. ***Run-length encoded Burrows-Wheeler transform.*** *$RLBWT$ is a sequence of pairs*

$$RLBWT(T) = \langle \lambda_i, c_i \rangle_{i=1,\ldots,r}$$

where $\lambda_i > 0$ is the length of the maximal *i-th c_i-run, $c_i \in \Sigma$. Equivalently, $RLBWT(T)$ is the* shortest *list of pairs $\langle \lambda_i, c_i \rangle_{i=1,\ldots,r}$ satisfying $BWT(T) = c_1^{\lambda_1} c_2^{\lambda_2} \ldots c_r^{\lambda_r}$*

In some cases we will need to distinguish between $RLBWT(T)$ seen as a plain list of pairs (see above) and as a RLBWT *data structure*. In such cases, with $RLBWT^+(T)$ we will denote a run-length encoded BWT *data structure* on the text T, taking $\mathcal{O}(r)$ words of space and supporting access to the i-th BWT character, functions LF, FL (mapping L-positions to F-positions and the other way round), and left-extension of the text in $\mathcal{O}(\log r)$ time. In Section 3.3 we describe how to build such a data structure.

Lempel-Ziv Factorization

The *LZ77 parsing* (or *factorization*) of a text T is defined as

Definition 2. ***Lempel-Ziv 77 (LZ77).*** *$LZ77(T)$ is the sequence of triples*

$$LZ77(T) = \langle \pi_i, \lambda_i, c_i \rangle_{i=1,\ldots,z}$$

where $\pi_i \in \{0, \ldots, n-1\} \cup \{\perp\}$ and $\perp$ stands for "undefined", $\lambda_i \in \{0, \ldots, n-2\}$, $c_i \in \Sigma$, and:

1. *$T = \omega_1 c_1 \ldots \omega_z c_z$, with $\omega_i = \epsilon$ if $\lambda_i = 0$ and $\omega_i = T[\pi_i, \ldots, \pi_i + \lambda_i - 1]$ otherwise.*

2. *For any* $i = 1, \ldots, z$*, the string* ω_i *is the* longest *prefix of* $\omega_i c_i \ldots \omega_z c_z$ *that occurs at least twice in* $\omega_1 c_1 \ldots \omega_i$

Each $T[\pi_i, \ldots, \pi_i + \lambda_i - 1]$ *is called a LZ77* factor *or* phrase.

When dealing with the Lempel-Ziv factorization of a text T, we will assume that T ends with a character # not appearing elsewhere in T. Our algorithms computing LZ77 employ an FM-index over the reversed text. For this reason, in Chapters 4 and 5 we will assume that the input text begins with \$ (BWT terminator) and ends with # (LZ77 terminator).

Space

Depending on the application, we will measure space usage of our data structures in bits or words. To avoid confusion, we always specify whether space is measured in bits or words.

The notation $o(f(n))$ indicates—informally—a function *asymptotically smaller* than $f(n)$. More formally:

Definition 3. ***Small o notation.***

$$f(n) \in o(g(n)) \Leftrightarrow \lim_{n\to\infty} f(n)/g(n) = 0$$

Equivalently:

$$f(n) \in o(g(n)) \Leftrightarrow f(n) \in \mathcal{O}(g(n)) \wedge g(n) \notin \mathcal{O}(f(n))$$

Example 1.

- $n/\log n \in o(n)$
- $n \log\log n/\log n \in o(n)$

In practice, adding $o(n)$ (or $o(n \log \sigma)$) bits of space to our data structures will be acceptable in some applications. For reasonable n (e.g. around 2^{32}), $n/\log n$ is just 3% of n, while $n \log\log n/\log n$ is 15% of n.

2

Indexing and Compression: a Tale of Time and Space

2.1 A Gentle Introduction to Compressed Indexing

A great deal of information in our daily life is stored without respecting any particular structure or order. Consider, for example, a sheet containing informations about the residents of a big city. We may imagine, for simplicity, that the sheet stores just the age of one million people, together with an index enumerating lines:

```
1          78
2          15
     ...
500000     35
500001     27
     ...
999999     89
1000000    31
```

It is clear that, without any particular ordering of this data, it is very difficult to answer even simples queries on it. For example, consider the problem of finding out how many residents are aged 27. The only option we have to solve this problem is to scan the list and count the number of entries equal to 27. Letting $n = 1000000$ be the number of people, this process requires exactly n steps since we cannot exclude any line from our search.

Indexing Suppose now that our list is ordered by increasing age:

```
1          1
     ...
335466     26
335467     27
     ...
335501     27
335502     28
     ...
1000000    92
```

The above problem becomes incredibly easier: we only have to find the first and last residents aged 27, and subtract their indexes (in the example above: $335501 - 335467 + 1 = 35$). Since ages are ordered, this task can be solved very efficiently with the same procedure used to look up words on a dictionary: binary search. In total, we need to perform two binary searches taking approximately $\log_2(1000000) \approx 20$ steps each, for a total of about 40 steps. This is approximately 25000 times faster than our previous strategy: clearly, *ordering data* first is a great idea if we want to solve some problems on it!

Let us now move from numbers to words. Suppose that we wish to find out if a particular word appears in a text. Consider a big text (say, 1 million words) containing the following extract (to improve readability, we use underscores instead of spaces):

```
...ordering_is_good...
```

Again (as the phrase suggests), ordering the data is the key. We can *lexicographically* order words:

```
...
good
...
is
...
ordering
...
```

Looking up a word in the above list can be done again with binary search: this reduces the number of search steps to be performed from n (the number of words) to approximately $\log_2 n$, which results again in a 25000x speedup on a text with $n = 1000000$ words (actually, slightly slower since at each step we have to compare several characters to find out if our word is smaller or larger than the word we are looking in the list).

We are now going to take a little step further, which however will complicate (not by little) our original problem. What if we wish to check the existence of *substrings* instead of words? a substring is any contiguous sequence of characters in the text. For example, `ing_is_g` is a substring of the above text. How can the sorting idea be applied to *full texts*? A first naive solution is the following: since we are looking for substrings, just enumerate and sort all possible text substrings. For example, all possible substrings of the following (small) text:

```
sort
```

are (in lexicographic order): `o`, `or`, `ort`, `r`, `rt`, `s`, `so`, `sor`, `sort`, `t`. Unfortunately, there are $n(n+1)/2$ substrings in a text of length n. A text of length $n = 1000000$ contains about *500 billion* substrings: clearly, this solution is not feasible in practice.

A more clever solution follows from the observation that a substring is a prefix of a text suffix. For example, in the text `...ordering_is_good...` the substring `ing_is_g` is a prefix of the suffix `ing_is_good...`. There are just n text suffixes, so it seems feasible to limit our attention to this limited subset of substrings. The list of lexicographically sorted text suffixes contains, in particular, the following sub-sequence (possibly interleaved with other suffixes not shown in this example):

```
_good...
_is_good...
d...
dering_is_good...
ering_is_good...
g_is_good...
good...
ing_is_good...
is_good...
ng_is_good...
od...
ood...
ordering_is_good...
rdering_is_good...
ring_is_good...
s_good...
```

Does the string `ing_is_g` appear in the text? with binary search we quickly discover that this string is indeed a prefix of the line starting with `ing_is_good...` (therefore the answer is positive).

While looking appealing, after a second look it is evident that also this solution is not feasible: the total number of characters in the n suffixes is (again) $n(n+1)/2$. This is much larger than the original text size (n characters). We are, however, much closer to a true solution. Letting i be the starting position of the string `ordering_is_good` in our text, we can enumerate positions as follows:

i	i+1	i+2	i+3	i+4	i+5	i+6	i+7	i+8	i+9	i+10	i+11	i+12	i+13	i+14	i+15
o	r	d	e	r	i	n	g	_	i	s	_	g	o	o	d

The last step comes from the observation that, instead of storing entire text suffixes in sorted order, we can store their *starting positions* in the text. In particular, the list of such numbers —representing starting position of lexicographically sorted suffixes—contains the following subsequence (possibly interleaved with the starting positions of other suffixes omitted from this example):

```
i+11 i+8 i+15 i+2 i+3 i+7 i+12 i+5 i+9 i+6 i+14 i+13 i i+1 i+4 i+10
```

For example, the starting positions of suffixes `good...` and `is_good...` are $i+12$ and $i+9$, respectively. `good...` is lexicographically smaller than `is_good...`; as a consequence, $i+12$ comes before $i+9$ in our list.

Note that each text position takes at most $\lceil \log_2 n \rceil$ bits to be written, n being the text length. On ASCII texts smaller than 4 GB, each of these numbers can therefore be written using 4 Bytes (i.e. characters). As a result, our ordered dataset takes just $4n$ characters. This is a huge improvement with respect to the previous solutions! Search of a text substring works as before, except that we need to jump in the text (at the position we are looking in the above list) in order to actually read the text suffix. In order to search for a string of length m, we need to perform about $\log_2 n$ steps of binary search. In each of these steps, we need to compare at most m characters to discover whether the current

suffix is smaller or bigger than our query. Overall, this amounts to at most $m \log_2 n$ steps. On a text with $n = 1000000$ characters, searching for a substring of length $m = 10$ takes approximately $m \log_2 n \approx 200$ steps. This is 5000 times faster than scanning the whole text (required in absence of an index).

The list of numbers we just introduced takes the name of *suffix array* and has been independently invented by Udi Manber and Gene Myers in 1990 [78] and (under the name of PAT array) by Gonnet, Baeza-Yates and Snider in 1992 [1, 44]. The suffix array is an example of what is known in computer science as *data structure*: an organization of the data that accelerates some queries on it. In the case the data is a *text* and the queries are *looking for substrings*, the data structure takes the name of *full-text index*. Several full-text indexes have been introduced in the literature since the first appearance of such structures in the seventies. The first full text index ever introduced—the *suffix tree* [121]—can be traced back to the year 1973: 20 years before the appearance of the suffix array. This index is well known to be very fast (supporting the search of a string of length m in about m steps: faster than the suffix array), but very space consuming. In practice, a well-engineered implementation of a suffix tree can take up to tens of times more space than the suffix array.

Note that, in our discussion, we slowly shifted our attention from *speed* to *space*. We discovered that searching for substrings can be done quite time-efficiently, but it is not yet clear if the same task can be performed in small space. In our context, *space* is measured in terms of the number of Bytes (read: characters) needed to store our full text index. Even though the suffix array appears to be space-efficient, its overall size is 4 times that of the text. If added to the space needed to store the text, this space becomes $5 \cdot n$ Bytes. It is clear that, if the text is very big (e.g. 10 GigaBytes), it is not always feasible to store all of this data.

Compression Let us momentarily shift our attention to a seemingly distant concept that could, however, solve our space problem: text compression. We achieve compression whenever we are able to exploit regularities in our text (for example, repetitions) to reduce the space needed to store it. Consider a text containing the following extract:

```
...compressed_text_is_smaller_than_uncompressed_text...
```

The substring `compressed_text` is repeated. Letting i be the starting position of the above extract in our text, we could replace the second occurrence of the repeated substring with the string `[i,15]`, indicating that we have to copy 15 characters starting from position i (we can assume that characters '[' and ']' are reserved, so that we can distinguish between text characters and integers):

```
...compressed_text_is_smaller_than_un[i,15]...
```

Letting n be the text length, the first representation takes n characters, while the second $n - 9$. Both strings represent the same text: we achieved compression. The technique used in this example is very close to the same—LZ77—used in the most efficient practical text compressors; this compressor is discussed more in detail in Section 2.2.3. This and other compression techniques are able to considerably reduce the size of typical texts. For example, texts written in the English language can often be reduced in size by a factor of 4 using compression (meaning that a text of size 4 GB can be compressed to

approximately 1 GB). The size of the compressed text can be used as an approximation of its effective information content: even if our English text takes 4 GB to be stored, the effective information contained in it amounts to about 1 GB (i.e. the size of the compressed text). This concept can be understood considering borderline cases: the text `AAA...AAA` composed by 1000000 A's can be compressed very efficiently by simply storing the instruction

```
copy 1000000 times A
```

taking only 20 Bytes. This means that the effective information content of this text is very low (close to, and possibly lower than, 20 Bytes).

Now, recall that our full-text index based on the suffix array takes $5n$ Bytes of space. Since English text can be compressed by a factor of 4, we have that the suffix array takes 20 times more space than the sheer information content of the text. It is clear that this is an unsatisfactory situation: we would like to be able to quickly search substrings in a text, but without wasting huge amounts of space. Compression and indexing, however, seem rather distant concepts. Can we compress an index or—alternatively—index a compressed text? This intriguing problem remained open until the beginning of the 21st century, when a series of works [33, 35, 60] showed that, essentially, compression and indexing are two sides of the same coin.

Compression $\Leftrightarrow$ Indexing Let us focus again on the text extract

```
...compressed_text_is_smaller_than_uncompressed_text...
```

whose compressed representation is

```
...compressed_text_is_smaller_than_un[i,15]...
```

It is not too hard to be convinced of the fact that, in this particular example, also the suffix array can be compressed. Let i be the starting position in the text of our extract. The substring `compressed_text` starting from position $i+34$ and ending at position $i+48$ is copied from position i, so we can actually omit text positions $i+34, i+35, ..., i+48$ from the suffix array without losing the ability of finding *any* text substring. We have two cases:

(i) The substring we are looking for does not start inside $[i+34, i+48]$. Example: substring `small`. Then, its starting position is inside the "compressed" suffix array and we can find it with the standard search procedure.

(ii) The substring starts from one of the omitted positions. Example: the rightmost occurrence of `ompress`. Then, this substring has been copied so it appears also before the omitted text portion: we will still find `ompress` since the starting position of its first occurrence (i.e. $i+1$) is inside the "compressed" suffix array.

Despite being over-simplified, this example illustrates what happens with indexes based on the *Lempel-Ziv* compressor [69, 123, 124]. These indexes are discussed more in detail in Section 2.4.3. Clearly, the general case is much more complicated: substrings of the text can be copied from substrings that are themselves copied from other text portions.

Moreover, substrings can even copy themselves. However, in this complicated general case something rather interesting happens: the compressed representation *becomes the index*. To find a particular text substring, we just follow copy-links (in our example, `[1,15]`), i.e. we "navigate" the structure of the compressed file. Even more remarkably, the compressed index *replaces* the text: we can delete the original text file and keep only the index. In this case, the data structure takes the name of *self-index*. This phenomenon happens also with other (rather different) compressed representations. Another important suffix array compression paradigm, introduced in Section 2.2.2, is the Burrows-Wheeler transform (BWT) [11]. The BWT is a text permutation obtained by replacing each suffix array entry j with the $(j-1)$-th text character. It can be shown (more details in Section 2.4.1) that the BWT is an *invertible permutation* of the text that can be used to emulate the suffix array. Being a text permutation, the BWT takes just n characters, and can therefore be considered a *compressed* suffix array. By employing compression techniques such as *entropy* compression or *run-length encoding*, the BWT can be furthermore compressed to (often much) less than n characters. We will discuss these techniques in Sections 2.2.1 and 2.4.2.

How do compressed indexes perform *in practice*? As we show experimentally in this thesis, on very repetitive texts a compressed index can be *thousands* of times smaller than a suffix array. Even on not very repetitive texts, a compressed index takes at least 4 times less space than the suffix array. `compression` $\Leftrightarrow$ `indexing` implies that we can accelerate by thousands of times substring searches while using (often, several times) less space than the text. The problem of designing and efficiently building fast full-text indexes for all these compressed representations is the subject of this thesis.

From this point, the description has to become more technical. We need to formally introduce some compression techniques and describe in detail what the field of compressed full-text indexing has achieved up to now. After this chapter, we will deal with the original contributions of this thesis.

2.2 Basic Concepts

In this section we introduce the concepts of entropy, text compression techniques (Burrows-Wheeler transform, Lempel-Ziv parsing), space measures (succinct, compact, compressed), and we describe the model of computation (word RAM) assumed throughout the thesis.

2.2.1 Entropy

Consider the problem of encoding an alphabet Σ by assigning a binary code $B(c) \in \{0,1\}^{>0}$ to each character $c \in \Sigma$. The goals are (i) to make the encoding *decodable*, and (ii) to minimize the bit-length of the encoded text $B(T) = B(T[0])B(T[1])\dots B(T[n-1])$. Requirement (i) can be achieved by using so-called *prefix-free* codes:

Definition 4. *A prefix-free code is an assignment $B : \Sigma \to \{0,1\}^*$ such that $B(x)$ is never a prefix of $B(y)$, for every $x \neq y \in \Sigma$.*

A clever way to minimize space occupancy of the encoded text is to assign shorter codes to more frequent characters. One question we could ask is therefore the following: given that our text is generated by a zero-order source $\mathcal{S}$ fully characterized by character frequencies $f : \Sigma \to [0,1]$ (i.e. character c is generated with probability $f(c)$ *independently* from characters that have already been generated), what is the shortest average bit-length that such an encoding can achieve? The answer (Shannon's source coding theorem, 1948 [109]) is the *zero-order entropy* of the source:

Definition 5. *The zero-order entropy $H_0(\mathcal{S})$ of a zero-order source $\mathcal{S}$ with associated character distribution $f : \Sigma \to [0,1]$ is defined as:*

$$H_0(\mathcal{S}) = -\sum_{c \in \Sigma} f(c) \log f(c)$$

The above definition applies to the case in which we know the exact distribution of the character source that generated the input text. In practice however, the input of our compression algorithm is a text $T \in \Sigma^n$ without information about the source that generated it (in the hypothesis that the text was generated by a zero-order source, which is not necessarily the case). In such case, not all hope is lost: for sufficiently large n, we can approximate the source distribution by counting absolute character frequencies in T. This approach leads to a different (and more practical) notion: *empirical entropy.*

Definition 6. *The zero-order empirical entropy $H_0(T)$ of a string T is defined as:*

$$H_0(T) = \sum_{c \in \Sigma} \frac{n_c}{n} \log \left(\frac{n}{n_c} \right)$$

where n_c is the number of occurrences of c in T.

When clear from the context, we will write H_0 instead of $H_0(T)$. A prefix-free encoding of the text cannot use less than nH_0 bits of space. H_0 reaches its maximum of $\log \sigma$ when all characters have the same frequency: this is expected, since in this case character frequencies do not tell us anything useful to compress the text.

Huffman encoding

A practical way to (almost) reach the entropy is Huffman encoding [51]. Huffman algorithm takes as input the alphabet Σ and a frequency $f(c)$ for every $c \in \Sigma$, and outputs a function $B : \Sigma \to \{0,1\}^{>0}$ that assigns a prefix encoding to Σ. The algorithm is very simple:

1. Pick the two characters $a, b \in \Sigma$ with the smallest frequency
2. group them in a *metacharacter* $\langle a, b \rangle$ and set $f(\langle a, b \rangle) = f(a) + f(b)$
3. remove a and b from Σ, insert $\langle a, b \rangle$ in Σ. Repeat from (1) until $|\Sigma| = 1$ holds.

The above algorithm creates a binary tree with one character per leaf. This tree encodes the function B: while descending the tree to reach a leaf $c \in \Sigma$, assign a 0 to left branches and a 1 to right branches. For example, $\langle a, \langle b, c \rangle \rangle$ encodes $B(a) = 0$, $B(b) = 10$, and $B(c) = 11$. By Huffman-encoding T, we use no more than $n(H_0 + 1)$ bits of space (note that this is a conservative upper bound that is actually not reached). This is slightly more than the nH_0 promised by theory because Huffman assigns an *integer* number of bits to each character. A consequence of this fact is that we spend *at least* 1 bit per character (which is not good if $H_0 \ll 1$). Other compression schemes such as *arithmetic encoding* or *binomial/multinomial codes* (see Section 2.3.1) are able to break the 1-bit-per-character barrier.

High-order entropy

How can we improve upon the above compression strategy? One idea could be to look at *contexts* (i.e. substrings) that *follow* each text character in order to choose the encoding of that character. That is, we could fix a $k \geq 0$ and, instead of using only one zero-order compressed encoding B, use σ^k encodings $B_{w_1}, ..., B_{w_{\sigma^k}}$, where the strings $w_1, ..., w_{\sigma^k} \in \Sigma^k$ are all possible combinations of k characters from Σ. At this point, the encoding of a character is chosen depending on the k symbols that follow it in the text.

For example, let $T = cgcgacgcg$ and $k = 3$. The first character is encoded as $B_{gcg}(c)$, the second as $B_{cga}(g)$, the third as $B_{gac}(c)$, and so on. Note that if $k = 0$ then we have only one encoding function B_ϵ: we are back to zero-order entropy compression. Why should this compress better than H_0? Consider the example of a Java textbook and let $k = 5$. What letter will most likely precede every occurrence of the string "mport"? clearly, the answer is 'i': this means that the function B_{mport} will assign a very short code to the letter 'i' (probably just 1 bit), and longer codes to other letters.

More formally:

Definition 7. *Let $w \in \Sigma^k$, $0 \leq k < n$ be a* context*. With w_T we denote the string consisting of the concatenation of individual symbols preceding w in T.*

Example 2. *Let $T = mississippi$*

- *$w = si$. Then, $w_T = ss$*
- *$w = i$. Then, $w_T = mssp$*
- *$w = is$. Then, $w_T = ms$*

- $w = \epsilon$ *(empty string). Then,* $w_T = T = mississippi$

Definition 8. *The k-th order empirical entropy* $H_k(T)$ *of a string* T *is defined*[1] *as:*

$$H_k(T) = \frac{1}{n} \sum_{w \in \Sigma^k} |w_T| H_0(w_T)$$

When clear from the context, we will write H_k *instead of* $H_k(T)$*.*

In the above definition, we are putting in the same "bucket" all characters followed by the same context, and then compressing each bucket with a zero-order compressor (e.g. Huffman).

In practice, if it is easy to predict the following text character given some preceding characters, then H_k compresses *much* better than H_0, for k big enough (see Table 2.1). It should be clear that the parameter k represents a trade-off: big k improves compression, but we also need somehow to take into account the overhead of storing the (up to σ^k) encoding functions. To understand this note, consider the limit case $k = n$. In this case, there is only one character per context, therefore $H_k = 0$. Obviously, this does not mean that the string can be compressed to $nH_k = 0$ bits: there is a hidden space complexity in the encoding functions that we necessarily need to store in order to retrieve the original string. When presenting a result involving high-order compression, the value of k should be explicitly stated. In some cases, the claimed space bounds hold for *every* value of k: this follows from the fact that the additional space overhead of the data structures increases (often exponentially) with k. In other cases, k is fixed (usually, $k \in o(\log_\sigma n)$).

There is a nice way of visualizing this idea of *partitioning* text characters *by context*. We could *sort* all the circular permutations of T in a matrix $M \in \Sigma^{n \times n}$. At this point, characters in the last column of M will be partitioned by length-k contexts, *for any* k. For reasons described in the next section, we assume that T ends with a special character \$ not appearing elsewhere in T and lexicographically smaller than all other alphabet's characters. Figure 2.1 represents this matrix for the text $T = mississippi\$$. For example, let $k = 2$. The fourth and fifth rows of the matrix correspond to context "is": the two characters 's' and 'm' in the last column at these rows are both followed by this context inside T and are thus adjacent in the last column of the matrix.

Note that the last column of the matrix is a permutation of T. We could factorize this permutation according to length-k contexts: in the above example, with $k = 2$ this factorization would be "i/p/s/sm/\$/p/i/ss/ii". At this point, we could apply Huffman encoding to each factor. As a result, we compress T to $n(H_k + 1)$ bits.

2.2.2 Burrows-Wheeler Transform

The text permutation introduced at the end of the previous section is known as the *Burrows-Wheeler Transform* (BWT) of the text [11]. We have already seen that the BWT acts as a *compression booster*, turning any zero-order compressor (e.g. Huffman) into a high-order compressor[2]. However, this boost would be useless if we were not able to

[1] Actually, the original definition uses context that *precede* a character. However, it can be shown that the two measures differ only by low-order terms, so we will use this one as it simplifies the analysis of the structures that will be presented.

[2] This, however, is not necessarily the partitioning of the text guaranteeing the *best* compression. See [31] for efficient algorithms computing such a partitioning.

Collection	H0	H1	H2
CODE SOURCES	5.537 (69.21%)	4.038 (50.48%)	3.012 (37.65%)
MIDI	5.633 (70.41%)	4.734 (59.18%)	4.139 (51.74%)
PROTEINS	4.195 (52.44%)	4.173 (52.16%)	4.146 (51.82%)
DNA	1.982 (24.78%)	1.935 (24.19%)	1.925 (24.06%)
ENGLISH	4.529 (56.61%)	3.606 (45.08%)	2.922 (36.53%)
XML	5.230 (65.37%)	3.294 (41.17%)	2.007 (25.09%)

Collection	H3	H4	H5
CODE SOURCES	2.214 (27.68%)	1.714 (21.43%)	1.372 (17.15%)
MIDI	3.457 (43.21%)	2.334 (29.18%)	1.259 (15.74%)
PROTEINS	4.034 (50.43%)	3.728 (46.61%)	2.705 (33.81%)
DNA	1.920 (24.00%)	1.913 (23.91%)	1.903 (23.79%)
ENGLISH	2.386 (29.83%)	2.013 (25.16%)	1.764 (22.05%)
XML	1.322 (16.53%)	0.956 (11.95%)	0.735 (9.19%)

Table 2.1: Values of $H_0, \dots H_5$ for six texts: code sources (Java, C++), MIDI files, a protein database, DNA files, English texts, and XML files. The values represented in the table are *bits per character* and—between parentheses—*compression rate* with respect to ASCII encoding (7 bits per character). Note that DNA is almost a random text (on $\Sigma = \{A, C, G, T\}$), and we always need $\log \sigma \approx 2$ bits per character to represent it. On the other hand, this particular XML file is extremely repetitive: H_5 leads to a 10x compression rate. This is about 7 times better than using just H_0. Data coming from [37].

$mississippi
i$mississipp
ippi$mississ
issippi$miss
ississippi$m
mississippi$
pi$mississip
ppi$mississi
sippi$missis
sissippi$mis
ssippi$missi
ssissippi$mi

Figure 2.1: Sorted circular permutations of T. In the last column, T characters are grouped *by context*.

invert $BWT(T)$ (i.e. obtaining the original T). As it turns out, $BWT(T)$ can be inverted without the need of storing additional information other than $BWT(T)$. At the core of this and many other important properties of the BWT, stands the *LF mapping* property. It is very convenient to reason about the BWT using its matrix form introduced in the previous section. We call the first and last columns of this matrix F and L, respectively. L

corresponds to $BWT(T)$. Most importantly, note that we do not need to store in memory the whole matrix: column L is sufficient to reconstruct (locally) the matrix and is all we will ever need to store during the execution of our algorithms.

The LF property, depicted in figure 2.2 (only on character 'i'), links characters between the L and F columns and can be stated as follows:

Theorem 1. LF mapping. *The i-th occurrence of $c \in \Sigma$ on column L (enumerating top-down the positions) corresponds to the i-th occurrence of c on column F (i.e. they represent the same position on the text).*

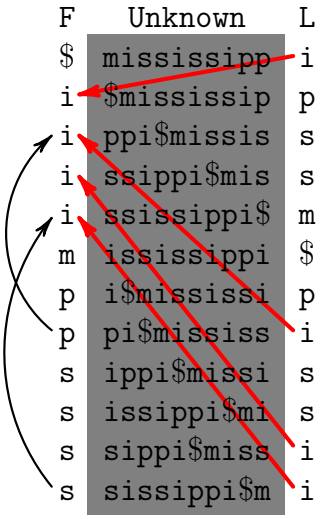


Figure 2.2: Red arrows: LF mapping (only on character 'i'). Black arrows: backward navigation of T. The central part of the matrix is not stored in practice, so here we declare it as "Unknown".

It is not hard to prove the above property. The key idea is to think about the *ordering* of equal characters on the two columns. In column F, equal characters (for example, the i's) are ordered according to the lexicographic order of the text suffixes that follow them (by definition of BWT). It is easy to see that the same property holds for equal characters on the L column, therefore occurrences of any $c \in \Sigma$ appear in the same order in the F and L columns.

A first important consequence of this property is that the BWT can be inverted. As displayed in Figure 2.2, by following a LF link and then going back to the L column at the same row (i.e. black arrows in Figure 2.2) we navigate T *backwards* by 1 position. Then, we can just start from character \$ (which we know to appear at the end of T) and reconstruct T *backwards* one character at a time by following LF links. As we will see in Section 2.4.1, LF links can be computed very quickly by adding some light structures on top of the BWT.

Much harder is the task of *efficiently building* the BWT. These are the best bounds known to date:

Theorem 2. *$BWT(T)$ can be built in $\mathcal{O}(n \log n / \log\log n)$ time and high-order compressed working space [87]*

Theorem 3. *$BWT(T)$ can be built in $\mathcal{O}(n)$ time and $\mathcal{O}(n \log \sigma)$ working space [4, 6, 82]*

To conclude this section, note that text substrings are (by definition) sorted in the F column of the BWT matrix. This suggests that the matrix could be used as a *text index* by binary-searching a pattern on it. As we will see (section 4), the LF property actually enables for an even more efficient and elegant search algorithm on the BWT: *backward search*.

2.2.3 Lempel-Ziv Parsing

Let us now introduce another compression paradigm: the Lempel-Ziv factorization (or parsing). The idea behind this family of compression techniques is to break (factorize) the text into *phrases* in such a way that each phrase appears before in the text. At this point, compression can be achieved by representing each phrase with a pointer to the text ($\mathcal{O}(\log n)$ bits). Also in this case, we append a terminator symbol # at the end of the text (for reasons explained below).

Definition 9. LZ78 factorization *[124]. The LZ78 factorization $S_1 S_2 ... S_z = T\#$ of $T\#$ is defined as:*

- *$S_i = a \in \Sigma \cup \{\#\}$ if this is the first occurrence of a*
- *$S_i = S_j a$ otherwise, where $j < i$, $a \in \Sigma \cup \{\#\}$. S_i is the longest phrase with this property*

Example 3.

- $T = ACGCGACACACACGGTGGGT$
- $lz78(T\#) = A|C|G|CG|AC|ACA|CA|CGG|T|GG|GT|\#$

Example 4.

- $T = AAAAAAAAAAAAAAAAAAAA$
- $lz78(T\#) = A|AA|AAA|AAAA|AAAAA|AAAAA\#$

Note that we need to append a character $\# \notin \Sigma$ at the end of the text because, otherwise, the last phrase could be not well-defined (as in example 2: without # the last phrase would be equal to its source).

If we represent the factors as z pairs $\langle text\ position, char \rangle$ (putting a NULL text position if the phrase is the first occurrence of a character) we achieve compression. To distinguish between the actual factorization of the text into substrings and its representation as a list of pairs, we use the notation lz78(T) for the former and LZ78(T) for the latter.

Example 5.

- $T = ACGCGACACACACGGTGGGT$
- $lz78(T\#) = A|C|G|CG|AC|ACA|CA|CGG|T|GG|GT|\#$
- $LZ78(T\#) = \langle -, A\rangle\langle -, C\rangle\langle -, G\rangle\langle 1, G\rangle\langle 0, C\rangle\langle 5, A\rangle\langle 1, A\rangle\langle 3, G\rangle\langle -, T\rangle\langle 2, G\rangle\langle 2, T\rangle\langle -, \#\rangle$

The following properties hold for the number z of phrases of the LZ78 factorization (refer to the next section for a definition of the small-o notation used here):

1. $z \log n \leq nH_k + o(n \log \sigma)$ for $k = o(\log_\sigma n)$
2. $z \in \Omega(\sqrt{n})$

Property 2 is easy to prove: in the best case each phrase adds 1 character to the longest previous phrase. For a proof of property 1, see [63].

A much more powerful variant of this technique is LZ77: a factor can be copied from *anywhere* in the text that precedes it (not just from the beginning of a previous phrase), and can end at *any* position (not just at the end of the copied phrase). In this case, we can encode each factor with a triple $\langle \mathit{text\ position}, \mathit{length}, \mathit{char} \rangle$, where *length* is the number of copied characters. LZ77 was formally defined in Definition 2.

Example 6.

- $T = ACGCGACACACACGGTGGGT$
- $lz77(T\#) = A|C|G|CGA|CA|CACACG|GT|GGG|T\#$
- $LZ77(T\#) = \langle -, 0, A\rangle\langle -, 0, C\rangle\langle -, 0, G\rangle\langle 1, 2, A\rangle\langle 1, 1, A\rangle\langle 6, 5, G\rangle\langle 2, 1, T\rangle\langle 13, 2, G\rangle\langle 15, 1, \#\rangle$

Example 7.

- $T = AAAAAAAAAAAAAAAAAAAA$
- $lz77(T\#) = A|AAAAAAAAAAAAAAAAAA\#$
- $LZ77(T\#) = \langle -, 0, A\rangle\langle 0, 19, \#\rangle$

Note that we can have *self-references*: a phrase can copy characters from *itself* (see, e.g. phrase 2 in Example 6). The following properties hold for LZ77:

1. As for LZ78, $z \log n \leq nH_k + o(n \log \sigma)$ [63]
2. However, with LZ77 is possible that $z \in \Theta(1)$

Property 2 is evident in Example 6 (compare it with Example 3). Since we encode each factor with $\mathcal{O}(\log n)$ bits, Property 2 implies that LZ77-based compressors can compress the text *exponentially*. This is a result we cannot achieve with BWT+entropy compression[3].

An important quantity linked to LZ77 is the *parse height*. We start by defining the *character height*. The character height $h_i,\ 0 \leq i < n$, is the number of times we have to follow copy-links starting from text position i until we reach a trailing character of some LZ77 phrase. More formally:

[3] Although combining the BWT with run-length encoding we can achieve exponential compression. More on this later.

Definition 10. ***Character height.*** *Let $0 \leq i < n$ be a text position. Let moreover $len(j) = \sum_{k=1}^{j}(\lambda_k + 1)$ be the cumulative length of the first j phrases (including trailing characters). If $T[i]$ is the trailing character c_j of some LZ77 phrase $\langle \pi_j, \lambda_j, c_j \rangle$, then $h_i = 0$. Otherwise, let $\langle \pi_j, \lambda_j, c_j \rangle$ be the phrase containing position i, i.e. such that $len(j-1) \leq i$ and $len(j) > i$. Let i' be the position such that $T[i]$ is copied from $T[i']$, i.e.*

$$i' = \pi_j + (i - len(j-1))$$

Then, h_i is defined recursively as $h_i = h_{i'} + 1$.

Definition 11. ***Parse height.*** *The parse height h is defined as the maximum character height:*

$$h = \max_{0 \leq i < n} h_i$$

h often appears in the complexity analysis of data structures based on LZ77 as it represents the worst-case complexity of extracting a character from $LZ77(T)$ by following copy-links. Note that, in the worst case, h can be as large as $\Theta(n)$ (e.g. consider extracting $T[n-2]$ from $LZ77(c^{n-1}\#) = \langle \perp, 0, c \rangle \ \langle 0, n-2, \# \rangle$).

2.2.4 Full-Text Indexing

This thesis deals with *full-text indexes*, i.e. data structures supporting fast substring search queries on a text. In particular, we will tackle the problem of *designing* and *building* compressed full-text indexes using a limited amount of resources (in terms of time and space). The indexes we will describe use the compression techniques introduced in this section in order to—at the same time—*index* and *compress* the text. We start by giving a formal definition of full-text index:

Definition 12. ***Full-text index.*** *A full-text index on a text $T \in \Sigma^n$ is a data structure $\mathcal{I}(T)$ that permits to answer efficiently to the following queries:*

1. `Locate(P)`*, where $P \in \Sigma^m$. Return the set $OCC = \{i \mid T[i, \ldots, i + |P| - 1] = P\}$, i.e. return all occurrences of P in T*
2. `Count(P)`*. Return the number $occ = |OCC|$ of occurrences of P in T*
3. `Extract(i,m)`*. Return $T[i, ..., i + m - 1]$*

In the above definition, the term "efficiently" means much faster than $\mathcal{O}(n)$ time (otherwise a scan of the text is sufficient in order to answer count and locate queries). More precisely, we are looking for indexes at most $\mathcal{O}(\log^{\mathcal{O}(1)} n)$ times slower than the optimal solution (suffix trees are optimal in this respect: read below). Note that `extract` queries are trivial if the text is stored in plain format. We will call an index that supports `extract` without the need of the text a *self-index*. In this case, the text can be deleted and we can keep just the index: a self-index is an augmented representation of a text that also supports search functionalities on it, while at the same time occupying (almost) the same space of the compressed text. As a comparison for future references, here we remind the space/time bounds of suffix trees:

- Space: $\mathcal{O}(n \log n)$ bits

- Count: $\mathcal{O}(m)$ time
- Locate: $\mathcal{O}(m + occ)$ time
- Extract: $\mathcal{O}(m)$ time (text is available)

Note that suffix trees support all queries in *optimal time*[4]. As noted in the Introduction, however, $\mathcal{O}(n \log n)$ bits of space are often too much in practice. In Section 2.4 and in the following chapters we show that compressed full-text indexes entirely solve this problem by integrating compression and indexing in the same data structure. The main idea behind compressed full-text indexes is to augment compressed representations of a text with some light data structures supporting search queries. Such data structures are described in detail in Section 2.3: we will start with succinct and compressed bitvectors with support for `rank`, `select`, and `access` (`RSA`) queries and use them to obtain compressed strings with support for `RSA` queries and geometric range data structures.

We describe and use in our results two classes of compressed full-text indexes: *FM-indexes* and *LZ-indexes*. Indexes belonging to the former class consist of a compressed Burrows-Wheeler transform with support for `rank` and `access` queries. These indexes support fast count and locate queries, but cannot achieve exponential compression. FM-indexes can employ any technique to compress the Burrows-Wheeler transform; we describe static and dynamic indexes based on entropy compression (Sections 2.4.1, 3.2, 4.2) and on run-length encoding (Sections 2.4.1, 3.3, 5.3.3).

As the name suggests, the latter class of indexes is based on the Lempel-Ziv parsing (either LZ77 or LZ78). Such indexes exploit the fact that LZ phrases are copied from previous text positions: intuitively, if a pattern P occurs inside a LZ phrase, then it occurs also in its source. Indexes based on LZ77 can achieve exponential compression. We describe LZ-indexes in Section 2.4.3.

2.2.5 Measuring Space

In this thesis we deal with *space-efficient* data structures on strings. This means that we need to be able to measure very accurately the space of these structures. Unfortunately, the big-$\mathcal{O}$ notation is not good for this purpose as it hides constants from the analysis: there is a huge difference, for example, between an index taking $2n \log \sigma$ bits of space and one taking $17n \log \sigma$ bits of space. Using big-$\mathcal{O}$, the space of both these structures is $\mathcal{O}(n \log \sigma)$ bits. There is a name for this space occupancy:

Definition 13. ***Compact space.*** *A compact data structure on a string $T \in \Sigma^n$ has size (in bits)*

$$\mathcal{O}(n \log \sigma)$$

For more precise measures of space we need to use the *small-o notation* (see Section 1.4 for a formal definition of small-o notation):

Definition 14. ***Succinct space.*** *A succinct data structure on a string $T \in \Sigma^n$ has size (in bits)*

$$n \log \sigma + o(n \log \sigma)$$

[4]For simplicity, we do not consider *packed* texts, i.e. texts where blocks of $w/\log_2 \sigma$ characters are packed in a single memory word. In this case, the lower bounds for `count`, `locate`, and `extract` would be of $\mathcal{O}(m \log \sigma/w)$, $\mathcal{O}(m \log \sigma/w + occ)$, and $\mathcal{O}(m \log \sigma/w)$, respectively.

or, equivalently, $n \log \sigma(1 + o(1))$ *bits*

Note that the information-theoretic minimum number of bits required to store a generic sequence of length n from Σ is $\log_2(|\Sigma^n|) = n \log \sigma$, so the above space occupancy is—in general—optimal up to low-order terms. If we move our attention from generic strings to a specific string $S \in \Sigma^n$ however, we can exploit regularities in S in order to represent it in *less* than $n \log \sigma$ bits. In this case, we refer to the structure as being *compressed*. Note that, when dealing with compressed data structures, the particular adopted compression measure (e.g. entropy, number of LZ77 phrases, or number of BWT runs) should also be clearly stated (as they behave differently on different types of data).

2.2.6 Model of Computation

To conclude the section, a few notes on the model of computation used: the *Word Random Access Machine* (word RAM) model. The model comes with a parameter, w, indicating the bit-size of a *memory word*. Data is organized into main memory in blocks of w bits, and we can randomly access in constant time any of such blocks. The model permits moreover to perform in *constant time* a certain set of operations between memory words $W \in \{0, 1\}^w$:

- `Multiplication`. $W_1 * W_2$
- `Division`. W_1/W_2
- `Addition/subtraction`. $W_1 \pm W_2$
- `Bitwise masks` (bitwise AND, OR, negation)
- `Population count`. $popcount(W)$ = number of bits set in W

We will also make use of *bit shift* operations $W >> i$ and $W << i$ (bits exiting from the word are lost). Such operations are particular cases of division and multiplication, respectively.

We will assume that the model is *trans-dichotomous*, i.e. that we can store a pointer to the text or a counter in a constant number of words. Letting n be the size of our text instance, this translates to the formal requirement $n \leq 2^{\mathcal{O}(w)}$. Note that this is an assumption often made but not always explicitly stated (e.g. in sorting algorithms, we assume we can represent the number n of integers, as well as any manipulated integer, in a constant number of memory words).

Despite including constant-time `popcount` in the model might sound permissive, we will make use of this operation only inside succinct static/dynamic bitvectors (see Sections 2.3.1, 6.3.2, and 3.2.1). In such cases, the assumption of a constant-time hardware implementation for `popcount` can be dropped by limiting the word size to $w = \log n$ (n being the bitvector length) and storing popcount values in an auxiliary precomputed table of size $\mathcal{O}(2^{w/c})$ words, for any constant c. This additional space is $o(n)$ for any choice of $c > 1$, so it will not impact on the leading terms in the space complexities of our structures.

2.3 Building Blocks

We can now start introducing some basic succinct and compressed data structures. We start with *bitvectors*. See [85] for another excellent and comprehensive survey on the topic.

2.3.1 Bitvectors

Bitvectors are, arguably, the most important data structures we are going to use in our indexes. Despite their simple definition, bitvectors are fundamental building blocks in structures representing strings, sets, geometry and graphs (to name a few). In this section we will introduce three different versions—differing in their space usage—of bitvectors: succinct, entropy-compressed, and gap-compressed. Let us start with the definition of *bitvector data structure*:

Definition 15. ***Bitvector.*** *A bitvector B over a length-n bit-sequence $b_0 b_1 ... b_{n-1} \in \{0,1\}^n$ is a data structure supporting efficiently the following operations (`RSA` queries):*

- *`Rank`.*
  - $B.rank_1(i) = \sum_{j=0}^{i-1} B[j]$
  - $B.rank_0(i) = i - B.rank_1(i)$
- *`Select`.*
  - $B.select_1(i) = max\{j \mid B.rank_1(j) = i\}$
  - $B.select_0(i) = max\{j \mid B.rank_0(j) = i\}$
- *`Access`.* $B[i] = b_i$

In the following, we will make no distinction between a bitvector data structure and its underlying bit-sequence, and we will use the term *bitvector* for both of them. Informally, $rank_b(i)$ is the number of bits equal to b before position i excluded. $select_b(i)$, on the other hand, is the position j of the i-th bit equal to b (enumerating bits from 0). We will use the abbreviation `RSA` to indicate rank-select-access queries. The particular case where only `rank`, $\texttt{select}_1$, and `access` queries are supported takes the name of *indexable dictionary*. Since we will need (see Section 2.3.2) to support `RSA` queries on general alphabets by using bitvectors as building blocks, in this section we will focus on the more general bitvector data structure—also called *fully indexable dictionary*—supporting also $\texttt{select}_0$.

What does *efficiently* mean in the above definition? by storing B in plain format (i.e. with just a sequence of n bits) we can answer `access` in constant time. This is clearly efficient. It is easy to answer also `rank` and `select` in constant time: `rank` requires just n counters storing all $\texttt{rank}_1$ values, while to implement $\texttt{select}_b$ we could store $B.rank_b(n)$ pointers to B. This solution is however not *space-efficient*, as the counters require $\mathcal{O}(n \log n)$ bits to be stored.

A second solution is to store just one $\texttt{rank}_b$ counter every $2 \log n$ positions of B (i.e. $n/\log n$ counters overall). In this way, $\texttt{rank}_b$ can be implemented in $\mathcal{O}(\log n)$ time by reading one counter and accessing at most $\mathcal{O}(\log n)$ bits. $\texttt{Select}_b$ could be implemented with a binary search on the $\texttt{rank}_b$ values, plus $\mathcal{O}(\log n)$ accesses ($\mathcal{O}(\log n)$ total time). This structure takes only $2n$ bits of space and is time-efficient. However it is not *optimal*,

neither in time (the optimal time is *constant* for all operations) nor in space (the optimal is n bits).

It turns out that we can still answer *all* queries in constant time with a data structure taking only $n + o(n)$ bits of space (succinct). This result is due to the pioneering works of Jacobson [52] (`rank`) and Clark [14] (`select`) on succinct data structures.

Constant-time rank in succinct space Here we consider only $\texttt{rank}_1$. The solutions for $\texttt{rank}_0$ are symmetric. We divide the bitvector in blocks of $\log n$ bits and superblocks of $\log^2 n$ bits. Then:

- For each superblock, we store explicitly the rank up to that position. This requires $\log n \cdot n / \log^2 n = n / \log n = o(n)$ bits of space
- For each block, we store the partial rank from the beginning of the corresponding superblock. Note: a superblock contains at most $\log^2 n$ bits set, so a local rank counter requires only $\log \log^2 n = \mathcal{O}(\log \log n)$ bits. Total space: $\mathcal{O}(\log \log n \cdot n / \log n) = o(n)$ bits
- We moreover store the plain bit sequence $b_0 b_1 \dots b_{n-1}$ *packing* blocks of w bits in memory words. That is, $b_0 b_1 \dots b_{n-1}$ is stored in memory as the sequence of memory words[5] $W_0 W_1 \dots W_{n/w-1}$, where $W_i = b_{i*w} b_{i*w+1} \dots b_{i*w+w-1}$

At this point, it is easy to answer $B.rank_1(i)$ in constant time: first, sum the values of the counters of the superblock and of the block containing position i. We are left with the problem of counting how many bits set appear before position i *inside* the block containing this position. A linear scan would take $\mathcal{O}(\log n)$ time; however, remember that we *packed* bits in memory words, and a memory word has size $w \geq \log n$. It is therefore easy to see how to retrieve this value in constant time with a constant number of `popcount` and `mask` queries on at most two memory words in $W_0 W_1 \dots W_{n/w-1}$ (i.e. the single word or the two words that intersect the block).

Example 8.

- *bit sequence: 11001011010010100101010010101110*
- $n = 32$
- *block size:* $\log n = 5$
- *superblock size:* $\log^2 n = 25$

For simplicity, assume that $w = \log n = 5$ *(blocks therefore correspond to memory words). We pack the bitvector in 7 memory words (in the last one we add a padding of 3 bits, not shown here):*

```
bitvector   11001 01101 00101 00101 01001 01011 10
superblocks 0                               12
blocks      0     3     6     8     10    0     3
```

[5] For simplicity, let us assume that w divides n

Constant-time select in succinct space Here we consider only $\texttt{select}_1$. The solutions for $\texttt{select}_0$ are symmetric. Let $t_1 = \log^2 n$. We divide the bitvector in r superblocks $B_0, ..., B_{r-1}$ containing exactly t_1 1's each. Note that these blocks are not all of the same size: they can contain any number of bits greater than t_1 (and smaller than n, of course). Let $k = \lfloor j/t_1 \rfloor$. Then:

$$B.select_1(j) = \sum_{h=0}^{k-1} |B_h| + B_k.select_1(j - k \cdot t_1)$$

Note that (each possible value of) the summation can be *explicitly stored* using r counters. This takes space $\mathcal{O}((n/t_1)\log n) = \mathcal{O}(n/\log n) = o(n)$ bits (maximum number of bits set is n) and reading the value of interest takes constant time. Let us see how to answer the *local* select query $B_k.select_1(j - k \cdot t_1)$. Depending on the size of B_k:

- If $|B_k| \geq \log^4 n$, then store explicitly all the t_1 select results. Overall space of these structures: at most $\mathcal{O}\left(\frac{n}{\log^4 n} t_1 \log n\right) = \mathcal{O}(n/\log n) = o(n)$ bits.
- Otherwise, divide the superblock in blocks containing $t_2 = \sqrt{\log n}$ bits set each. Repeat the main strategy: store explicitly the sums (one counter of $\log(\sqrt{\log n}) \in \mathcal{O}(\log\log n)$ bits per block) and local select.
  - *Explicit sums*: overall, at most $\mathcal{O}(\frac{n}{t_2}\log\log n) = \mathcal{O}(n\log\log n/\sqrt{\log n}) = o(n)$ bits
  - *Local select (1)*: if size of the block is $\geq \frac{\log n}{2}$, then store explicitly the t_2 select results. At most $2n/\log n$ such blocks, so in total $\mathcal{O}(\frac{n}{\log n} t_2 \log\log n) = \mathcal{O}(n\log\log n/\sqrt{\log n}) = o(n)$ bits.
  - *Local select (2)*: for blocks of size $< \frac{\log n}{2}$, we use a *universal table* storing all possible t_2 results of a select query on all possible combinations of blocks of length $\frac{\log n}{2}$ bits. The table takes $\mathcal{O}(2^{\log n/2} \cdot t_2 \cdot \log\log n) = \mathcal{O}(\sqrt{n\log n}\log\log n) = o(n)$ bits

Clearly, with the packed representation of the bit sequence described in Subsection 2.3.1, we can answer `access` in constant time (access a word and perform a shift and a mask). We obtain the following result:

Theorem 4. *The bitvector data structure above described takes $n + o(n)$ bits of space and answers `access`, `rank`, and `select` queries in constant time.*

Achieving compression: the RRR bitvector

It is not too difficult to improve the space of the bitvector to $nH_0 + o(n)$ bits (where $0 < H_0 \leq 1$) without affecting query times. The idea, described Raman, Raman, and Rao [105][6], is to divide the bit-sequence $b_0 b_1 \ldots b_{n-1}$ in blocks of $u = \log n/2$ bits and compress each block using the *binomial* code: there are $\binom{u}{C}$ possible bitvectors of length u with C bits set; the binomial code encodes the length-u block B with the pair $\langle C, O\rangle$

[6]Here we outline their structure presented at section 4.1: *Fully Indexable Dictionaries for Dense Sets.* The solution for indexable dictionaries is more space-efficient as it uses the information-theoretic minimum number of bits (modulo low-order terms).

(*class-offset*), where O (*offset*) is the position of B in the ordering of all possible length-u bitvectors with C bits set.

Example 9. *Let $u = 5$. Consider the block $B = 01000$. We have $C = 1$. The number of length-u bitvectors with $C = 1$ bits set is $\binom{5}{1} = 5$: 00001, 00010, 00100, 01000, 10000. B is the 4th in the list, so we will encode it as $\langle 1, 4\rangle$*

The number C requires only $\log u$ bits to be written. The number O has a variable length of $\log \binom{u}{C}$ bits that depends on C: from 1 bit (when $C = 0$) to $\mathcal{O}(u)$ bits (when $C = u/2$). It follows that blocks with few 1's or 0's are considerably (exponentially) compressed from u to $\log u$ bits. Again, we divide the bitvector in blocks of size u^2 bits. We can store the pairs $\langle C, O\rangle$ sharing a superblock consecutively in a bit-array, and store explicitly their start positions ($\mathcal{O}(\log u)$ bits each) inside this array to guarantee constant-time access (this requires $o(n)$ additional bits). It can be shown that with this encoding each block is compressed to its zero-order entropy, and with few more calculations we derive that the whole bit sequence is compressed to $nH_0 + o(n)$ bits. On top of this, we build the additional structures described in the previous section ($o(n)$ bits). We can use a universal table of $\mathcal{O}(2^u \cdot u) = o(n)$ bits[7] to decompress in constant time the blocks of interest while computing `rank`, `access`, and `select`.

Static gap-encoded bitvectors

Suppose that the input bitvector is very sparse, i.e. the number m of bits set is very small with respect to the bitvector size n. In this case, the $o(n)$ spatial term that comes with the solution described in the previous section could be asymptotically bigger than m, therefore we need yet another compression scheme capable to exploit the sparsity of the bitvector.

The RRR indexable dictionary described by Raman, Raman, and Rao [105] achieves this goal, but does not support $\texttt{select}_0$ and $\texttt{rank}_0$. Here we describe a very simple and self-contained solution to the fully indexable dictionary problem taking $m \log n + \mathcal{O}(m)$ bits of space and answering `RSA` queries in $\mathcal{O}(\log n)$ time. The advantage of this solution is that it is easy to dynamize (we will discuss dynamization in the next chapters). More space-efficient and faster solutions include Gupta et al. fully indexable dictionary [48] and Elias-Fano encoding of increasing sequences [26, 28, 92]. Let $0 \leq i_1 < \cdots < i_m < n$ be the positions of the bits set in bitvector $B[0, \ldots, n]$. The sequence $\langle i_1, \cdots, i_m\rangle$ (taking $m \log n$ bits of space) is sufficient to answer `rank` and `access` queries in $\mathcal{O}(\log m)$ time with a binary search on it, and $\texttt{select}_1$ queries in constant time with just one access. To support $\texttt{select}_0$, we can simply group the integers in blocks of size $\mathcal{O}(\log n)$ and build a binary tree having these blocks as leaves. We moreover store in each internal tree node x the number of 0's of the corresponding bitvector subsequnce induced by the integers stored in the subtree rooted in x. The tree takes $\mathcal{O}\left(\frac{m}{\log n} \log n\right) \in \mathcal{O}(m)$ bits of space. It is easy to see that we can answer $\texttt{select}_0$ queries in $\mathcal{O}(\log n)$ time by navigating the tree from its root to one of the leaves and then reading at most $\mathcal{O}(\log n)$ integers in the target leaf.

[7] The table is a matrix $H[C][O]$ indexed by class and offset values and whose elements are length-u bitvectors

2.3.2 Wavelet Trees

Our aim is now to generalize the result obtained in the previous section to *strings* on general alphabets. That is, we want to be able to efficiently answer `RSA` queries on a general string $S = c_0 c_1 \dots c_{n-1} \in \Sigma^n$. Letting $c \in \Sigma$, we want to support:

- $S.rank_c(i) = |\{j \mid c_j = c \ \wedge j < i\}|$
- $S.select_c(i) = max\{j \mid S.rank_c(j) = i\}$
- $S[i] = c_i$

Informally, $S.rank_c(i)$ is the number of characters equal to $c \in \Sigma$ before position i excluded, while $S.select_c(i)$ is the position of the i-th character equal to c.

Wavelet trees, first used in this context by Grossi et al. [47], are an extremely elegant data structure that permits to reduce `RSA` queries (as well as updates; more on this later) on general strings to `RSA` queries on bitvectors. See [30, 84] for two excellent surveys on the topic. In a wavelet tree, the string is represented as a binary tree with σ leaves. Each internal node of the tree is labeled with a bitvector data structure. Fix a prefix encoding $c : \Sigma \to \{0,1\}^*$ of the alphabet, and let $c(a)[i]$ be the i-th leftmost bit of $c(a)$, $a \in \Sigma$. $WT(S)$, $S \in \Sigma^n$, is a binary tree defined recursively as follows:

Definition 16. ***Wavelet tree****. The wavelet tree is a binary tree whose internal nodes are bitvectors and whose leaves are alphabet characters. The root of the tree is the bitvector $c(S[0])[0] \dots c(S[n-1])[0]$ formed by taking the first bit of each character in S. Then, build a sequence S_0 (resp. S_1) by taking (in S-order, left to right) all characters in S whose binary code starts with 0 (resp. 1) and by removing the leftmost bit from their codes. The left and right children of the root are, respectively, $WT(S_0)$ and $WT(S_1)$. The base case is reached when we are given as input a binary string $S \in \{0,1\}^{>0}$. In this case we build a node labeled with the bitvector S. The node has two leaves labeled with the two characters of the original alphabet Σ whose binary codes correspond to the path from the root to these leaves.*

The following example should be self-explanatory:

Example 10. *$S = mississippi$. Binary encoding of the alphabet: $c(m) = 00$, $c(i) = 01$, $c(s) = 10$, $c(p) = 11$. Figure 2.3 depicts $WT(S)$.*

Some important notes:

1. We *do not* store characters inside internal nodes (we store only bits). In Example 10, characters are shown also inside tree nodes only for explanatory purposes
2. In general, we need however to store characters in the leaves (to encode function c). However, for some encoding this can be avoided, e.g. when c is the natural binary encoding $c(0) = 00$, $c(1) = 01$, $c(2) = 10$, $c(3) = 11$ (we will use this encoding later with geometric data structures)
3. if we store the tree as $\mathcal{O}(\sigma)$ separate bitvectors, we have a $\mathcal{O}(\sigma \log n)$ extra term in the space usage ($\mathcal{O}(\sigma)$ pointers to nodes). It is actually possible to concatenate all bitvectors in a single bitvector while still being able to solve all queries in the same

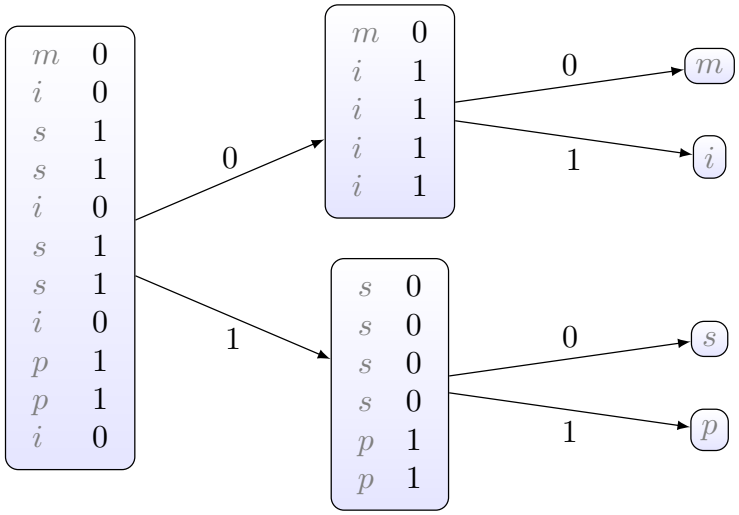


Figure 2.3: Wavelet tree of the string *mississippi*. Note that the i-th level stores the i-th bits in the binary representations of the characters. There are σ leaves, each labeled with an alphabet character.

time bounds. With this solution, the extra $\mathcal{O}(\sigma \log n)$ term vanishes. The tree of Example 10 is stored *level-wise* as the single bitvector (the delimiter | is used here only for clarity):

$$00110110110|01111|000011$$

Access To answer $S.access(i)$ using $WT(S)$, start from the bitvector B_R at the root R of the wavelet tree. By accessing $B_R[i]$ we get the first bit of the binary representation of $S[i]$. Note that the bitvector B_0 at node $R.child(0)$ stores the second bits of the binary representation of characters in S that start with bit 0 (similarly for bitvector B_1 at node $R.child(1)$). Let $x = B_R[i]$. It follows that the second bit of the binary representation of $S[i]$ is $B_x[B_R.rank_x(i)]$. By recursively repeating this strategy, we finally reach the leaf of $WT(S)$ labeled with character $S[i]$. Since we descend the tree from the root to a leaf and spend only constant time at each level (one access and one rank), running time is $\mathcal{O}(|c(S[i])|)$ (number of bits of the binary representation of $S[i]$).

The procedure is reported as Algorithm 1. We use the following notation: *root* is the root node of the WT, $N.bitvector$ is the bitvector associated with the node N, and $N.child(b)$, with $b \in \{0, 1\}$ is the node child of N reached following the edge labeled with bit b.

Algorithm 1: $access(i)$

```
1 N ← root;
2 while N is not leaf do
3 |   B ← N.bitvector;
4 |   b ← B[i];
5 |   N ← N.child(b);
6 |   i ← B.rank_b(i);
7 return N.label;
```

Rank Rank does not differ much from access. Let us start from bitvector B_R at the root R of the tree. In order to answer $S.rank_x(i)$ using $WT(S)$, first count how many characters in S start with bit $b = c(x)[0]$ before position i: $i' = B_R.rank_x(i)$. Then, move at position i' of the bitvector B_b associated with node $R.child(b)$ and repeat with the second bit of $c(x)$. By recursively repeating this strategy for all bits of $c(x)$, we reach the leaf of $WT(S)$ labeled with x. The last rank operation performed at the parent of this leaf yields exactly the value $S.rank_x(i)$. Running time of this procedure is $\mathcal{O}(|c(x)|)$.

Algorithm 2: $rank_x(i)$

```
1 N ← root;
2 k ← 0;
3 while N is not leaf do
4 |   B ← N.bitvector;
5 |   b ← c(x)[k];
6 |   i ← B.rank_b(i);
7 |   N ← N.child(b);
8 |   k ← k + 1;
9 return i;
```

Select To answer $S.select_x(i)$ using $WT(S)$, start from the parent N of the leaf of $WT(S)$ labeled with x. Let B_N be the bitvector associated with node N. Let $b = c(x)[|c(x)| - 1]$ be the last bit of $c(x)$. By computing $i' = B_N.select_b(i)$ we retrieve the position i' of the i-th character equal to x in bitvector B_N. Now, jump to the parent N' of N and repeat the strategy substituting i with i' and performing select with bit $b' = c(x)[|x| - 2]$. The procedure terminates at the root R of $WT(S)$. The select operation performed on B_R yields exactly the value $S.select_x(i)$. Again, running time of this procedure is $\mathcal{O}(|c(x)|)$.

The procedure is reported as Algorithm 3. We use the following additional notation: $leaf(x)$, $x \in \Sigma$, is the leaf of the WT labeled with character x. $N.parent()$ is the parent node of N.

Algorithm 3: $select_x(i)$

```
1 N ← leaf(x);
2 k ← |c(x)| − 1;
3 while N is not root do
4 |   N ← N.parent();
5 |   B ← N.bitvector;
6 |   b ← c(x)[k];
7 |   i ← B.select_b(i);
8 |   k ← k − 1;
9 return i;
```

What encoding?

Note that we can encode each character by using just $\log \sigma$ bits[8]. By implementing each tree node with the succinct bitvector described in the previous section, the tree takes only $n \log \sigma(1 + o(1))$ bits of space. This is (modulo the small $o(n \log \sigma)$ term) exactly the *information-theoretic minimum* number of bits required to store any sequence of length n over Σ. Notably, the tree supports `access`, `rank`, and `select` queries in $\mathcal{O}(\log \sigma)$ time.

However, the only requirement we asked for the encoding $c()$ is to be a *prefix code*. What happens if we use Huffman encoding? The algorithms described in the previous section still work, and we obtain a *Huffman-shaped* wavelet tree. Remarkably, this data structure takes $nH_0(1+o(1))+n$ bits of space and answers all queries in (average) $\mathcal{O}(H_0)$ time: the tree is both *compressed* and *faster*.

Example 11. *$S = mississippi$. Use the Huffman encoding m = 001, i = 01, s = 1, p = 000. Figure 2.4 depicts $WT(S)$ with the shape induced by this encoding.*

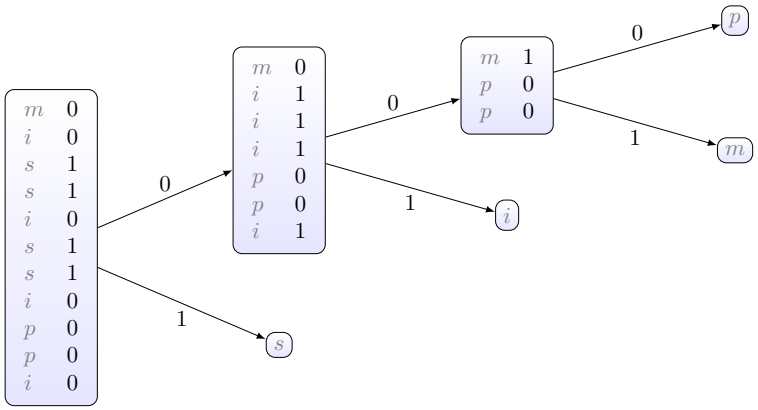


Figure 2.4: Huffman-shaped wavelet tree of the string *mississippi*. Note that the depth of each leaf is inversely proportional to the frequency of its label: on average, this improves running times from $\mathcal{O}(\log \sigma)$ (of the fixed-length encoding of Figure 2.3) to $\mathcal{O}(H_0)$.

[8]For simplicity, we assume the alphabet size is a power of 2. The exact formula is $\lceil \log_2 \sigma \rceil$ bits

Huffman-shaped wavelet trees are all we need to build a simple compressed BWT-based self-index (Section 2.4.1).

2.3.3 Run-Length Compressed Strings

In this section, we show how to support RSA queries on a string compressed with *run-length encoding* (RLE). RLE encoding represents a string S in a space proportional to the number r_S of its equal-letter runs. This compression strategy will turn out to be particularly effective when applied to the Burrows-Wheeler transform (Section 2.4.2). The material of this section is taken from Mäkinen et al. [77].

To support RSA queries on a string $S \in \Sigma^n$, we store one character per run in a string $H \in \Sigma^r$, we mark the end of the runs with a bit set in a bit-vector $V_{all}[0, \ldots, n-1]$, and for every $c \in \Sigma$ we store all c-runs lengths consecutively in a bit-vector V_c as follows: every m-length c-run is represented in V_c as $0^{m-1}1$.

Example 12. *Let* $S = bc\#bbbbccccbaaaaaaaaaaa$. *We have* $H = bc\#bcba$, $V_{all} = 11100010001100000000001$, $V_a = 00000000001$, $V_b = 100011$, $V_c = 10001$, *and* $V_\# = 1$

In the following we denote with $RLE(S)$ the above compressed representation of S. RSA queries on $RLE(S)$ can be reduced to RSA queries on H, V_{all}, and V_c as follows.

Access to answer $RLE(S).access(i)$, we need to find out the rank of the run containing text position i. Using this value, we then access H and retrieve $S[i]$:

$$RLE(S).access(i) = H[V_{all}.rank_1(i)]$$

Rank Algorithm 4 reports the pseudocode answering $RLE(S).rank_c(i)$ queries. First, we find the rank t of the run containing position i (Line 1). Then, if the t-th run is a c-run, we calculate the offset off from the beginning of the run to position i (Line 3), otherwise we set $off = 0$ (Line 5). Finally, we count the number k of c's contained in all c-runs preceding position i (excluding the current run if it is a c-run, Line 7) and return $off + k$ (Line 8).

Algorithm 4: $rank_c(i)$

input : Position i on the text, character $c \in \Sigma$
output: Number of c's in S before position i excluded

```
1 t ← V_all.rank_1(i);                          /* rank of run contaning position i */
2 if H[t] == c then
3 |  off ← i − (V_all.select_1(t − 1) + 1);      /* if t == 0, set off to i */
4 else
5 |  off ← 0;
6 t' ← H.rank_c(t) − 1;                  /* ranks of full c-runs before i:  0,...,t' */
7 k ← V_c.select_1(t') + 1;                      /* if t' == −1, set k to 0 */
8 return off + k;
```

Select Algorithm 5 reports the pseudocode answering $RLE(S).select_c(i)$ queries. In Line 1 we discover the rank t (inside V_c) of the c-run containing the i-th c with a $\texttt{rank}_1$ query on V_c. In line 2 we map the t-th c-run on H with a $\texttt{select}_c$ query on H; the result is the rank t' of the run containing the i-th c. In Line 3 we discover the first position k of the t'-th run with a $\texttt{select}_1$ query on V_{all} (as noted in the pseudocode, if $t' == 0$ then we set k to 0). All we have left to do is to compute the offset off representing the relative position of the i-th c inside its c-run. This is simply the distance between i and the first position fp in V_c of the c-run containing the i-th c. In line 4 we compute fp (as noted in the pseudocode, we have to take care of the special case in which this is the first c-run). Then (Line 5), off is computed as $i - fp$. To conclude, we return $k + off$ (Line 6).

Algorithm 5: $select_c(i)$

input : Integer $i \geq 0$, character $c \in \Sigma$
output: Text position of the i-th c

```
1 t ← V_c.rank_1(i);                          /* c-run containing the i-th c */
2 t' ← H.select_c(t);                         /* run containing the i-th c */
3 k ← V_all.select_1(t' − 1) + 1;             /* if t' == 0, set k to 0 */
4 fp ← V_c.select_1(V_c.rank_1(i) − 1) + 1;   /* if V_c.rank_1(i) == 0, set fp to 0 */
5 off ← i − fp;
6 return k + off;
```

Insert If the structures for H, V_{all}, and $V_c, c \in \Sigma$, support dynamism, then also `insert` operations can be supported on $RLE(S)$. In particular, we need support for `insert` queries on H, V_{all}, and V_c and $\texttt{delete}_0$ queries on V_{all} and V_c. In Sections 3.3.2, 6.3.1, and 6.3.2 we describe structures supporting efficiently all these operations.

Here we only give an overview of the strategy to support `insert`, as the full algorithm is quite involved[9]. We can distinguish three main cases. (i) We insert a c inside a c-run. Then, we have to insert a 0 in V_{all} and in V_c inside the two runs of zeros corresponding to the c-run. (ii) We insert a c inside a a-run, with $a \neq c$ (note: not at the end or at the beginning of the a-run: this case is slightly different). Then: (1) we insert the string ca inside H after the run-head a corresponding to the splitted a-run, (2) we replace[10] a 0 with a 1 inside V_a in correspondence to the point where the a-run has been splitted by the inserted c, (3) we replace a 0 with the pattern 11 inside V_{all} in correspondence to the point where the a-run has been splitted by the inserted c, and (4) we insert a 1 in V_c at the position corresponding to the new c-run. Note that these operations are slightly different in the case the c is inserted at the end or at the beginning of the a-run. (iii) We insert a c between a a-run and a b-run, with $c \neq a, b$ and $a \neq b$. Then: (1) we insert a c in H between the run heads a and b corresponding to the involved a- and b- runs, (2) we insert a 1 in V_c at the position corresponding to the new c-run.

[9] In our library `DYNAMIC`, the full procedure takes 200 lines of code even using the highest level of abstraction

[10] Note that replacing a 0 with a 1 can be implemented with a $\texttt{delete}_0$ and a $\texttt{insert}_1$

Compression Note that, if we gap-encode bitvectors V_{all} and V_c with the technique described in Section 2.3.1 and implement H with a wavelet tree (Section 2.3.2), the structure takes $\mathcal{O}(r_S)$ words of space. Note moreover that none of the queries requires $select_0$ on the gap-encoded bitvectors: this is important, since we can use an indexable dictionary to encode these components (e.g. based on Elias-Fano encoding: see, e.g., [94]) rather than a (more powerful) fully-indexable dictionary (see the beginning of Section 2.3.1 for the definition of indexable and fully-indexable dictionaries).

2.3.4 Geometric Data Structures

Consider a $n \times n$ two-dimensional grid where every row and every column contain exactly one two-dimensional point (see Table 2.2 for an example), i.e. a permutation of the numbers $\{0, \ldots, n-1\}$. We want to build a data structure that permits to efficiently answer *range reporting* queries on the set of grid points: given a rectangle $\mathcal{Q}$ on the grid, report all points in the grid lying inside the rectangle $\mathcal{Q}$. In our case, *efficiently* means in time proportional to the number k of reported points. We will describe a solution based on wavelet trees that solves the problem in $\mathcal{O}((k+1)\log n)$ time and $n \log n(1 + o(1))$ bits of space. Note that $n \log n$ bits is the information-theoretic lower bound required to store a generic permutation of $\{0, \ldots, n-1\}$, so our solution is optimal in space up to lower-order terms. See [75] for a full description of the original result.

5		•				
4					•	
3				•		
2			•			
1						•
0	•					
	0	1	2	3	4	5

Table 2.2: Example: 6×6-grid with exactly one point per row and column. X and Y positions are numbered starting from 0. This grid corresponds to the permutation $\langle 0, 5, 2, 3, 4, 1\rangle$ of the set $\{0, 1, 2, 3, 4, 5\}$.

5		•				
$\underline{4}$					•	
$\underline{3}$				•		
$\underline{2}$			•			
$\underline{1}$						•
0	•					
	0	$\underline{1}$	$\underline{2}$	$\underline{3}$	4	5

Table 2.3: Range reporting query on the rectangle $\mathcal{Q} = [1, 3] \times [1, 4]$. The result is the set of points $\{\langle 2, 2\rangle, \langle 3, 3\rangle\}$

More formally, our data structure should support efficiently the following queries:

Definition 17. ***4-sided range reporting.*** *Let $\langle i_0, \ldots, i_{n-1}\rangle$ be a permutation of $\{0, \ldots n-1\}$, and let $\mathcal{P} = \{\langle 0, i_0\rangle, ..., \langle n-1, i_{n-1}\rangle\}$ be a set of points on the $n \times n$ grid. Moreover,*

let $\mathcal{Q} = [i', i''] \times [j', j'']$ *be a query rectangle on the grid, where* $0 \leq i' \leq i'' < n$ *and* $0 \leq j' \leq j'' < n$*. The 4-sided range reporting problem asks to report all points in the intersection* $\mathcal{P} \cap \mathcal{Q}$

The structure $WT(i_0 \dots i_{n-1})$ (wavelet tree over the sequence $i_0 \dots i_{n-1}$) is sufficient to solve the 4-sided range reporting problem with the above mentioned time bounds. Figure 2.5 depicts how the grid in Table 2.2 is encoded with a wavelet tree.

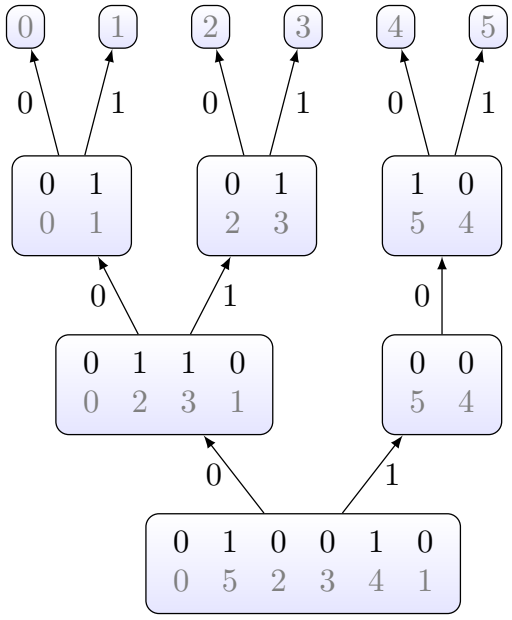


Figure 2.5: Wavelet tree for the permutation $S = 052341$. We encode numbers with 3 bits each by using the natural binary encoding of integers (e.g. c(5)=101). Only bits are stored (not numbers in gray). X coordinates of the 2D points are encoded in the position of the sequence, while Y coordinates are encoded in the sequence content: point $\langle x, y \rangle$ is in the structure if and only if $S[x] = y$.

Note that every tree node is associated with an interval of Y-coordinates. For example, the node reached by following from the root the path 01 is associated with the interval $[010, 011] = [2, 3]$, and the node reached by following from the root the path 1 is associated with the interval $[100, 111] = [4, 7]$

At this point, the algorithm to answer 4-sided range reporting queries $Q = [i', i''] \times [j', j'']$ using the wavelet tree is easy. We start at the root on the subsequence $S[i', i'']$, and map this interval recursively on children nodes[11]. We stop whenever we reach a node whose associated range of Y-coordinates does not intersect $[j', j'']$. At the end, we will reach $d \leq k$ intervals on d bitvectors above the leaves. These d intervals contain the k Y-coordinates of the points in the intersection $\mathcal{P} \cap \mathcal{Q}$. We can retrieve their corresponding X-coordinates by navigating the tree from these positions to the root (*select* queries). This

[11] i.e. interval $[i', i'']$ on the root forks in the 2 intervals $[B.rank_0(i'), B.rank_0(i'' + 1) - 1]$ on child 0 and $[B.rank_1(i'), B.rank_1(i'' + 1) - 1]$ on child 1, where B is the bitvector at the root.

procedure terminates in $\mathcal{O}((k+1)\log n)$ time. $[j', j'']$ is exactly covered by $\mathcal{O}(\log n)$ tree nodes. It takes $\mathcal{O}(\log n)$ time to reach these nodes. From these nodes, all positions in the intervals that we map to children nodes correspond to solutions, since their Y-coordinates are contained in the original $[j', j'']$ interval. Finally, for each of the k solutions navigate upwards to the root to retrieve their X-coordinates.

2.4 Compressed Full-Text Indexing

We now have all the ingredients we need to describe two classes of compressed indexes: FM and LZ indexes.

2.4.1 FM-Indexes

Indexes from the FM-index families achieve high-order compression (close to nH_k bits) and support all queries in time linear in the pattern length. The first index of this kind was originally proposed by Ferragina and Manzini [33], and is based on the suffix-sorting properties of the Burrows-Wheeler transform. The data structure is essentially a BWT with support for `rank` operations (we do not need `select`); as we will see, `rank` permits to compute efficiently the LF function (Theorem 1), which is at the core of the search algorithm. In their original proposal, the authors describe a rank solution that works for constant-sized alphabets; they implement `rank` on the BWT in constant time by sampling rank values (for every character) at regular positions of the BWT. The alternative solution here described is based on wavelet trees and works for general alphabet size.

Backward search

Patterns can be efficiently searched on the BWT using the *backward search* algorithm. The reason for this name is that we search the pattern from its last to first character (right-to-left instead of left-to-right as done in suffix trees or suffix arrays). The algorithm is based on two observations:

1. All occurrences of $P \in \Sigma^m$ appear *contiguously* in a range of rows in the BWT matrix
2. Let $[l, r]$ be the BWT range of a pattern P. Then, characters in $BWT[l, r]$ *precede* the occurrences of P in the text

Refer to Figure 2.1 to understand the meaning of the above observations. For example, the two occurrences of pattern "iss" occur contiguously at rows 3 and 4 (counting from 0) in Figure 2.1. Then, characters in $BWT[3, 4] = sm$ precede the two occurrences of "iss" in the text. This means that, if we map (LF property) all characters 's' (in this case, just one) from $BWT[3, 4]$ to the F column, we will obtain the range for the pattern "siss". Moreover, since all occurrences of "siss" (in this case, just one) appear contiguously in the BWT matrix, mapping all 's' from $BWT[3, 4]$ to column F will require applying the LF mapping only to the first and last 's' inside $BWT[3, 4]$: this means that one backward search step requires a *constant* number of (`rank`) operations. See Figure 2.6 for a graphical example.

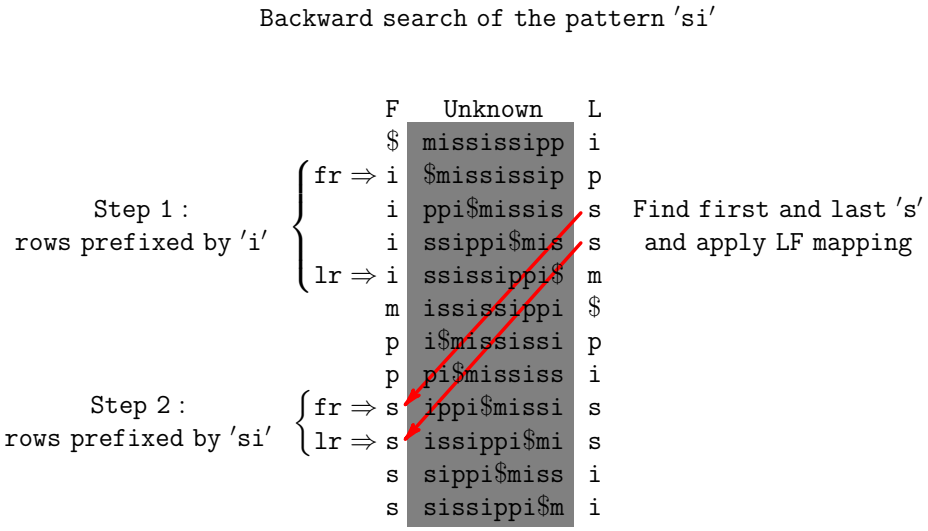


Figure 2.6: Backward search. To search "si", first find the range $[1, 4]$ of 'i' (this can be easily done). Then, locate the first and last 's' in $BWT[1, 4]$ (this requires just 2 rank operations), and map them to the F column, obtaining the range $[8, 9]$ corresponding to pattern "si".

Two rank operations are enough to perform a backward search step. All we need to do is to *count* the number of $c \in \Sigma$ (current character extension) before the beginning and before the end of the current BWT interval. This gives us the rank of the first and last c of the new range on the F column. The procedure is reported as Algorithm 6. Here, $BWT.F(c)$, with $c \in \Sigma$, is the position of the first c in column F of the BWT matrix (example: in Figure 2.6, $BWT.F(p) = 6$). This function can be encoded with an array taking $\sigma \log n$ bits, or can be computed efficiently using the BWT at no additional space cost. "fr" and "lr" stand for *first* and *last* positions of the BWT range, respectively. At the beginning, the range $[fr, lr]$ is the full range $[0, |T| - 1]$ corresponding to the empty string.

Algorithm 6: *backward_search*(P)

```
1 fr ← 0;
2 lr ← |T| − 1;
3 for i = |P| − 1 downto 0 do
4 |   c ← P[i];
5 |   fr ← BWT.F(c) + BWT.rank_c(fr);
6 |   lr ← BWT.F(c) + BWT.rank_c(lr + 1) − 1;
7 return [fr, lr];
```

It follows that, with just a Huffman-encoded wavelet tree on the BWT of the text, we can answer very efficiently *count queries* in compressed space:

	F											**L**
	\$	m	i	s	s	i	s	s	i	p	p	i
	i	\$	m	i	s	s	i	s	s	i	p	p
	i	p	p	i	\$	m	i	s	s	i	s	s
	i	s	s	i	p	p	i	\$	m	i	s	s
	i	s	s	i	s	s	i	p	p	i	\$	m
0	m	i	s	s	i	s	s	i	p	p	i	\$
9	p	i	\$	m	i	s	s	i	s	s	i	p
	p	p	i	\$	m	i	s	s	i	s	s	i
6	s	i	p	p	i	\$	m	i	s	s	i	s
3	s	i	s	s	i	p	p	i	\$	m	i	s
	s	s	i	p	p	i	\$	m	i	s	s	i
	s	s	i	s	s	i	p	p	i	\$	m	i

Table 2.4: We augment the BWT with n/t suffix array (SA) samples (here, $t = 3$)

Theorem 5. *$WT^{Huff}(BWT(T))$ is a partial text index taking $n(H_0 + 1)(1 + o(1))$ bits of space and supporting* count *of a pattern $P \in \Sigma^m$ in average $\mathcal{O}(m(H_0 + 1))$ time*

How can we support `locate` and `extract`? As for `locate`, the idea is to store on the BWT a suffix array sample (i.e. a text position) every t text positions, then use LF to backward-navigate the text until a sample is found.

Example 13. *We sample one over $t = 3$ text positions: m̲is̲is̲ip̲pi\$ (i.e. 0,3,6,9). Table 2.4 depicts how the BWT is augmented with these text positions.*

We still have one problem left to solve: how do we store SA samples *only* on sampled BWT positions? in the example in Table 2.4, we want to store the samples 0,9,6,3 only at positions 5,6,8,9 of the BWT. We cannot simply store a '0' in all other positions since this would require $n \log n$ bits (instead of $(n/t) \log n$). The solution is rather simple: we keep a bitvector B with a '1' in all sampled F-positions of the BWT and '0' in other positions. Then, we store the SA samples contiguously in a vector *SSA* (sampled Suffix Array) of length n/t. If $B[i] = 1$, then the sample associated with BWT position i is $SSA[B.rank_1(i)]$. Table 2.5 reports this strategy.

By encoding B with a succinct bitvector data structure, the overall space taken by SSA + B is $(n/t) \log n + n + o(n)$ bits. We choose $t = \log^{1+\epsilon} n$, where $\epsilon > 0$ is a constant. Then, this space becomes $n + o(n)$ bits. We can do better: since[12] $nH_0(B) \in \mathcal{O}((n/t) \log t)$, then $nH_0(B) \in \mathcal{O}((n/\log^{1+\epsilon} n) \log \log^{1+\epsilon} n) = o(n)$, so if we encode B with a RRR-bitvector (Section 2.3.1) the overall space of SSA + B is only $o(n)$ bits.

Algorithm 7: *locate*(i)

```
if B[i] = 1 then
    return SSA[B.rank_1(i)];
else
    return 1 + locate(BWT.LF(i));
```

[12] To prove this, use the definition of zero-order entropy and the expansion $\log x = \frac{x-1}{x} + \frac{1}{2}\left(\frac{x-1}{x}\right)^2 + \frac{1}{3}\left(\frac{x-1}{x}\right)^3 + ...$

B	F											**L**
0	$	m	i	s	s	i	s	s	i	p	p	i
0	i	$	m	i	s	s	i	s	s	i	p	p
0	i	p	p	i	$	m	i	s	s	i	s	s
0	i	s	s	i	p	p	i	$	m	i	s	s
0	i	s	s	i	s	s	i	p	p	i	$	m
1	m	i	s	s	i	s	s	i	p	p	i	$
1	p	i	$	m	i	s	s	i	s	s	i	p
0	p	p	i	$	m	i	s	s	i	s	s	i
1	s	i	p	p	i	$	m	i	s	s	i	s
1	s	i	s	s	i	p	p	i	$	m	i	s
0	s	s	i	p	p	i	$	m	i	s	s	i
0	s	s	i	s	s	i	p	p	i	$	m	i

Table 2.5: We can now store *contiguously* the sampled suffix array elements in an array `SSA=<0,9,6,3>`. To access sample at position i of the F column, access $SSA[B.rank_1(i)]$

The procedure for converting a BWT position to a text position is reported as Algorithm 7. Here, function $locate(i)$ takes as input a position i on the F column of the BWT matrix and returns the corresponding text position.

Note that, since we sampled one out of t *text* positions, procedure 7 performs at most t backward steps. Each step takes $\mathcal{O}(\log \sigma)$ time (LF function). In order to locate all $occ = lr - fr + 1$ occurrences of a pattern P, first locate its BWT range $[fr, lr]$ with Algorithm 6. Then, compute $locate(i)$ for every $fr \leq i \leq lr$.

To implement *extract*, we proceed in a similar way: we pick one out of t text positions, and on these positions we write the corresponding BWT position. The elements of this array—called here *SBP* (sampled BWT positions)—can be stored contiguously in $(n/t) \log n$ bits of space. Then, to extract $T[i, ..., i + m]$ just find the nearest BWT sample k after text position $i + m$ (i.e. the sample at text position $\lceil (i + m)/t \rceil \cdot t$: $k = SBP[\lceil (i + m)/t \rceil]$), jump to the BWT at position k and extract (backwards) at most $m + t$ characters using the LF function.

Remember that we have chosen $t = \log^{1+\epsilon} n$. Moreover, remember that by using Huffman-shaped wavelet trees we can improve times on average and achieve compression. We obtain:

Theorem 6. Huffman-shaped wavelet-tree FM-index (HWT-FMI). *$WT^{Huff}(BWT(T)) + B + SSA + SBP$ is a compressed self-index taking*

$$n(H_0 + 1)(1 + o(1))$$

bits of space and supporting:

- *Count in average $\mathcal{O}(m(H_0 + 1))$ time*
- *Locate in average $\mathcal{O}((m + occ \log^{1+\epsilon} n)(H_0 + 1))$ time, and*
- *Extract in average $\mathcal{O}((m + \log^{1+\epsilon} n)(H_0 + 1))$ time*

Where $\epsilon > 0$ is a constant fixed at construction time.

Note that $H_0 + 1$ can be *much smaller* than $\log \sigma$ on compressible texts: compression both *reduces the size* of the index and makes it *faster*.

High-order compression

How can we approach high-order compressed space? Remember that the BWT acts as a *compression booster*: this means that there exists a partitioning of the BWT in substrings such that, if we compress each of them to their zero-order entropy, then the BWT is compressed to nH_k bits (for every k). See [31] for full details. A simple (though not optimal) strategy to achieve this goal is to fix k and to consider the sub-optimal partitioning of the text induced by length-k contexts. There are in total σ^k contexts of length k, and for every context we need to store σ partial ranks storing the rank of every character up to the beginning of the context. We use a RRR-bitvector to mark context boundaries, and choose $k = \alpha \log_\sigma n - 1$, $0 < \alpha < 1$. It can be easily shown that with this data structure we can answer `access` and `rank` queries on the compressed BWT, so we obtain the following:

Theorem 7. HAF-FMI: Huffman-shaped alphabet-friendly FM-index *The structure described in this section is a compressed self-index taking*

$$n(H_k + 1)(1 + o(1))$$

bits of space, where $k = \alpha \log_\sigma n - 1$, $0 < \alpha < 1$, and supporting:

- *Count in average $\mathcal{O}(m(H_k + 1))$ time*
- *Locate in average $\mathcal{O}((m + occ \log^{1+\epsilon} n)(H_k + 1))$ time, and*
- *Extract in average $\mathcal{O}((m + \log^{1+\epsilon} n)(H_k + 1))$ time*

Note that the choice of using Huffman-shaped wavelet trees improves running times on average: with balanced WT the multiplicative term $(H_k + 1)$ in the query times is replaced by $\log \sigma$.

Note that we can improve the space usage to $nH_k + o(n \log \sigma)$ if we use balanced wavelet trees and encode their nodes with RRR-bitvectors (see Section 2.3.1). The original proposal described by Ferragina et al. [36]—the alphabet-friendly FM-index (`AF-FMI`)—uses this technique. In this case, the multiplicative term $(H_k + 1)$ in all query times is replaced by $\log \sigma$ and times become worst-case (not average-case). The name of the index (*alphabet friendly*) derives from the fact that there is only a logarithmic dependence with the alphabet size in the space term and query times. In the original proposal [33] of the FM-index, this dependence was exponential in the space term and linear (w.r.t. σ) in the query times. Recent results showed that it is possible to attain high-order compression in FM indexes also using a BWT partitioning based on fixed-size blocks [59]. Even more remarkably, it can be shown [74] that the zero-order compression scheme of Raman et. al [105] applied to the whole BWT (without partitioning at all) leads *implicitly* to high-order compression.

2.4.2 Run-Length Indexes

A key property of the Burrows-Wheeler transform is that it tends to contain a small number of equal-letter runs if the text is repetitive. The reason of this behavior is that repeated substrings are contiguous in the first columns of the BWT matrix, therefore characters preceding them are contiguous in the BWT. The number r of equal-letter runs of the BWT is tightly connected with the self-repetitiveness of the text and can be very small if the text is highly repetitive. Let $\Sigma = \{s_1, \ldots, s_\sigma\}$ be the alphabet. A naive lower bound on r is $\Omega(\sigma)$. This lower bound is reached in some strings, e.g. $(s_1 s_2 \ldots s_\sigma)^e$, for any $e > 0$, or in Standard words [79] (i.e. the finite counterparts of Characteristic Sturmian words). In the worst case however, r can be $\Theta(n)$. This is the case—for example—of *de Bruijn sequences* of order $k > 1$. In Section 5.1 we discuss more in detail such cases.

More important for the scope of this thesis is that r is related to the number of repetitions in the text. In particular, the following important theorem [111] gives an expected upper bound to the number of runs in the BWT of a highly repetitive text collection:

Theorem 8. Sirén, 2012 [111]. *Let $C = \{T_1, \ldots, T_t\}$ be a collection of t copies of a random text $T[1, n]$ over an alphabet of size σ and r be the number of runs in the BWT of T. Let C_0 be the same collection after randomly substituting a total of s characters with other characters. Then the expected number of runs in the BWT of collection C_0 is at most $r + \mathcal{O}(s \log_\sigma(tn))$*

Since each pair of consecutive differences between a text and one of its copies delimits a repetition in the collection, Theorem 8 links r with the number of self-repetitions of a (repetitive) text.

The Burrows-Wheeler transform can be compressed with *run-length encoding* by replacing it with the shortest list of pairs $RLBWT(T) = \langle c_i, \ell_i \rangle_{i=1,\ldots,r}$, $c_i \in \Sigma$, $\ell_i \in \mathbb{N}$ such that $BWT(T) = c_1^{\ell_1} c_2^{\ell_2} \ldots c_r^{\ell_r}$. Since r can be constant in extremely repetitive texts, by run-length encoding the BWT we can achieve (up to) *exponential* compression. Such representation can be augmented to support `RSA` queries as described in Section 2.3.3, and enables the definition of a simple run-length FM index, dubbed here `rlbwt`. To improve readability, in what follows we drop the $(1 + o(1))$ multiplicative term present in all space analysis.

Theorem 9. *`rlbwt` index. Let $k > 0$. The `rlbwt` index is a compressed self-index taking*

$$r(2 \log n + \log \sigma) + \mathcal{O}(r) + (n/k) \log n$$

bits of space and supporting:

- *`Count` in $\mathcal{O}(m \log n)$ time*
- *`Locate` in $\mathcal{O}((m + occ \cdot k) \log n)$ time, and*
- *`Extract` in $\mathcal{O}((m + k) \log n)$ time*

Proof. We encode the BWT with a run-length string as described in Section 2.3.3, using the gap-encoded bitvector described in Section 2.3.1 and a wavelet tree (Section 2.3.2) to encode its components. Letting $k > 0$ be the sampling factor, Theorem 9 easily follows. □

Note: by using the fully indexable dictionary of [48] to encode bitvectors, the leading term in the space of Theorem 9 can be reduced to $r(2\log(n/r) + \log\sigma) + (n/k)\log n$ bits, and the $\log n$ multiplicative term in all query times can be reduced to[13] $o(\sqrt{\log r}) + \log\sigma$. Space can be reduced to $2r\log(n/r) + r\log\sigma + \mathcal{O}(r) + (n/k)\log n$ bits also by using an Elias-Fano indexable dictionary [26, 28, 94] to encode gap-lengths. In this case, the $\log n$ multiplicative term in all query times is reduced to $\log(n/r) + \log\sigma$. This last solution is more practical, and efficient C++ implementations are available [43].

Another approach to run-length encoded indexes is that of combining *compressed suffix arrays* with run-length encoding. Compressed suffix arrays store in a compressed form the inverse ψ of LF function. It can be shown that ψ can be decomposed in σ strictly increasing subsequences, therefore it can be efficiently compressed with gap-encoding techniques (e.g. Elias delta or gamma encoding [27]). Another way to compress ψ is to replace each subsequence of integers $\langle j, j+1, j+2, \ldots, j+k\rangle$ with its length k (opportunely encoded). It can be shown [77, 111] that the number of such subsequences is r, i.e. the number of runs in the BWT of the text. This enables the definition of an index—the run-length compressed suffix array (`rlcsa`)—with the following space-time bounds:

Theorem 10. *`rlcsa` [77, 111]. The `rlcsa` (run-length compressed suffix array) is a compressed self-index taking*

$$r(2\log(n/r) + \log\sigma)(1+\epsilon) + (n/k)\log n$$

bits of space for any $k, \epsilon > 0$, and supporting:

- *`Count` in $\mathcal{O}(m\log n)$ time*
- *`Locate` in $\mathcal{O}((m + occ\cdot k)\log n)$ time, and*
- *`Extract` in $\mathcal{O}((m+k)\log n)$ time*

Note that all above solutions have a $r\log(n/r)$ extra spatial term with respect to a direct run-length encoding of the BWT (which can be represented in just $r\log(n/r)+r\log\sigma$ bits). This is due to the fact that runs are represented *twice* (separately for each character and globally for the whole sequence). One of the contributions of this thesis is to show (Section 5.3.2) how to reduce the coefficient of the term $r\log(n/r)$ from 2 to $(1+o(1))$. In practice (see Chapter 7), this will almost halven the space of the `rlcsa` while maintaining its time efficiency.

2.4.3 LZ-Indexes

Let us now move to a completely different indexing paradigm: indexes based on the Lempel-Ziv factorization. First of all, why do we need indexes based on yet another compression scheme? After all, entropy-compressed FM-indexes reach optimal space bounds—with respect to H_k—and nearly-optimal running times. The problem with such indexes is the compression scheme itself: entropy compression is not good if the text is *highly repetitive*.

[13] The exact time bound for rank operations—$AT(n, r)$—is more involved and we do not report it here. See the original paper [48] for full details.

Consider a text T, and a collection of texts $T_1, ..., T_q$ that are *almost identical*[14] to T. What is the size of the entropy-compressed concatenation $T_1...T_q$? To simplify analysis, we can assume $T_1 = ... = T_q = T$. Remember that H_k is defined using only *character frequencies* in the text. Note that in the concatenation TT character frequencies are *the same* as in the text T; as a result, $H_k(TT) \approx H_k(T)$ and therefore (generalizing this to q copies of T) $qnH_k(T^q) \approx qnH_k(T)$: the compressed representation of T^q takes q times the compressed representation of T (see Kreft and Navarro [67] for a more formal proof). We can easily do much better than that: for instance, we could compress T and append the note "decompress and copy q times". This representation takes only $nH_k(T) + \mathcal{O}(\log q)$ bits. It is therefore clear that entropy compression is not good for repetitive texts.

In Section 2.2.3 we got the impression that LZ compressors are particularly good for repetitive texts. This impression is correct. Let $z_X(T)$ (X=77,78) be the number of LZX factors of T. It is easy to see that $z_{77}(T^q) = z_{77}(T) + 1$: the LZ77-compressed representation of T^q takes almost the same space of the LZ77-compressed representation of T. LZ78 is less powerful but—as we will see—easier to index. In this section we study a compressed index based on LZ78 or LZ77 (the technique works for both compressors). This index requires the text stored in plain format ($n \log \sigma$ additional bits) in order to work, so it is not a self-index. We conclude by showing how to get rid of the text turning the index into a *self-index*. From now on, z will denote $z_X(T)$ (X=77,78 will be clear from the context).

State of the art

We quickly review the state of the art in the field of LZ-based indexing.

- `KU-LZI` (LZ78). Kärkkäinen and Ukkonen, 1996 [60]. $\mathcal{O}(z \log n) + (1 + o(1))n \log \sigma$ bits of space, $\mathcal{O}(m^2 + occ \log^\epsilon n)$ locate. *Not a self-index.*
- `FM-LZI` (BWT+LZ78). Ferragina and Manzini, 2005 [34]. Combines the FM-index with LZ78. $\mathcal{O}(nH_k \log^\epsilon n) + o(n \log \sigma)$ bits of space, where $\epsilon > 0$, $k \in o(\log_\sigma n)$. *Optimal* $\mathcal{O}(m + occ)$ locate time.
- `NAV-LZI` [83] (LZ78). Navarro, 2004. $\mathcal{O}(z \log n)$ bits of space, $\mathcal{O}(m^3 \log \sigma + (m + occ) \log n)$ locate. *Self-index* based on LZ78.
- `KN-LZI` [67] (LZ77). Kreft and Navarro, 2011. $\mathcal{O}(z \log n)$ bits, $\mathcal{O}((m^2 h + (m + occ) \log z) \log(n/z))$ locate. *Self index* based on LZ77. h is the *height of the parse*, see Definition 11.

Note that NAV-LZI and KN-LZI reach $\mathcal{O}(z \log n)$ bits of space. All other indexes have a $o(n)$ space term. In Section 2.2.5 we considered the spatial term $o(n)$ acceptable. Here, we are going to change our mind: $o(n)$ can be *exponentially* bigger than z (when using the LZ77 compression scheme), so an index fully exploiting the compression power of LZ77 should take only $\mathcal{O}(z \log n)$ bits of space.

In this section we first study a full-text index based on LZ77/LZ78 using techniques from Kärkkäinen and Ukkonen [60], Navarro [83], and from Kreft and Navarro [67] (the material presented in this section is therefore not new; see these works for full details). The index offers the following tradeoffs:

[14] e.g. all revisions of a software or a Wikipedia web page

- $\mathcal{O}(z \log n) + n \log \sigma$ bits of space
- $\mathcal{O}(m(m + \log z) + occ \log z)$-time locate

This index is not a self-index as we need the plain text to operate on it. We describe the structure using LZ78, but with small modifications the technique works also with LZ77.

Finally, by using a technique from NAV-LZI [83], we *get rid of the text*, turning the index into a *self-index*. This will give us an index of size $\mathcal{O}(z \log n)$ bits with the same query times as above. This technique works only with LZ78. We conclude (using ideas from [67]) by sketching how to apply the same ideas to LZ77-based self indexing.

A LZ-based full-text index

We first give an overview of the strategy we will use to locate pattern occurrences with LZ-indexes. Consider the following LZ78-parsed text. We underline *an occurrence* of the pattern $P = GACAC$ we are searching.

$$\text{LZ78(T\#)} = \text{A|C|G|CG|\underline{ AC|ACA}|CA|CGG|T|GG|GT|\#}$$

In order to locate the occurrence:

1. Split P in 2 parts: the one contained in the rightmost phrase (AC) and the rest (GAC)
2. Find the interval $[l^{fw}, r^{fw}]$ of AC among the lexicographically sorted LZ phrases
3. Find the interval $[l^{rev}, r^{rev}]$ of CAG (GAC reversed) among the lexicographically sorted reversed text prefixes that end at phrase boundaries (e.g. the first 4 such reversed prefixes are A, CA, GCA, GCGCA)
4. We build a geometric range structure connecting adjacent text prefixes and LZ phrases. With a 4-sided range search on the interval $[l^{fw}, r^{fw}] \times [l^{rev}, r^{rev}]$ we retrieve the text position of the pattern split

If—on the other hand—the occurrence of P is entirely contained in a phrase, we will use a different strategy: in this case, the occurrence is entirely copied from the phrase source. We will therefore first recursively find this source and then retrieve the occurrence contained in the phrase. Also this can be implemented using range search. Note that in step (1) we do not know how the pattern is split, so we have to try all possible m splits.

As pointed out in the above example, we need to treat separately two kinds of pattern occurrences:

1. `Primary occurrences`: those spanning at least two LZ phrases or that end a phrase
2. `Secondary occurrences`: those contained in a single LZ phrase (and that do not end a phrase)

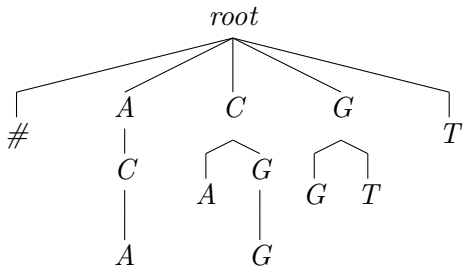
Let us start by introducing a fundamental structure in LZ78 indexes: the LZ78 trie.

LZ78 trie Each LZ78 phrase adds a character to a previous phrase, so all phrases can be organized in a trie with $z+1$ nodes[15].

Example 14. *Consider the text factored as*

$$LZ78(T\#) = A|C|G|CG|AC|ACA|CA|CGG|T|GG|GT|\#.$$

The resulting LZ78 trie is the following:



Phrases are in bijection with tree nodes (root excluded), so the trie has exactly $z+1$ nodes. We can store this structure in $\mathcal{O}(z \log z + z \log \sigma) \subseteq \mathcal{O}(z \log n)$ bits.

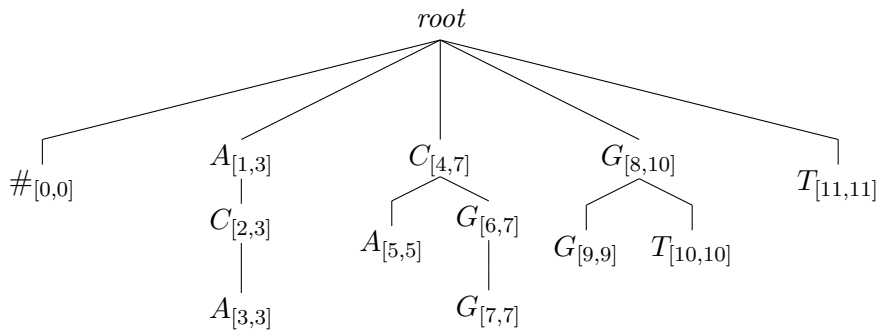
Consider now the *lexicographic order of the LZ78 phrases*. We can easily augment the LZ78 trie with the range $[l^{fw}, r^{fw}]$ of lexicographic ranks of phrases in the subtree rooted in every trie node.

Example 15. *The lexicographic order of the phrases in Example 14 is:*

0. $\#$
1. A
2. AC
3. ACA
4. C
5. CA
6. CG
7. CGG
8. G
9. GG
10. GT
11. T

Example 16. *We use the following notation on trie nodes to indicate lexicographic ranges:* $c_{[l^{fw}, r^{fw}]}$*, where* $c \in \Sigma$ *is the node label. Example: node corresponding to* G *is augmented with the interval* $[8, 10]$ *because all LZ78 phrases starting with* G *are (in order)* G*,* GG*,* GT *and their ranks are 8,9,10, respectively. Note that storing these ranges requires* $\mathcal{O}(z \log n)$ *bits of additional space.*

[15] The LZ78 trie is the main reason why LZ78 is easier to index than LZ77: in the latter, phrases cannot be organized in a (small) trie



Now we are going to do the same with reversed text prefixes ending at phrase boundaries.

Example 17. *Using the text of the above examples, the lexicographic order of reversed text prefixes ending at phrase boundaries is:*

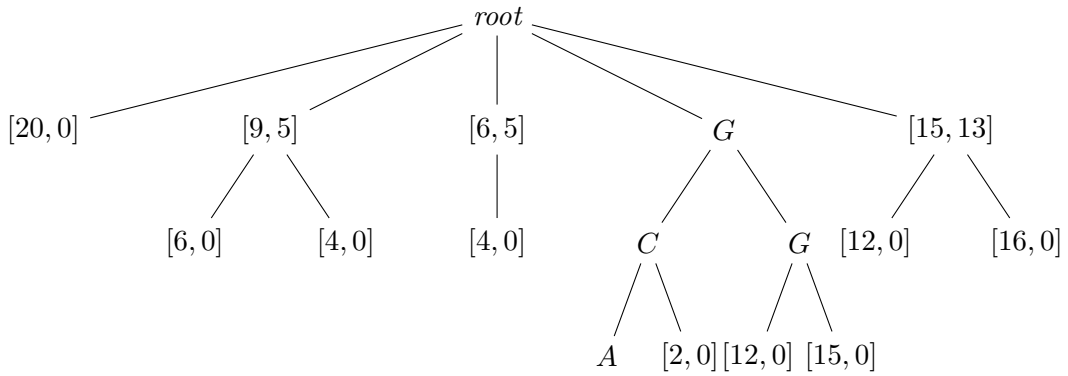
0. $\#TGGGTGGCACACACAGCGCA$
1. A
2. $ACACACAGCGCA$
3. $ACACAGCGCA$
4. CA
5. $CAGCGCA$
6. GCA
7. $GCGCA$
8. $GGCACACACAGCGCA$
9. $GGTGGCACACACAGCGCA$
10. $TGGCACACACAGCGCA$
11. $TGGGTGGCACACACAGCGCA$

Note that there are exactly z suffixes, but if we organize them in a trie we end up with $\mathcal{O}(zn)$ nodes. However, the trie has z leaves, so the number of internal nodes with at least two children (explicit nodes) is $\mathcal{O}(z)$. We use the same technique employed in suffix trees: path compression. Instead of storing explicitly unary paths, we store 2 pointers $[begin, end]$ to the text (where $begin > end$: we read the text backwards). For each explicit node N, we also store the interval $[l^{rev}, r^{rev}]$ of lexicographic ranks of the strings in the subtree rooted in N. This tree takes $\mathcal{O}(z \log n)$ bits of space. Note that we *need access to the text* to reconstruct unary paths, so we store it in plain format: $n \log \sigma$ additional bits.

Example 18. *For clarity, we write only* $[begin, end]$ *labels.* $[l^{rev}, r^{rev}]$ *labels can be added easily in the picture as we have done in Example 16. The parsed text is (we write also text positions for clarity):*

```
0  1  2  3 4  5 6  7 8 9  10 11  12 13 14  15  16 17  18 19  20
A | C | G | C G | A C | A C A | C  A | C  G  G | T | G  G | G  T | #
```

The resulting path-compressed tree of all reversed text prefixes ending at phrase boundaries is:



Note: if we wish to use LZ77 instead of LZ78, the trie of phrases can have more than $\mathcal{O}(z)$ nodes (up to $\mathcal{O}(n)$). However, if we have access to the plain text, we can represent the LZ77-trie in $\mathcal{O}(z \log n)$ bits with path compression. The index described in this section therefore works also with LZ77.

Handling primary occurrences We are now going to create a geometric range structure to connect LZ phrases with the text prefixes that precede them. For each phrase starting at position $t > 0$ in T, let i be the lexicographic rank of the (reversed) prefix ending in position $t-1$, and j be the lexicographic rank of the phrase starting in position t. We add a labeled 2D point $\langle\langle i, j\rangle, t\rangle$ to the range structure [16]. Since we store $z-1$ labeled points on the coordinate space $[0, z-1] \times [0, z-1]$ and $z-1$ labels, this structure takes $z \log z(1 + o(1)) + z \log n \in \mathcal{O}(z \log n)$ bits of space. Note that with this structure a range search query is answered in time $\mathcal{O}((occ + 1) \log z)$. Figure 2.7 reports the range structure resulting from the text in the previous examples.

Algorithm 8 shows the full procedure to find all primary occurrences. Here, *rev_tree*, *trie*, and *Geom_4_sided* are the tree of reversed text prefixes ending at phrase boundaries, the trie of phrases, and the 4-sided geometric structure, respectively.

Algorithm 8: $primary(P)$

input : Pattern $P = p_1 \dots p_m \in \Sigma^m$
output: Set of all pairs $\langle l, r\rangle$ such that $P = T[l, \dots, r]$ and $T[l, \dots, r]$ is a primary occurrence of P

```
1 m ← |P|;
2 OCC ← ∅;
3 for k = 1, ..., m do
4     [l^rev, r^rev] ← rev_tree.range(p_k...p_1);
5     [l^fw, r^fw] ← trie.range(p_{k+1} ... p_m);          /* If k = m this is the full range */
6     S ← Geom_4_sided.report([l^rev, r^rev] × [l^fw, r^fw]); /* set of points ⟨⟨i, j⟩, t⟩ in the query
        rectangle */
7     for each ⟨⟨i, j⟩, t⟩ ∈ S do
8         OCC ← OCC ∪ {⟨t − k, (t − k) + m − 1⟩};
9 return OCC;
```

Figure 2.7 reports the search of the pattern CAC with split CA/C. We have to perform a range search for all the splits C/AC, CA/C, and $CAC/$. It follows that the complexity

[16] i.e. we store the point $\langle i, j\rangle$, and keep a vector $LABELS$ of size $z-1$ such that $LABELS[i] = t$

of locating all occ_1 primary occurrences is $\mathcal{O}(m(m + \log z) + occ_1 \log z)$. The only way to count occurrences is to locate them with this procedure, so *count* and *locate* (of primary occurrences) have the same complexity.

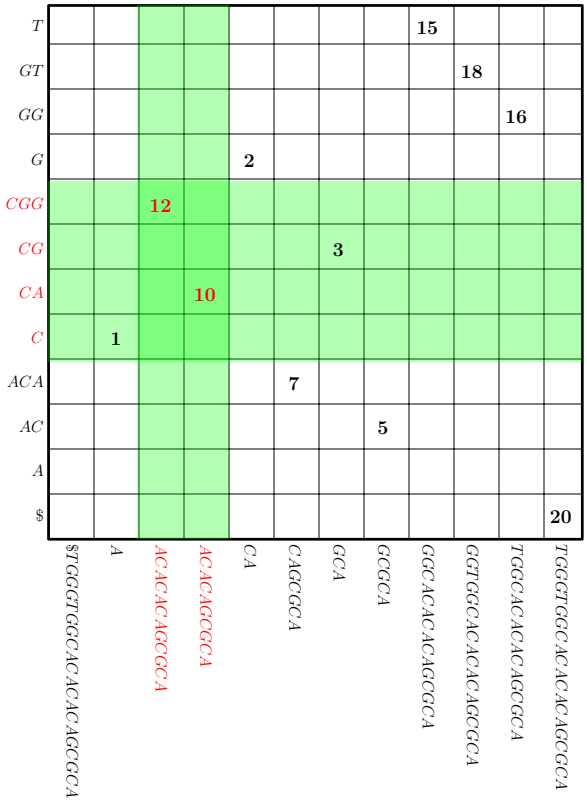


Figure 2.7: Strategy to find primary occurrences. In this example, we search the pattern CAC and we consider the split CA/C. Then, $[l^{rev}, r^{rev}] = [2, 3]$ and $[l^{fw}, r^{fw}] = [4, 7]$. The rectangle $[2, 3] \times [4, 7]$ contains the labels 12 and 10. There are 2 characters (CA) at the left of the pattern split, so the primary occurrences of CAC (with split CA/C) in the text are $12 - 2 = 10$ and $10 - 2 = 8$.

Handling secondary occurrences We can locate secondary occurrences either by using the LZ78 trie [60, 83] or by looking from where such occurrences are copied in the text with the help of a geometric data structure [67]. We are going to follow this second strategy (since it works also with LZ77). The following material is taken from [67]. The next theorem guarantees that if we follow (right-to-left) a chain of substring copies we end up in a primary occurrence:

Theorem 11. *Each text substring P has at least one primary occurrence*

Proof. Consider the leftmost secondary occurrence of P, at position i. Then, by definition, this occurrence is contained in a phrase Z. Z is copied from a previous text position, so P appears in the text also in another position $j < i$. If we assume that the occurrence of P in

position j is secondary we obtain a contradiction because we assumed that the occurrence in i was the leftmost. It follows that the occurrence of P in position j is primary. □

Note that Theorem 11 shows that following right-to-left a chain of copies from a secondary occurrence , we must end up in a primary occurrence. This gives us a strategy to find secondary occurrences: first find all primary occurrences, and then look for phrases entirely copying them. Repeat this recursively (a secondary occurrence can be copied from a secondary occurrence) until no more occurrences are found.

The problem we need to solve is then the following: how do we find which phrases entirely copy $T[i, \ldots, j]$? In [83] this problem is solved using the LZ78 trie itself. Here we discuss the alternative and more general solution of Kreft and Navarro [67] based (again) on range search. This solution works also with LZ77 (unlike the one discussed by Navarro [83]). For each phrase $Z = T[t, ..., t']$ copied from $T[i', ..., j']$ $(t' - t = j' - i')$, we add a labeled 2D point $\langle\langle i', j'\rangle, t\rangle$. Then, to find phrases copying $P = T[i, \ldots, j]$ we query the structure on the rectangle $[0, i] \times [j, \infty]$. This query is called *2-sided* since the query area is delimited by only two parameters (i and j). Figure 2.8 depicts this strategy for a single phrase Z and for a query point (text substring) P.

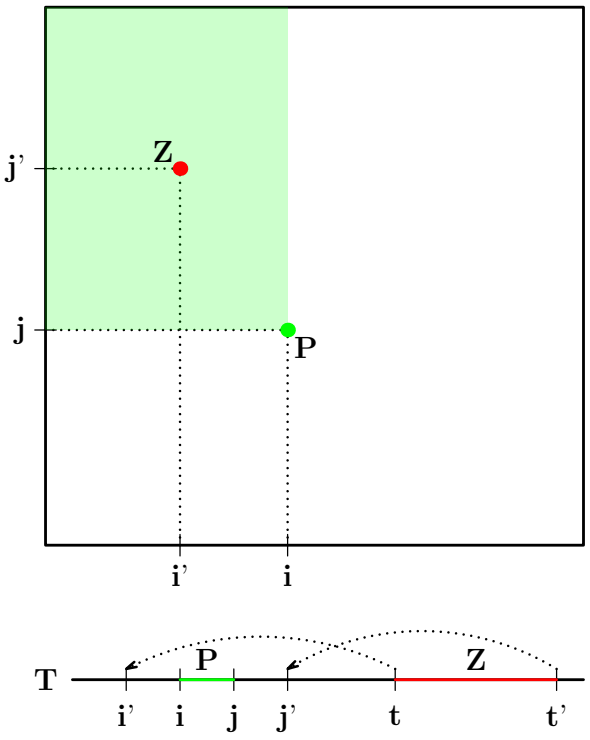


Figure 2.8: Pattern P occurs in $T[i, ..., j]$ and Z entirely copies it. It follows that P occurs also in $T[t + (i - i'), ..., t + (j - i')]$. Note that Z is a point inside the structure, while P is a query point (we do not insert P in the structure)

Note we have two potential problems: (1) some x-coordinates will not have an associated point (because $n \geq z$) and (2) there could be multiple points sharing their x-

coordinate (this happens when 2 or more phrases are copied from the same text position). This means that we need to solve a more general problem than the one discussed in Section 2.3.4. Problem (1) can be solved by re-mapping positions from the domain $[0, n-1]$ to $[0, z-1]$. This can be done in $\mathcal{O}(z \log n)$ bits of space by storing x-coordinates in increasing order in a plain vector V. Then, re-mapping coordinates from $[0, n-1]$ to $[0, z-1]$ requires just two binary searches on V. In problem (2), two (or more) points $\langle i, j' \rangle$ and $\langle i, j'' \rangle$ share the same x-coordinate. In this case we can *insert* a new empty column[17] at position $i+1$ and modify the second point to $\langle i+1, j'' \rangle$. We then mark position i with a '1' and position $i+1$ with a '0' in an auxiliary succinct bitvector B of total size $z + o(z)$ bits. It can be shown that B and V are sufficient to reduce the general problem to the special case (permutation) described in Section 2.3.4.

The procedure to recursively find secondary occurrences is reported as Algorithm 9. Here, *Geom_2_sided* is the geometric 2-sided range structure.

Algorithm 9: $secondary(i, j)$

input : Boundaries $i \leq j$ of a pattern occurrence $T[i, \ldots, j]$
output: Set of pairs $\langle l, r \rangle$ such that $T[i, \ldots, j] = T[l, \ldots, r]$ and $T[l, \ldots, r]$ is in the chain of copies starting in $T[i, \ldots, j]$

```
1 S ← Geom_2_sided.report([0, i] × [j, ∞]);  /* Returns set of labeled points ⟨⟨i', j'⟩, t⟩ in the
    query rectangle */
2 OCC ← ∅;
3 for each ⟨⟨i', j'⟩, t⟩ ∈ S do
4     l ← t + (i − i');
5     r ← t + (j − i');
6     OCC' ← secondary(l, r);
7     OCC ← OCC ∪ OCC' ∪ {⟨l, r⟩};
8 return OCC;
```

While Theorem 11 ensures that we will find *all* secondary occurrences, it does not prove that we will report each of them just *once*. We have therefore to show that any two sets of secondary occurrences obtained with Algorithm 9 starting from two *distinct* primary occurrences are disjoint.

Theorem 12. *Let $T[l, \ldots, r] = T[l', \ldots, r']$, with $l' > l$, be two distinct primary occurrences of a pattern P. Then,*

$$secondary(l, r) \cap secondary(l', r') = \emptyset$$

Proof. In the following we use the notation $T[i, \ldots, j] = T[i', \ldots, j']$ to indicate that $i = i'$ and $j = j'$.

Assume by contradiction that there exists a $\langle l'', r'' \rangle \in secondary(l, r) \cap secondary(l', r')$. It must be the case that $l'' > l' > l$, since phrases are copied from *previous* positions.

Let the notation $T[i, \ldots, j] \rightarrow T[i', \ldots, j']$, with $T[i', \ldots, j']$ secondary occurrence, indicate that text substring $T[i', \ldots, j']$ is copied from $T[i, \ldots, j]$ (i.e. $T[i', \ldots, j']$ is contained in a phrase whose source contains $T[i, \ldots, j]$). The chain of copies starting in $T[l, \ldots, r]$ and ending (by assumption) in $T[l'', r'']$ is

$$T[l, \ldots, r] \rightarrow T[l_1, \ldots, r_1] \rightarrow \cdots \rightarrow T[l_k, \ldots, r_k] = T[l'', r'']$$

[17] i.e. we also increase by 1 x-coordinates of points after column i

Analogously, the chain of copies starting in $T[l', \ldots, r']$ and ending (by assumption) in $T[l'', r'']$ is

$$T[l', \ldots, r'] \to T[l'_1, \ldots, r'_1] \to \cdots \to T[l'_{k'}, \ldots, r'_{k'}] = T[l'', r'']$$

We distinguish two cases:

1. $k = k'$. We prove by induction on k that $l = l'$ (obtaining therefore a contradiction since $l' > l$). If $k = 1$, then the two chains of copies are

   $$T[l, \ldots, r] \to T[l_1, \ldots, r_1] = T[l'', r'']$$

   and

   $$T[l', \ldots, r'] \to T[l'_1, \ldots, r'_1] = T[l'', r'']$$

   By definition of LZ parsing a phrase is copied only from *one* location, therefore, since $T[l_1, \ldots, r_1] = T[l'_1, \ldots, r'_1]$, we obtain $l = l'$ (absurd). If $k > 1$, then the two chains of copies are

   $$T[l, \ldots, r] \to \cdots \to T[l_{k-1}, \ldots, r_{k-1}] \to T[l_k, \ldots, r_k] = T[l'', r'']$$

   and

   $$T[l', \ldots, r'] \to \cdots \to T[l'_{k'-1}, \ldots, r'_{k'-1}] \to T[l'_{k'}, \ldots, r'_{k'}] = T[l'', r'']$$

   Since $T[l_k, \ldots, r_k] = T[l'_{k'}, \ldots, r'_{k'}]$, with the same reasoning used above we obtain $l_{k-1} = l'_{k'-1}$. We can therefore use the inductive hypothesis on the "truncated" chains and obtain again $l = l'$.

2. $k < k'$ (the case $k' < k$ is symmetric). We prove by induction on k that $l = l'_i$ for some $1 \leq i < k'$. This is an absurd because $T[l, r]$ is a primary occurrence, while $T[l'_i, r'_i]$ is a secondary occurrence. If $k = 1$, the two chains of copies are

   $$T[l, \ldots, r] \to T[l_1, \ldots, r_1] = T[l'', r'']$$

   and

   $$T[l', \ldots, r'] \to \cdots \to T[l'_{k'-1}, \ldots, r'_{k'-1}] \to T[l'_{k'}, \ldots, r'_{k'}] = T[l'', r'']$$

   Then, since $T[l_1, \ldots, r_1] = T[l'_{k'}, \ldots, r'_{k'}]$ and, by definition, a LZ phrase has only one source, we get $l = l'_{k'-1}$ (absurd). If $k > 1$, then the two chains of copies are

   $$T[l, \ldots, r] \to \cdots \to T[l_{k-1}, \ldots, r_{k-1}] \to T[l_k, \ldots, r_k] = T[l'', r'']$$

   and

   $$T[l', \ldots, r'] \to \cdots \to T[l'_{k'-1}, \ldots, r'_{k'-1}] \to T[l'_{k'}, \ldots, r'_{k'}] = T[l'', r'']$$

   Since $T[l_k, \ldots, r_k] = T[l'_{k'}, \ldots, r'_{k'}]$, with the same reasoning used above we obtain $l_{k-1} = l'_{k'-1}$. We can therefore use the inductive hypothesis on the "truncated" chains and obtain again an absurd.

□

Combining the two procedures Combining the two procedures is now easy: just find all primary occurrences and then apply procedure *secondary* to each of them. The full procedure for finding pattern occurences with our index is reported as Algorithm 10.

Algorithm 10: *locate_occ*(P)

input : Pattern $P \in \Sigma^m$
output: All pairs $\langle i, j \rangle$ such that $P = T[i, \ldots, j]$

```
1 OCC ← ∅;
2 S ← primary(P);
3 for each ⟨i, j⟩ ∈ S do
4 |   OCC' ← secondary(i, j);
5 |   OCC ← OCC ∪ OCC';
6 OCC ← OCC ∪ S;
7 return OCC;
```

We can state our result:

Theorem 13. *The LZ index we described is a* full-text index *based on LZ78/LZ77 taking*

$$\mathcal{O}(z \log n) + n \log \sigma$$

bits of space and supporting count and locate in $\mathcal{O}(m(m + \log z) + occ \log z)$ *time, occ being the number of occurrences of* P *in* T*. The text is stored in plain format, so extract is supported in optimal time.*

A LZ78 self-index

Actually, if we restrict our attention to LZ78 it is quite easy to turn the LZ full-text index into a *self-index*. The LZ78 trie property we are going to exploit—taken from Navarro [83]—is the following: the LZ78 trie can be augmented with $2z$ pointers to nodes ($2z \log z$ bits) so that it supports the *extraction of any length-L substring* of the *reversed text* in *optimal* $\mathcal{O}(L)$ *time* (given a starting point). Instead of using the text, we can then use the LZ78 trie itself to support path compression in the sparse suffix tree.

Note that:

1. There is a bijection between LZ78 phrases and trie nodes (root excluded)
2. The relation from text positions to trie nodes (root excluded) is a *surjective function*: each trie node corresponds to ≥ 1 text positions and each text position is associated to only one node

We are going to link each phrase (node) with the phrase (node) preceding it in the text. To extract (backwards) a text substring, we then need just to follow parent links from a node to the root, and then jump to a new node (previous phrase) by following previous-phrase links. Let X and Y be two LZ78 phrases (i.e. two nodes of the trie). we add the following edges:

- `Parent`: if Y is child of X in the trie, then we add the edge $\pi(Y) = X$

- `Previous phrase`: if X *immediately* precedes Y in the text then we add the edge $pr(Y) = X$

We need to take care of these particular cases:

- $\pi(root) = root$
- If X is the first text phrase, $pr(X) = NULL$

To simplify the description, we assume that we do not reach $T[0]$ during extraction. Suppose we want to extract (backwards) m text characters starting from text position i. Let:

- X be the *phrase* (i.e. trie node) containing position i
- N be the trie node corresponding to text *position* i

Note that phrases can be identified with an integer in $[0, z-1]$. To start extraction, we just need the coordinate pair $\langle X, N\rangle$. The full procedure is reported as Algorithm 11. Figure 2.9 depicts an example of extraction.

Algorithm 11: $extract(\langle X, N\rangle, m)$

```
1 if m = 0 then
2     return ε;                                          /* return empty string */
3 c ← char(N);                                  /* character stored in node N */
4 if π(N) = root then
5     X ← pr(X);                                        /* jump to previous phrase */
6     N ← X;                          /* next character is the last of previous phrase */
7     return extract(⟨X, N⟩, m − 1) · c;   /* next m − 1 chars concatenated with this char */
8 else
9     return extract(⟨X, π(N)⟩, m − 1) · c;
```

To complete our index, we only need to substitute intervals $[begin, end]$ in the sparse suffix tree with pairs $\langle\langle X, N\rangle, L\rangle$, where:

- X is the phrase (trie node) containing text position $begin$
- N is the trie node corresponding to $T[begin]$
- $L = begin - end + 1$

We finally obtain:

Theorem 14. *The LZ index described in this section is a* self-index *based on LZ78 taking*

$$\mathcal{O}(z \log n)$$

bits of space and supporting count and locate in $\mathcal{O}(m(m + \log z) + occ \log z)$ *time, occ being the number of occurrences of* P *in* T.

extract of arbitrary text substrings requires some other structures not discussed here (we basically need to implement space-efficiently the function that maps text positions to trie nodes).

$$A|C|G|CG|AC|A\overleftarrow{CA|CA|CG}G|T|GG|GT|\$$$

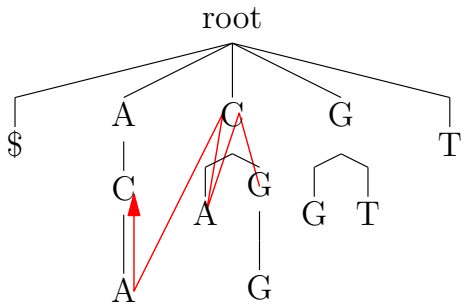


Figure 2.9: Using the LZ78 trie to extract text. We want to extract (backwards) $m = 6$ characters starting from position $i = 13$. The phrase (trie node) containing position i is CGG. Position i corresponds to node CG in the trie. The starting point in the trie is therefore $\langle CGG, CG\rangle$. We climb the trie up to the root, extracting characters GC. At this point, we jump to node $pr(CGG) = CA$. We climb up to the root extracting characters AC, and then jump to node $pr(CA) = ACA$. We extract the remaining 2 characters AC.

A LZ77 self-index

In this paragraph we quickly sketch how to build a LZ77-based self-index. See Kreft and Navarro [67] for more details and a more efficient solution. The only ingredient we need to add to the LZ77 full-text index of Section 2.4.3 is the ability to *extract* text characters from a LZ77-compressed version of the text (we need this to perform path compression on the trees). We can easily support this operation by using the representation of LZ77 as triples $\langle text\ position, length, char\rangle$, plus a vector storing the beginning of LZ phrases. In order to extract $T[i]$, just locate the phrase containing text position i (binary search on the vector storing the beginning of LZ phrases) and find out the position $j < i$ from where $T[i]$ is copied by using the triples representation. We need to recursively repeat this operation at most h times, where h is the *height* of the LZ77 parse (see Definition 11). In this index, all queries are therefore supported with an additional multiplicative factor of $\mathcal{O}(h \log z)$ (h binary searches). The solution presented by Kreft and Navarro [67] is more involved but reduces the time to extract a character to $\mathcal{O}(h)$.

2.5 Online Construction of the Burrows-Wheeler Transform

Most of the results discussed in this thesis rely on a well-known online BWT construction algorithm based on the concept of backward search [12, 76, 77]. The algorithm builds $BWT(T)$ in $\mathcal{O}(n)$ steps by inserting T characters from right to left in a dynamic string data structure. The concept is rather simple: we remind that the backward search algorithm permits to find the lexicographic range of a string among all text suffixes. Now, consider the problem of computing $BWT(cT)$ starting from $BWT(T)$. The idea is to backward-

search the string cT inside $BWT(T)$. The result is an empty range (empty because cT does not appear in T) $[t, t)$, where t is the lexicographic range of cT among all T's suffixes. At this point, all we have to do in $BWT(T)$ is to replace the terminator \$ with the new character c (because now c follows the terminator in the circular text cT) and insert a new terminator at position t (because the new inserted text suffix ends with \$). Most importantly, each character extension requires only a constant number of queries on $BWT(T)$ (we do not actually need to search the whole cT inside it). As depicted in Figure 2.10, the insert position t can be computed by finding the c preceding \$ (one rank operation) and by applying the LF mapping (in the example, $c =' s'$ and $T = ississippi\$$). Figure 2.11 depicts $BWT(cT)$ after the updates.

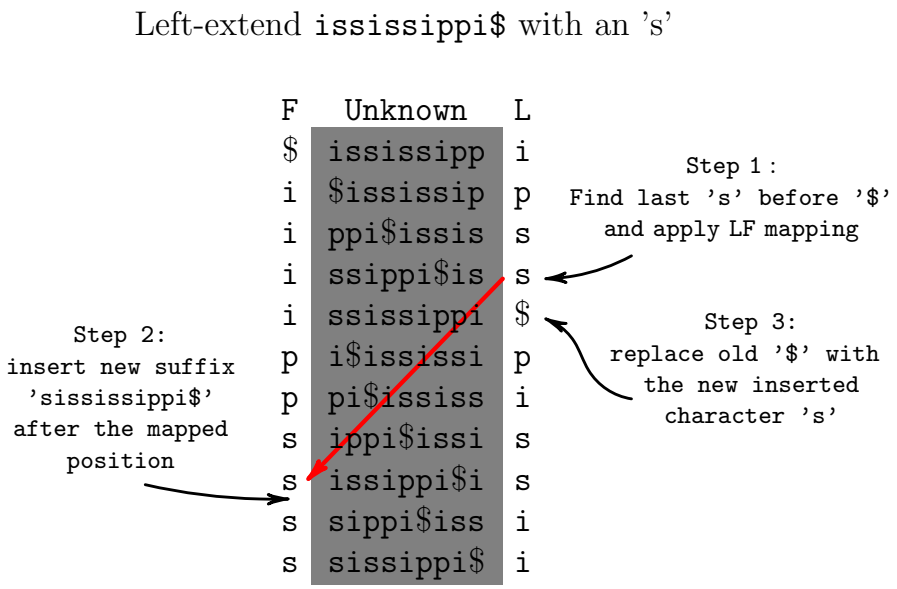


Figure 2.10: Building $BWT(cT)$ starting from $BWT(T)$ (a). We need to find the lexicographic position of the new text suffix (in the example, `sississippi`) among T's suffixes. The insert position t can be found by searching the new text suffix in $BWT(T)$ with backward search. On the BWT, this will result in two updates: the terminator \$ is inserted at position t (because it ends the new suffix), and the old terminator is replaced with the inserted letter (which follows the terminator in the extended text).

Analysis

The update algorithm above described can be used to build $BWT(T)$ in $\mathcal{O}(|T|)$ steps. Note that we need to perform `insert`, `rank`, and `replacement` operations on a string, therefore we need a *dynamic string* data structure. It is easy to implement such a structure by using red-black trees. The idea is to store string characters in the leaves of the tree and augment each internal node with counters storing the total count of each character (in

Left-extend `ississippi$` with an 's'

F	Unknown	L
$	sississipp	i
i	$sississip	p
i	ppi$sissis	s
i	ssippi$sis	s
i	ssissippi$	s
p	i$sississi	p
p	pi$sississ	i
s	ippi$sissi	s
s	issippi$si	s
s	ississippi	$
s	sippi$siss	i
s	sissippi$s	i

Figure 2.11: Building $BWT(cT)$ starting from $BWT(T)$ (b). In yellow, the new text suffix and the new letter appearing in the BWT are highlighted. On the BWT, there are only two updates: $ is replaced with the new letter, and a new terminator is inserted at the position of the new text suffix.

order to implement `rank`) and the size of the substring represented by the node. This approach requires $\mathcal{O}(n\sigma \log n)$ bits of space and is therefore not feasible in practice. One could remove the σ term by employing wavelet trees (see Section 2.3.2); it is easy to show how to reduce insert operations on generic strings to inserts on a binary string by using wavelet trees with only a $\log \sigma$ slowdown in all operations. Space usage could be further improved by packing $\Theta(\log n)$ bitvector's bits in a single tree leaf (therefore reducing the number of leaves to $\mathcal{O}(n/\log n)$). This approach, adopted by Lippert et al. [70], requires $\mathcal{O}(n \log \sigma)$ bits of space and supports all operations on the dynamic string in $\mathcal{O}(\log n \log \sigma)$ time. Finally, one could aim at using only compressed space. The complexity of this approach is deeply influenced by the inherent complexity of dynamic string data structures, which have been proved to have a $\Theta(\log n / \log \log n)$ lower [40] and (amortized) upper [87] bound for queries and updates. In particular, the result in [87] has as direct consequence (clearly mentioned in that paper) that the BWT can be constructed in $nH_k + o(n \log \sigma)$ bits of space and $\mathcal{O}(n \log n / \log \log n)$ worst-case time. In Section 3.2 we show that in the average case we can beat the lower bound and build the BWT in just $\mathcal{O}((H_k + 1))$ time per character while using compressed working space. In Section 3.3 we consider how to build a run-length compressed BWT.

3

Computing the BWT in Compressed Working Space

In many applications it is often not feasible to load the whole dataset in RAM in order to compress it. Ideally, one would aim at using an amount of working space proportional to the output of the compressor while performing compression.

This chapter is devoted to the problem of building a compressed Burrows-Wheeler Transform while using a compressed amount of working space. The content of the chapter is based on papers `(i)`, `(iii)`, and `(iv)`. We remind (see Section 1.4) that r denotes the maximum between the numbers of runs in the Burrows-Wheeler transforms of the text and of the reversed text. We provide algorithms to compute:

- The Burrows-Wheeler transform using high-order compressed working space. The algorithm runs in average $\mathcal{O}((H_k + 1))$ time per character on nearly-uniform texts (e.g. DNA)
- The Burrows-Wheeler transform in $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r)$ words of working space

Our solutions are based on the online BWT construction algorithm described in Section 2.5.

3.1 Related Work

Building the Burrows-Wheeler transform is still a bottleneck in many applications, including text compression and the construction of compressed self-indexes. To date, none of the solutions in the literature is able to guarantee simultaneously both $\mathcal{O}(n)$ construction time and $nH_k + o(n \log \sigma)$ bits of space. The most time-efficient (also in practice) $\mathcal{O}(n)$ techniques to date, rely on the construction of suffix arrays (see for example [90]), which however require $\mathcal{O}(n \log n)$ bits of space. Very recently, it has been shown that the space requirements can be reduced to $\mathcal{O}(n \log \sigma)$, while maintaining the optimal construction time $\mathcal{O}(n)$ [4, 6, 82]. Despite compact space being asymptotically optimal in the uncompressed domain, the hidden constant in practice could be high and makes this kind of algorithms impractical, especially for large texts (e.g. big genomes). This problem motivates the search for more space-efficient algorithms, being able either to work in external memory or to exploit the compressibility of the text in order to reduce RAM requirements. External and semi-external solutions on genomic data include [8], which requires about 1 byte in RAM per input symbol and works in linear-time, and [119], which requires about

2 bits per input symbol in RAM and works in $\mathcal{O}(n \log^2 \log n)$ average time. Of particular interest is the more general implementation described by Ferragina et al. [29], which requires a *constant* (i.e. text independent) amount of RAM working space. Building a compressed BWT is another common solution in order to save working space. Usually, this is done by backward-inserting text characters in a compressed dynamic string data structure with the algorithm described in Section 2.5. In [87], the authors follow this approach and show that the BWT can be constructed in $nH_k + \mathcal{O}(\sigma \log n) + o(n \log \sigma)$ bits of space and $\mathcal{O}(n \log n / \log \log n)$ worst-case time. Table 3.1 summarizes the above discussed results, comparing them with the bounds described in this chapter.

Space	avg-case time	worst-case time	ref
$\mathcal{O}(n \log n)$	-	$\mathcal{O}(n)$	[90]
$\mathcal{O}(n \log \sigma)$	-	$\mathcal{O}(n)$	[4]
$nH_k + \mathcal{O}(\sigma \log n) + o(n \log \sigma)$	-	$\mathcal{O}(n \log n / \log \log n)$	[87]
$n(H_k + 1)(1 + o(1)) + \mathcal{O}(\sigma \log n)$	$\mathcal{O}(n(H_k + 1))$	$\mathcal{O}\left(n(H_k + 1)(\log n / \log \log n)^2\right)$	3.2
$\mathcal{O}(r)$ words	-	$\mathcal{O}(n \log r)$	3.3

Table 3.1: Comparison among some of the most interesting space-time tradeoffs presented in the literature. If not specified, space is measured in bits.

3.2 High-Order Compressed Working Space

The content of this section is taken from paper (i). Our algorithm relies on the fact that BWT characters can be partitioned in *contexts*. On the grounds of this observation we generalize the classic backward insertion algorithm to work with multiple dynamic strings (one per context), instead of only one for the whole BWT. The classic algorithm becomes, then, a particular case of ours when the context length k is 0. This strategy reduces considerably the average size of internal data structures, thus leading to better performance. We call our algorithm cw-bwt (context-wise BWT). In this section we prove the following:

Theorem 15. *We can build the BWT of a length-n text in average $\mathcal{O}(n(H_k + 1))$ time using $n(H_k + 1)(1 + o(1)) + \mathcal{O}(\sigma \log n)$ bits of space, where $k = \log_\sigma(n / \log^2 n) - 1$*

In the worst case, our structure is slower by a factor of $(\log n / \log \log n)^2$; as we show in Chapter 7, however, practical texts behave nearly-uniformly with our structures as our algorithm runs in average $\mathcal{O}((H_k + 1))$ time per character on many texts of practical interest.

In Figure 3.1 we contextualize Theorem 15 in our framework of algorithmic tools (Figure 1.1).

3.2.1 Data Structures

We remind the reader that the Burrows-Wheeler transform of $T\$$ can be obtained by sorting all circular permutations of $T\$$, representing them in "conceptual" matrix M (see Table 3.2), and then taking the last column $M^n = L$ of M (the first column will be denoted $M^0 = F$), where $M^i,\ 0 \leq i \leq n$ is the i-th column of M. Notice that length-k

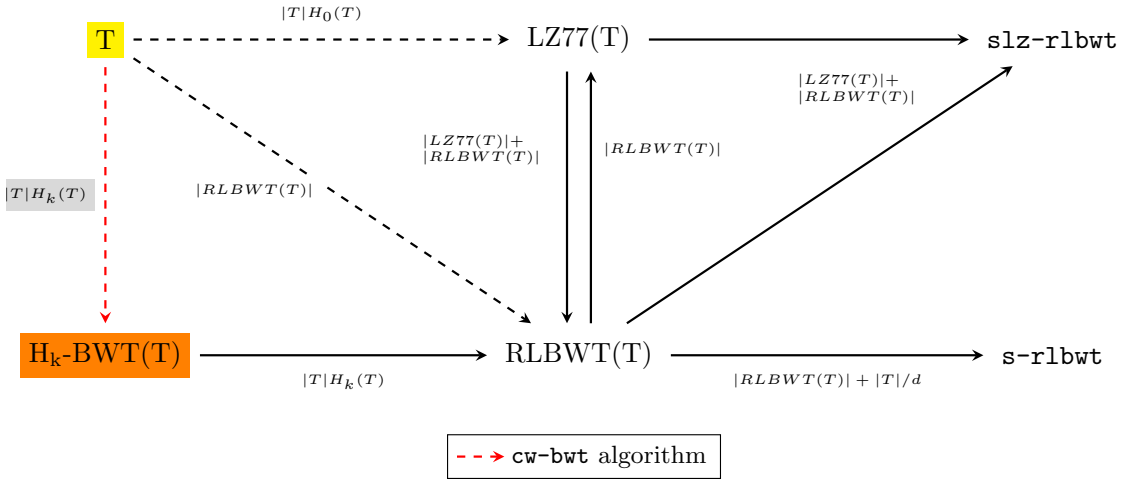


Figure 3.1: Our `cw-bwt` algorithm builds a high-order compressed BWT

contexts appear lexicographically sorted in the first k columns of M, and thus they induce a partition of the BWT rows (as depicted in Table 3.2).

$	m	i	s	s	i	s	s	i	p	p	i
i	$	m	i	s	s	i	s	s	i	p	p
i	p	p	i	$	m	i	s	s	i	s	s
i	s	s	i	p	p	i	$	m	i	s	s
i	s	s	i	s	s	i	p	p	i	$	m
m	i	s	s	i	s	s	i	p	p	i	$
p	i	$	m	i	s	s	i	s	s	i	p
p	p	i	$	m	i	s	s	i	s	s	i
s	i	p	p	i	$	m	i	s	s	i	s
s	i	s	s	i	p	p	i	$	m	i	s
s	s	i	p	p	i	$	m	i	s	s	i
s	s	i	s	s	i	p	p	i	$	m	i

Table 3.2: Conceptual BWT matrix M of the text `mississippi$`. Last column is the BWT (`ipssm$pissii`). There are 9 different contexts of length $k = 2$, corresponding to a partitioning of the BWT in 9 substrings. With our strategy, we will keep one dynamic string data structure for each of these substrings.

We remind also that a fundamental property of the BWT matrix is the *LF property*: the i-th occurrence of c ($c \in \Sigma \cup \{\$\}$) in the last column corresponds to the i-th occurrence of c in the first column (i.e. they represent the same text position). This property can be generalized. Given a h-context $s \in (\Sigma \cup \{\$\})^h$, for some h, let $\mathcal{C}^s(M^i)$ be the class of the partition of M^i induced by s, i.e. the substring $M^i[l, \ldots, r]$ such that all and only BWT rows $l, \ldots, r$ are prefixed by s.

Lemma 1. *(Context-wise LF property) If $T[j]$ is i-th occurrence of c in $\mathcal{C}^s(L)$, then $T[j]$*

is the i-th occurrence of c in $\mathcal{C}^{cs}(F)$.

The classical LF-property is the special case for $h = 0$ of the context-wise LF property. Our main structure will be, essentially, a de Bruijn automaton: a labeled subgraph of a de Bruijn graph [24], having k-contexts as states. For each automaton's state s we will store a compressed dynamic string encoding the class $\mathcal{C}^s(L)$ and a partial sum data structure encoding the class $\mathcal{C}^s(M^k)$. These data structures will be better specified below. Then, the main algorithm will proceed by reading text's characters (right to left) while navigating automaton's states and updating the corresponding dynamic strings and partial sums structures. Correctness follows from Lemma 1. Figure 3.2 depicts the data structures used in our algorithm.

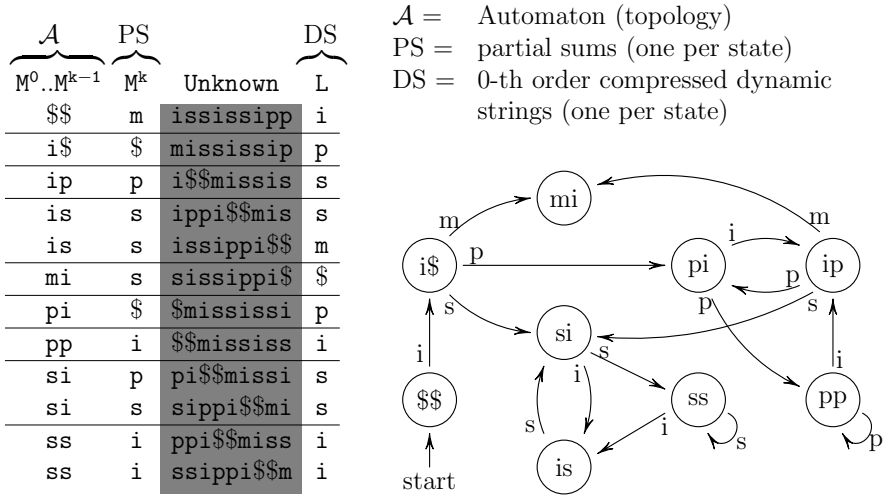


Figure 3.2: Data structures used in the cw-bwt algorithm. A de Bruijn automaton is used to navigate BWT contexts. Every automaton state $w \in \Sigma^k$ (i.e. a k-mer) is associated with a partial sum structure PS memorizing the substring $\mathcal{C}^w(M^k)$ of the k-th BWT column, and with a 0-th order compressed dynamic string memorizing the BWT substring $\mathcal{C}^w(L)$.

de Bruijn Automata

With the term *de Bruijn automaton* we indicate a labeled subgraph of a de Bruijn graph [24], having k-mers appearing in $T\$^k$ as nodes. More in detail, our de Bruijn automaton is $\mathcal{A} = \langle Q, \Sigma, \delta, \$^k \rangle$, where $Q = \{q \mid q \text{ is a } k\text{-mer in } T\$^k\}$ is the set of states, Σ is the set of input symbols, $\delta : Q \times \Sigma \to Q$ is the transition function defined as $\delta(u, c) = v$ if and only if $v = cu[0, ..., k-2]$, and $\$^k$ is the start state. Accepting states are not relevant for our application, so we omit them. Using array indexes as k-digit integers in base σ, the representation of the automaton will turn out to be implicit in our data structures. This choice gives the additional benefit that automaton's states can be visited in lexicographic order without overhead, a feature that we will use in our algorithm.

In order to refer to the structures associated to each automaton state, we will use the following notation. Each automaton's state $s \in Q$ will carry a dynamic string de-

noted with $\mathcal{A}[s].DS$ and a partial sum data structure denoted with $\mathcal{A}[s].PS$. The function $\mathcal{A}.GOTO(s,c) \in Q,\ s \in Q,\ c \in \Sigma$, encodes the automaton's transition function: $\mathcal{A}.GOTO(s,c) = \delta(s,c)$.

Succinct Dynamic Bitvectors

The problem of designing a lightweight bitvector offering efficient query and update operations has been extensively discussed in the literature [45, 49, 87, 88]. Let u be the bitvector length. Given the lower bound $\Omega(\log u/\log\log u)$ on the maximum of update and queries [40] and the most recent optimal-time and compressed $\mathcal{O}(uH_0)$ space solutions [87,88], this problem can be considered essentially solved for a general bitvector size u. However, under the word RAM model with word size w, better solutions can be found for a small enough bitvector size u (e.g. $u \in \mathcal{O}(poly(w))$). In the following we assume the maximum bitvector length u is known a priori (so that the size $\log u$ of the counters we use does not change with time). This will be our case, as this value will be known at bitvector construction time.

The core of our bitvector data structure is a packed B-tree. Each leaf of the tree stores $W = p \cdot w$ bits, for a suitable integer $p > 0$. Note that, within W bits, we can pack $d = \lfloor W/\log u \rfloor$ counters of $\log u$-bits each. We use d as the maximum number of children per internal node. At this point, we assign to each internal node $3pw = 3W$ bits and we store, for each node's children, one rank counter (number of bits set), one size counter (number of bits), and a pointer to the children. Note that d could be asymptotically larger than the number $\mathcal{O}(p)$ of words in each node; however, by using standard bit-parallelism techniques (typically employed in packed B-trees), we will spend only $\mathcal{O}(p)$ time per node during tree navigation (essentially, bit-parallelism techniques allow us to perform constant-time additions and subtractions between all integers packed in two words). With h we denote the height of the tree. Access and rank operations are implemented in $\mathcal{O}(h \cdot p)$ time: size counters guide the search from the root to the leaves, and rank counters give partial rank information (of the subtrees) while searching the leaf ($\mathcal{O}(p)$ operations for each node on the path). The main novelty resides in the insertion algorithm, which is studied to maximize leaf usage and minimize expensive re-arrangement operations. While inserting a bit, if a leaf/node is not full, then we simply insert the bit/key in the right place, updating accordingly the counters. If a leaf/node is full, 4 cases can appear. Let $b = \sqrt{d}$. If a leaf is full, then we count the number m of bits in b adjacent leaves, including the current full leaf (apart from being adjacent, the way leaves are chosen is not relevant for the analysis). If $m > b(W-b)$, then we create a new leaf and redistribute uniformly the m bits in the resulting $b+1$ leaves. If $m \leq b(W-b)$, then we redistribute uniformly the m bits among the b leaves without creating a new one. If an internal node is full, then we choose b adjacent nodes (included the current full node) and we count the total number m of children. If $\lfloor m/(b+1) \rfloor \geq b$, then we create a new internal adjacent node and we uniformly redistribute the children in the resulting $b+1$ nodes. If $\lfloor m/(b+1) \rfloor < b$, then we uniformly redistribute the children among the b adjacent nodes. All the above redistributions can be implemented in $\mathcal{O}(p \cdot b)$ time using shifts and masks. In all 4 the cases, it can be easily shown that after a redistribution the number of free bits/positions in the manipulated leaves/nodes is always $\Omega(b)$: as a consequence, if a redistribution takes place, then at least b "easy" ($\mathcal{O}(h \cdot p)$ time) insertions have been made beforehand, resulting in $\mathcal{O}(h \cdot p)$ amortized cost for the insertion. Due to the internal node redistribution policy,

the minimum number of children per node is b. It follows that the height of the tree is $h \in \mathcal{O}(\log_b(u))$. In order to keep the space always succinct, we choose $p = \log u / \log w$. We obtain $d = \mathcal{O}(w/\log w)$ and $b = \mathcal{O}(\sqrt{w/\log w})$. The minimum number of used bits per leaf is (following the redistribution policy) $b(W-b)/(b+1)$. From this fact, it can be shown that the maximum overhead (total number of bits allocated in the leaves but not used) is of $\mathcal{O}(u/b)$ bits. Moreover, the maximum number of leaves is $n_L \in \mathcal{O}(u/W)$, thus the maximum number of internal nodes is $(n_L - 1)/(b-1) \in \mathcal{O}(n_L/b) = \mathcal{O}(u/(bW))$, totalling $W \cdot \mathcal{O}(u/(bW)) = \mathcal{O}(u/b)$ bits of space occupancy for the internal nodes. The extra space required by the tree is thus $o(u)$, so the whole structure occupies $u + o(u)$ bits of space. Finally, from the particular value chosen for p, it can be easily shown (see above) that all operations have cost $\mathcal{O}(h \cdot p) = \mathcal{O}((\log_w u)^2)$ (in the amortized sense for insertion). This cost is $\mathcal{O}(1)$ or $\mathcal{O}((w/\log w)^2)$ if the bitvector size is $u \in \mathcal{O}(poly(w))$ or $u \in \mathcal{O}(2^w)$, respectively.

Compressed Dynamic Strings

Given the bitvector data structure discussed above, a generalization to dynamic strings can be easily made using wavelet trees (see Section 2.3.2 and [84] for a complete survey on the subject). Assuming that the frequency of each character to be inserted in the string is known beforehand, we implement this structure using a Huffman-shaped wavelet tree. Huffman encoding requires, on average, at most $H_0 + 1$ bits per symbol. The total space of the structure is bounded by $n(H_0+1)(1+o(1)) + \mathcal{O}(\sigma \log n)$ bits [51,84], where the last term comes from the codebook. Queries and update operations are supported in average $\mathcal{O}(H_0+1)$ or $\mathcal{O}((H_0+1)(w/\log w)^2)$ time if the string size is $u \in \mathcal{O}(poly(w))$ or $u \in \mathcal{O}(2^w)$, respectively.

Partial Sums

Our algorithm will require, for each automaton's state s, one partial sum structure of length σ encoding the class $\mathcal{C}^s(M^k)$. A partial sum data structure PS of length j is a list of values $PS[0], ..., PS[j-1]$ offering efficient partial sum and update queries. With $PS.sum(i)$ we will denote the quantity $\sum_{k=0}^{i-1} PS[k]$, and $PS.increment(i)$ will denote the operation $PS[i] \leftarrow PS[i] + 1$.

Efficient solutions offering optimal worst-case space and time bounds, appeared in the literature [104]. See also Sections 3.3.1 and 6.2 for another partial sum structure with different space-time bounds. For self-containedness, here we describe a simple structure based on packed B-trees supporting constant-time operations for the particular case where $\sigma \in \mathcal{O}(poly(w))$. The main idea is to use a packed B-tree to store temporary partial sums information, plus an array $S[0, ..., \sigma - 1]$ initialized at $S[i] = 0,\ 0 \leq i < \sigma$. The packed B-tree has σ leaves (each storing a counter) and internal nodes store the partial sums of the corresponding subtrees. Each counter in the tree is composed by $\log \sigma = \mathcal{O}(\log(w))$ bits, so the height of the tree is $\log_{w/\log w} \sigma \in \mathcal{O}(\log_{w/\log w} poly(w)) = \mathcal{O}(1)$. Updates are implemented as follows. When incrementing a counter (i.e. $PS.increment(i)$), the packed B-tree is updated accordingly (i.e. by incrementing counters from the i-th leaf up to the root). At steps of σ *increment* operations, $S[i]$ is incremented with the content of the i-th leaf ($0 \leq i < \sigma$) of the packed B-tree, and the tree is re-initialized (i.e. each counter is reset to 0). Since a single update on the packed B-tree takes constant time and

the tree is re-built every σ operations, updates on the whole structure PS take constant amortized time. Finally, a query $PS.sum(i)$ is implemented in $\mathcal{O}(1)$ time by returning the sum between $S[i]$ and the i-th partial sum stored in the packed B-tree. Given that each counter will store a number less than or equal to n (the text length), the described data structure occupies $\mathcal{O}(\sigma \log n)$ bits of space.

3.2.2 Cw-bwt Algorithm

Here we describe how to improve the BWT construction algorithm described in Section 2.5 by allowing it to work *context-wise*. We name our algorithm *cw-bwt*.

First of all notice that, since we partition the BWT using length-k contexts and each partition is Huffman-compressed, as a by-product we obtain that, overall, the dynamic string data structures require globally $n(H_k + 1)(1 + o(1))$ bits of space in memory [80] (excluding codebooks sizes, see below). The number of states of the automaton is bounded by $|Q| \leq \sigma^k + k \in \mathcal{O}(\sigma^k)$. For each automaton state we store a partial sum data structure of size $\mathcal{O}(\sigma \log n)$ bits. This is, asymptotically, the same space required to store a codebook. The choice $k = \log_\sigma \left(n/\sigma \log^2 n\right) = \log_\sigma(n/\log^2 n) - 1$ results in a total space occupancy of all the above discussed structures of $\mathcal{O}(\sigma \log n)\mathcal{O}(\sigma^k) = \mathcal{O}\left(\sigma \log n \cdot n/(\sigma \log^2 n)\right) = \mathcal{O}(n/\log n) = o(n)$ bits. Note that, for big alphabets, the formula we used for k yields $k = 0$; in this case, it is not true that $\mathcal{O}(\sigma \log n)\mathcal{O}(\sigma^k) = o(n)$ (i.e. $\mathcal{O}(\sigma \log n)$ gets absorbed in $o(n)$ only if $\log_\sigma(n/\log^2 n) \geq 1$). We therefore need to account also for an additional $\mathcal{O}(\sigma \log n)$-bits overhead to include also the big alphabet case. Summing up, the total space occupancy of the cw-bwt algorithm is $n(H_k + 1)(1 + o(1)) + \mathcal{O}(\sigma \log n)$ bits.

Our algorithm is reported as Algorithm 1. See below for a detailed discussion of the pseudocode.

In line 3 the de Bruijn automaton is constructed and data structures are initialized. As mentioned in Section 3.2.1, a simple direct-hashing strategy permits to perform the automaton construction implicitly and with no overhead. The initialization of dynamic string data structures requires the frequency of each character to be computed for each class $\mathcal{C}^s(L)$ (for Huffman encoding). This step can be easily done in linear $\mathcal{O}(n)$ time and $\mathcal{O}(\sigma^k \sigma \log n) = o(n)$ bits of space using, again, direct hashing.

The *for* loop in line 6 scans backwards all text characters, starting from the rightmost. Variables *head* and *tail* at lines 7 and 8 store the current text character and the rightmost symbol of the current context, respectively. t is the position where *head* has to be inserted in the current state. The new automaton state s' is computed (line 9) by appending *head* at the beginning of the current state and by removing *tail* from its rightmost end. The subsequent 4 lines represent the core of our algorithm. First of all, the current text character $head = T[i]$ is inserted at position t in the dynamic string associated with the class $\mathcal{C}^s(L)$ (line 10). In the BWT matrix, this operation corresponds to the substitution of the terminator \$ character with *head* (notice that \$ is not explicitly inserted at each step: we just remember its coordinates $\langle s, t \rangle$). The next operations correspond to the insertion in the BWT matrix of the current text suffix T_i, having as prefix $head \cdot s = s' \cdot tail$ and ending with \$. Since we will need information about the first $k + 1$ columns of M (see Lemma 1), we need to keep track of the fact that in $\mathcal{C}^{s'}(M^k)$ a new symbol *tail* has been added. Since symbols in $\mathcal{C}^{s'}(M^k)$ appear in lexicographic order, this task is accomplished simply by incrementing a partial sum counter (line 11). In line 12 the new position of \$ in $\mathcal{C}^{s'}(L)$ is computed (remember that \$ is not explicitly inserted). The operation

Algorithm 12: cw-bwt(T)

input : Text $T \in \Sigma^n$, without terminator appended at the end.
output: BWT of $T\$$.

```
1  n ← |T|;
2  k ← max(⌈log_σ(n/log² n) − 1⌉, 0);                          /* optimal k */
3  A ← init_automaton(T, k);          /* init automaton and data structures */
4  s ← $^k;                                                    /* current state */
5  t ← 0;                    /* position of the insertion in current state */
6  for i = n − 1 downto 0 do
7      head ← T[i];                        /* symbol entering in the context */
8      tail ← s[k − 1];                   /* symbol exiting from the context */
9      s' ← A.GOTO(s, head);                                   /* next state */
10     A[s].DS.insert(head, t);                 /* insert current character */
11     A[s'].PS.increment(tail);                    /* update partial sums */
12     t ← A[s'].PS.sum(tail) + A[s].DS.rank(head, t);          /* update t */
13     s ← s';                                                 /* update state */
14 A[s].DS.insert($, t);                                  /* insert terminator */
15 BWT ← ε;                                                  /* the BWT of T$ */
16 for s ∈ Q in lexicographic order do
17     BWT.append(A[s].DS);          /* append dynamic strings to BWT */
18 return BWT;
```

$\mathcal{A}[s].DS.rank(head, t)$ returns the number of characters equal to *head* before the position that contained \$. Since we are computing this value in $\mathcal{C}^s(L)$, this is the number of text suffixes starting with $head \cdot s$ and lexicographically smaller than the current text suffix T_i. Lemma 1 implies that, in order to compute the new position of \$ in $\mathcal{C}^{s'}(L)$ (i.e. the lexicographic position of the new text suffix in the BWT matrix), we need to add to this value the number of characters smaller than *tail* in $\mathcal{C}^{s'}(M^k)$, i.e. $\mathcal{A}[s'].PS.sum(tail)$ (line 12).

As mentioned above, we never explicitly insert the terminator character \$ during construction. For this reason, at the end of the first *for* loop, \$ is explicitly inserted (line 14). Finally, the *for* loop at line 16 scans lexicographically the automaton states in order to reconstruct the BWT in the correct order. Notice that states can be scanned in lexicographic order with no overhead if the automaton has been implemented using direct hashing, as described in Section 3.2.1. Moreover, in this step the BWT can be stored directly to disk, for no additional RAM consumption. The operation $BWT.append(\mathcal{A}[s].DS)$ at line 17 is implemented by appending the characters of $\mathcal{A}[s].DS$ one by one to the string BWT ($|A[s].DS|$ access operations).

Time Complexity

The most expensive operations in the *for* loops are those at lines 10, 12, and 17 (insert, rank, and access, respectively). All other operations have cost $\mathcal{O}(1)$ (see Section 3.2.1). Assuming a uniform text distribution, the expected length of each dynamic string is $\mathcal{O}(n/\sigma^k) = \mathcal{O}(\sigma \log^2 n)$. This value, under the other two assumptions $\sigma \in \mathcal{O}(poly(w))$ and $w \in \Theta(\log n)$, is equal to $\mathcal{O}(polylog(n)) = \mathcal{O}(poly(w))$. These observations imply that the amortized cost of queries/updates on the dynamic strings is of $\mathcal{O}(H_k + 1)$. Theorem 15 follows.

The worst case scenario, on the other hand, is represented by a highly repetitive text T in which one or more k-contexts appear $\Theta(n)$ times. This results in the length of the corresponding dynamic strings being $\Theta(n) = \Theta(2^w)$, thus (see Section 3.2.1) in $\mathcal{O}((H_k + 1)(\log n/\log\log n)^2)$ cost for queries/updates. The following holds:

Theorem 16. *The cw-bwt algorithm builds the BWT of a length n text in worst-case time $\mathcal{O}(n(H_k+1)(\log n/\log\log n)^2)$ using $n(H_k+1)(1+o(1))+\mathcal{O}(\sigma \log n)$ bits of space, where $k = \log_\sigma(n/\log^2 n) - 1$*

Implementation

We implemented `cw-bwt` in two versions: the first, available in the `BWTIL` library [96], uses the succinct bitvector implemented by Nicola Gigante [42]. This bitvector is a direct implementation of the structure described in Section 3.2.1, and guarantees constant-time operations on bit sequences of size $w^{\mathcal{O}(1)}$. The resulting `cw-bwt` implementation runs in $\mathcal{O}((H_k + 1))$ time per character under the assumption of near-uniform text distribution (Theorem 15). See Chapter 7 for an experimental demonstration of this statement. The second `cw-bwt` version we implemented uses a simpler bitvector supporting logarithmic queries and can be found in the `DYNAMIC` library [97]. On moderately big inputs, this version runs faster than the first one. See Section 6.4.2 for a description and theoretical analysis of this alternative implementation, and Chapter 7 for its experimental evaluation.

3.3 Run-Length Compressed Working Space

The generality of the online BWT construction algorithm described in Section 2.5 leaves full freedom in the choice of the data structure/compression scheme used to represent the dynamic BWT. In this section, we focus on *run-length encoding*. We first describe a reduction from the dynamic gap-encoded bitvector problem to the *Searchable Partial Sums with inserts* (SPSI) problem (see below for a definition). This structure can be plugged in the run-length dynamic string of Section 2.3.3. By plugging this run-length string in the algorithm of Section 2.5, we will obtain an algorithm to build the BWT in space proportional to the number r of its equal-letter runs. This data structure will be used in Section 4.3 in order to compute the LZ77 parsing in $\mathcal{O}(r)$ words of space.

In Figure 3.3 we contextualize the result presented in this section in our framework of algorithmic tools (Figure 1.1).

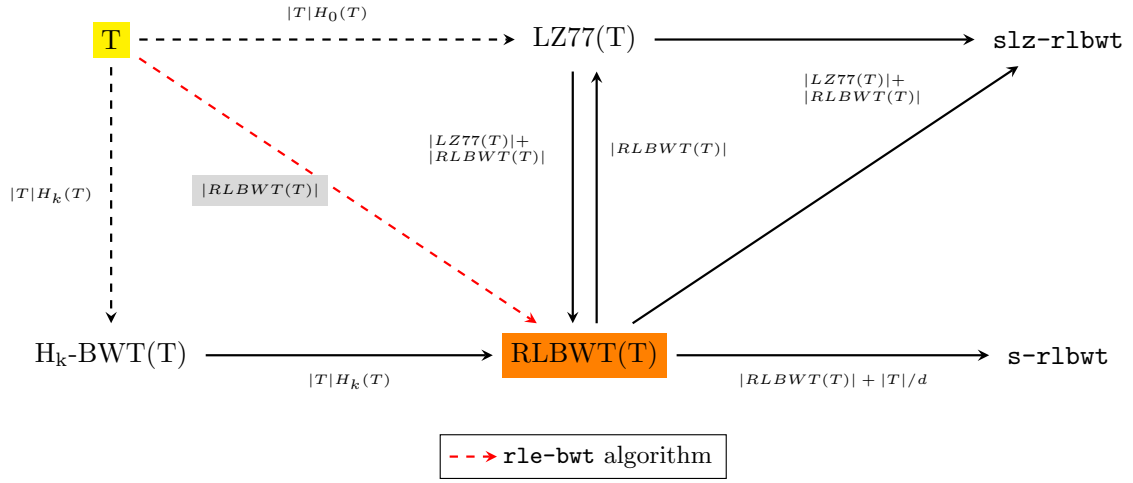


Figure 3.3: Our `rle-bwt` algorithm builds a run-length compressed BWT

3.3.1 The Searchable Partial Sums with Indels Problem

We start by describing a simple and self-contained compact-space solution to the *Searchable Partial Sums with Indels* (SPSI) problem. The SPSI asks for a data structure PS to maintain a sequence $s_1, \ldots, s_m$ of non-negative k-bits integers supporting the following operations:

- `PS.sum`$(i) = \sum_{j=1}^{i} s_j$;
- `PS.search`(x) is the smallest i such that $\sum_{j=1}^{i} s_j > x$;
- `PS.update`(i, δ): update s_i to $s_i + \delta$. δ can be negative as long as $s_i + \delta \geq 0$;
- `PS.insert`(i): insert 0 between s_{i-1} and s_i (if $i = 0$, insert in first position).

We will not need *delete* operations, so we do not consider them here. Here we describe a simple solution taking $\mathcal{O}(m)$ words of space. In Section 6.2 we will describe a more space-efficient and practical solution that has been implemented in our `DYNAMIC` C++ library.

The main idea is to employ red-black trees (RBT) to represent the dynamic sequence of integers. We store $s_1, \ldots, s_m$ in the leaves of a RBT and we store in each internal node of the tree the number of nodes and partial sum of its subtrees. `sum` and `search` queries can then be implemented with a traversal of the tree from the root to the target leaf. `update` queries require finding the integer (leaf) of interest and then updating $\mathcal{O}(\log m)$ partial sums while climbing the tree from the leaf to the root. Finally, `insert` queries require finding an integer (leaf) s_i immediately preceding or following the insert position, substituting it with an internal node with two children leaves s_i and 0 (the order depending on the insert position—before or after s_i), incrementing by one $\mathcal{O}(\log m)$ subtree-size counters while climbing the tree up to the root, and applying the RBT update rules. This

last step requires the modification of $\mathcal{O}(\log m)$ counters (subtree-size/partial sum) if RBT rotations are involved. All operations take $\mathcal{O}(\log m)$ time.

3.3.2 Dynamic Gap-Encoded Bitvectors

Hence, a length-n bitvector $B = 0^{s_1-1}10^{s_2-1}1\dots0^{s_m-1}1$ ($s_i > 0$) can be encoded in $\mathcal{O}(m)$ words of space with a partial sum PS on the sequence $s_1, \dots, s_m$. See [88] for a similar reduction. We need to show how to answer the following queries on B: $\texttt{B}[i]$ (`access`), $\texttt{B.rank}_1(i)$, $\texttt{B.select}_1(i)$, $\texttt{B.insert}_\texttt{b}(i)$ (insert bit $b \in \{0,1\}$ between positions $i-1$ and i), and $\texttt{B.delete}_0(i)$, where $\texttt{B}[i] = 0$ (delete $\texttt{B}[i]$). It is easy to see that $\texttt{rank}_1$, `access` and $\texttt{select}_1$ operations on B reduce to `access`, `search`, and `sum` operations on PS, respectively. $\texttt{B.delete}_0(i)$ requires just a search and an update on PS (decrementing by one a counter). To support `insert` on B, we can operate as follows: $\texttt{B.insert}_0(i)$, $i > 0$ requires incrementing by one a counter and is implemented as $\texttt{PS.update}(\texttt{PS.search}(i), 1)$. $\texttt{B.insert}_1(0)$ is implemented with $\texttt{PS.insert}(0)$ followed by $\texttt{PS.update}(0, 1)$. $\texttt{B.insert}_1(i)$, $i > 0$, "splits" an integer into two integers: let $j = \texttt{PS.search}(i)$ and $\delta = \texttt{PS.sum}(j) - i$. We first decrease s_j with $\texttt{PS.update}(j, -\delta)$. Then, we insert a new integer $\delta + 1$ with $\texttt{PS.insert}(j + 1)$ and $\texttt{PS.update}(j + 1, \delta + 1)$. All operations are supported in $\mathcal{O}(\log m)$ time.

3.3.3 A Dynamic Run-Length BWT

Let $S \in \Sigma^n$. To implement the run-length string $RLE(S)$ of Section 2.3.3, we combine the above-described dynamic gap-encoded bitvector with a dynamic string to encode component H (i.e. run heads). For H we can use the result in [87], guaranteeing $\mathcal{O}(r_S)$ words of space. Note that in Navarro's work [87] there is an extra $\mathcal{O}(\sigma \log r_S)$ bits spatial term amounting, in our case, to $\mathcal{O}(r_S)$ words, since $\sigma \leq r_S \leq n$. This structure supports $\mathcal{O}(\log r_S)$-time `rank`, `select`, `access`, and `insert`. Note that this is not the strategy that has been implemented in practice in our library: in Section 6.3.2 we represent H with a wavelet tree built upon dynamic succinct bitvectors. We obtain:

Lemma 2. *Our dynamic run-length string $RLE(S)$ takes $\mathcal{O}(r_S)$ words of space and supports `rank`, `access`, and `insert` operations on S in $\mathcal{O}(\log r_S)$ time.*

By combining the result of Lemma 2 with the algorithm described in Section 2.5, we obtain:

Theorem 17. *There exists a data structure representing $BWT(S)$ in $\mathcal{O}(r)$ words of space and supporting text `left-extension` (i.e. update the structure to represent $BWT(cS)$) in $\mathcal{O}(\log r)$ time.*

Theorem 17 directly implies that we can build $BWT(\overleftarrow{S})$ *online* in $\mathcal{O}(r)$ words of working space and $\mathcal{O}(n \log r)$ time. Applying the reasoning made at the beginning of this chapter, we obtain:

Theorem 18. *We can build $BWT(S)$ with an algorithm running in $\mathcal{O}(n \log r)$ time and using $\mathcal{O}(r)$ words of working space.*

Implementation

The algorithm described in this section has been implemented in our `DYNAMIC` library, where it takes the name `rle-bwt`. See Section 6.4.2 for a description and theoretical analysis of the implementation. See Chapter 7 for an experimental evaluation of `rle-bwt`.

4

Computing LZ77 in Compressed Working Space

The Lempel-Ziv factorization LZ77 (Definition 2) is an important tool in text compression, being its size z closely related with the number of repetitions in the processed string. Moreover, by augmenting it with additional (proportional-size) structures, one can obtain fast and high-order compressed full-text indexes [66, 67]. Structures based on LZ77 have been shown to be competitive in terms of space on repetitive text collections with respect to BWT-based self indexes [16,67], and a careful combination of the two techniques stands at the basis of some of the most time-and-space efficient repetition-aware indexes [5].

4.1 Related Work

The Lempel-Ziv factorization can be computed in linear time and $\mathcal{O}(n \log n)$ bits of working space by using suffix trees or suffix arrays [22, 23, 58]. Recent results—building up on the FM index [33] data structure—reduced space to *compact* ($\mathcal{O}(n \log \sigma)$ bits), while retaining linear running time [82]. The best space bound to date is achieved by the algorithm discussed in Kreft's thesis [65], which builds the LZ77 factorization of the text in $\mathcal{O}(n \log^{1+\epsilon} n)$ time ($\epsilon > 0$) and $n(H_k + 2) + o(n \log \sigma)$ bits of space (although the $\mathcal{O}(n)$ term prevents space from being *fully* compressed).

A line of this research is focused on the *online* computation of the LZ factorization. Okanohara et al. [93] showed that this task can be carried out in $\mathcal{O}(n \log^3 n)$ time using only $(1+o(1))n \log \sigma + \mathcal{O}(n)$ bits of working space. Starikovskaya [114] reduced the running time to $\mathcal{O}(n \log^2 n)$, while slightly increasing the working space to $\mathcal{O}(n \log \sigma)$ bits. Finally, Yamamoto et al. [122] obtained $\mathcal{O}(n \log n)$ running time within $\mathcal{O}(n \log \sigma)$ bits of working space by using Directed Acyclic Word Graphs (DAWGs). Our first result (Section 4.2) is an online algorithm computing LZ77 in $\mathcal{O}(n \log n)$ time and zero-order compressed space, therefore improving upon all above solutions.

After presenting this result, we focus on run-length compression. While fixed-order statistical methods are not able to capture long repetitions [41], techniques such as LZ77, grammar compression [13], and run-length encoding of the Burrows-Wheeler transform [77, 111] have been shown superior in the task of compressing highly repetitive texts. In these domains, algorithms working in space $\Theta(n \log n)$ [22], $\mathcal{O}(n \log \sigma)$ [7, 91], or even $\mathcal{O}(nH_k)$ [67] bits are of little use as they could be much more memory-demanding than the final compressed representation. Very recent results suggested that it is possible to achieve these goals in repetition-aware working space. Fischer et al. [39] proposed a randomized

algorithm to compute in $\mathcal{O}(\epsilon^{-1} n \log n)$ time and $\mathcal{O}(z)$ words of space an approximation of the parsing consisting of at most $(1+\epsilon)z$ phrases, where $0 < \epsilon \leq 1$. Nishimoto et al. [89] show how to build the LZ77 parsing in $\mathcal{O}(z \log n \log^* n)$ words of space. In Section 4.3 we present two algorithms computing LZ77 in $\mathcal{O}(n \log r)$ time and space proportional (in words) to r (i.e. the number of runs in the Burrows-Wheeler transform of the text). All algorithms presented in this chapter have been implemented in our library `DYNAMIC`. See Chapter 7 for an experimental evaluation of our solutions.

To sum up, in this chapter we present algorithms computing LZ77 within the following time/space bounds:

- $\mathcal{O}(n \log n)$ time and $nH_0 + o(n \log \sigma) + \mathcal{O}(\sigma \log n)$ bits of working space (Section 4.2)
- $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r)$ words of working space (Section 4.3)

The basic structure we use is a (entropy/run-length compressed) dynamic FM index over the reversed text, updated by inserting T-characters from the first to the last.

4.2 Zero-Order Compressed Working Space

In this section we prove the following theorem:

Theorem 19. *The LZ77 factorization of a text $T \in \Sigma^n$ can be built online in $nH_0 + o(n \log \sigma) + \mathcal{O}(\sigma \log n)$ bits of working space and $\mathcal{O}(n \log n)$ time*

The content of this Section is based on paper `(ii)`. The core of our algorithm is a zero-order compressed dynamic FM index. We start by presenting all employed data structures, and then proceed by combining them in our final solution.

In Figure 4.1 we contextualize Theorem 19 in our framework of algorithmic tools (Figure 1.1).

4.2.1 Data Structures

Our theoretical result builds upon a recent insight by Navarro and Nekrich on the optimal representation of dynamic strings [87]: there exists a data structure that permits to represent a sequence $S[0, n-1]$ over an alphabet $\Sigma = \{0, ..., \sigma-1\}$ in $nH_0 + o(n \log \sigma) + \mathcal{O}(\sigma \log n)$ bits of space and that supports queries (access, rank, select) and updates (insertions and deletions) in $\mathcal{O}(\log n / \log \log n)$ time. The bound is worst-case for the queries and amortized for the updates.

We use the optimal sequence representation of Navarro and Nekrich [87] to build a dynamic FM index taking $nH_0 + o(n \log \sigma) + \mathcal{O}(\sigma \log n)$ bits of space that supports (amortized) $\mathcal{O}(\log n / \log \log n)$-time left-extension of the text with an arbitrary character and LF function computation, and $\mathcal{O}(\log^2 n / \log \sigma)$-time locate. Note that this is not the solution we adopt in practice in our implementation (where we use instead wavelet trees built upon dynamic bitvectors), see Section 6.4.3 for full details. Our algorithm scans the text from its first to last character, building the dynamic FM index of the reversed text. At each step (i.e. text character), we (1) update the BWT interval of the current LZ phrase and (2) insert a new text character in the index. Each time the BWT interval becomes empty, we have reached the end of the current LZ phrase and we use a locate query to compute the LZ-factor.

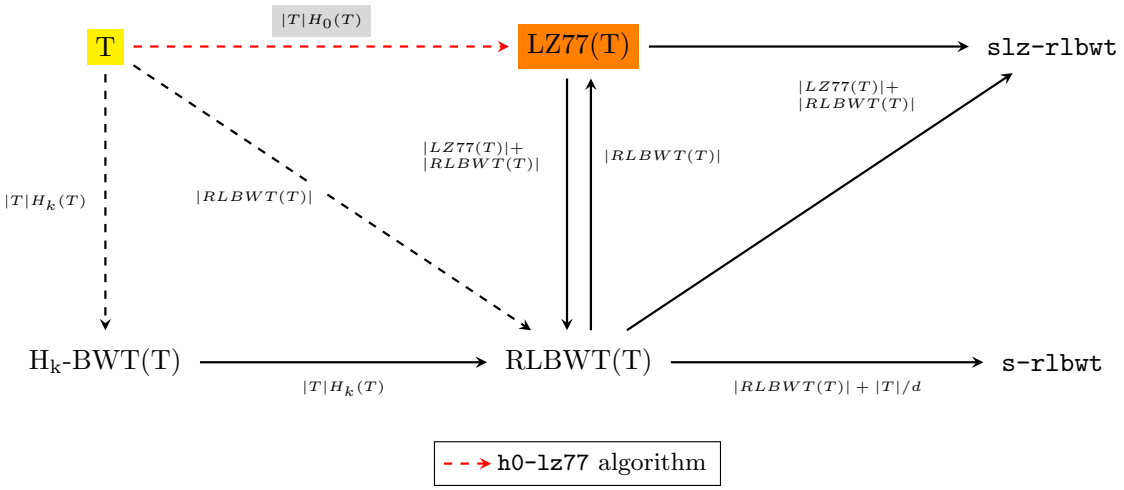


Figure 4.1: Our `h0-lz77` algorithm builds online the Lempel-Ziv parsing using zero-order compressed working space

Dynamic FM Index The principal component of our dynamic FM index is a dynamic BWT, updated with the algorithm described in Section 2.5. Dynamic FM indexes were firstly described by Chan et al. [12] and by Mäkinen and Navarro [76]. Our dynamic FM index differs from these proposals in that we also compress the bitvector marking sampled BWT positions, therefore achieving overall compressed working space (whereas the `MARK` structure of [12] requires $\mathcal{O}(n)$ bits of space). For completeness, in this section we describe all details of this dynamic index. We use two *different* terminator symbols—$\# \in \Sigma$ and $\$ \notin \Sigma$—to mark the end of the forward (LZ77 algorithm) and reverse (BWT algorithm) text, respectively. Our algorithm will therefore work on texts of the form $\$W\#$, $W \in \Sigma^*$.

In our algorithm, we index the sequence $S = \overleftarrow{T}\$$. By using the dynamic sequence representation of Navarro and Nekrich [87], we can build $BWT(\overleftarrow{T}\$)$ online in overall $\mathcal{O}(n \log n / \log\log n)$ time and $nH_0 + o(n \log \sigma) + \mathcal{O}(\sigma \log n)$ bits of space by inserting characters in the order $\$, T[0], ..., T[n-1]$ with the procedure of Section 2.5. In the following paragraphs, we will denote with BWT the Burrows-Wheeler transform of the current suffix of $S = \overleftarrow{T}\$$.

The second ingredient we need in order to compute the LZ77 factorization of T is a dynamic suffix array sampling to support fast locate. The main challenge is to add such functionality without asymptotically increasing space usage. Let $\gamma > 0$ be the sample rate, and $m = \lceil n/\gamma \rceil$ be the number of stored suffix array pointers. To this end, we employ two structures:

1. A compressed dynamic bitvector B to mark with a "1" sampled F-positions.
2. A dynamic sequence representation $SA[0, m-1]$ over the alphabet $[0, n-1]$ taking compact space ($\mathcal{O}(m \log n)$ bits) and supporting $\mathcal{O}(\log n)$-time access and insert operations.

We use a sample rate of $\gamma = \log_\sigma n \log\log n$. For component (1), we use again the

dynamic sequence representation of Navarro and Nekrich. We remind the reader that the size of a zero-order compressed bitvector B' with b bits set is $nH_0(B') \leq b\log(n/b)+b\log e$. Since B has $m = \lceil n/\gamma \rceil = \lceil \frac{n}{\log_\sigma n \log\log n} \rceil$ bits set, it follows easily that B takes overall $nH_0(B) + o(n) + \mathcal{O}(\log n) = o(n)$ bits of space.

For component (2), we use a simple balanced tree (e.g. a red-black tree or a B-tree with constant fanout) where we store suffix array samples in the leaves and we augment each internal node with the size of the corresponding subtree. Access and insert in position i are then implemented by descending the tree according to the subtree-size counters, accessing/inserting the suffix array pointer in the leaves, and (in the case of insert) updating $\mathcal{O}(\log m)$ subtree-size counters. The tree takes overall $\mathcal{O}(m \log n) = o(n \log \sigma)$ bits of space, and access/insert operations take $\mathcal{O}(\log m) = \mathcal{O}(\log n)$ time. Structures B and SA take overall $o(n \log \sigma)$ bits of space.

Implementing *extend* With $BWT.extend(c) \in \{0, ..., |BWT|\}$, $c \in \Sigma \cup \{\$\}$, we will denote the function that:

1. updates the BWT of the current S suffix by left-extending it with a new character c
2. updates the suffix array samples, and
3. returns the L-position of character \$ *after* the left-extension has taken place.

To avoid updating the already inserted suffix array pointers at each text extension, in structure SA we enumerate S-positions starting from the last. In this sense, $S[n] = \$$ corresponds to SA-position 0, and $S[0]$ corresponds to SA-position n (remember that $|S| = |\overleftarrow{T}\$| = n + 1$). Suppose we have built the structures for the length-$(i-1)$ suffix of S and that we want to left-extend it with the new character $S[n-i+1]$. Let j be such that $BWT[j] = \$$, $r = BWT.rank(S[n-i+1], j)$, and $k = BWT.F(S[n-i+1]) + r$. Operation $BWT.extend(S[n-i+1])$ is implemented as follows:

1. We update BWT with the new text character $S[n-i+1]$ as described in Section 2.5
2. If $i \mod \gamma = 0$, then we insert a new suffix array pointer in SA and mark with a "1" the corresponding F-position in B: $SA.insert(i-1, B.rank(1,k))$ and $B.insert(1,k)$
3. Otherwise ($i \mod \gamma \neq 0$), we mark with a "0" the new suffix F-position in B: $B.insert(0,k)$

Step (1) takes $\mathcal{O}(\log n/\log\log n)$ amortized time. The insertion of a bit in B takes $\mathcal{O}(\log n/\log\log n)$ time, and the insertion of a suffix array pointer in SA takes $\mathcal{O}(\log n)$ time. Since we update SA every $\log_\sigma n \log\log n$ left-extensions, *extend* takes overall $\mathcal{O}(\log n/\log\log n)$ amortized time.

Implementing *locate* Let BWT be the Burrows-Wheeler transform of the current S suffix. Operation $BWT.locate(i)$ returns the S-position (enumerated from right to left) corresponding to the F-position i. We implement this operation as usual, i.e. by backward-navigating the current S suffix until a sampled F-position or the first suffix position is found:

1. If i is such that $BWT[i] =' \$'$, then we return $|BWT| - 1$.
2. Otherwise:
   (a) If $B[i] = 1$, then we return $SA[B.rank(1, i)]$.
   (b) If $B[i] = 0$, then we return $BWT.locate(i') - 1$, where $i' = BWT.F(c) + BWT.rank(c, i)$ and $c = BWT[i]$.

Since we use a sample rate of $\log_\sigma n \log\log n$ and access and rank operations on BWT take $\mathcal{O}(\log n/\log\log n)$ time, after $\mathcal{O}(\log^2 n/\log\sigma)$ time we find a marked F-position. Then, extracting the suffix array pointer from structure SA takes $\mathcal{O}(\log n)$ time. Since we assume $\sigma \leq n$, *locate* takes overall $\mathcal{O}(\log^2 n/\log\sigma)$ time.

Implementing LF Function The LF function requires a constant number of rank and access operations on BWT, so it takes overall $\mathcal{O}(\log n/\log\log n)$ time.

4.2.2 The Algorithm

The extension step of our algorithm is described in Algorithm 13. The algorithm takes as input one T character c, and outputs either the LZ factor ended by c or nothing if c does not end a factor. In Algorithm 13, variables BWT (the dynamic BWT described in section 4.2.1), $[l, r]$ (BWT interval of the current phrase), len (length of the current phrase), and i (L-position of character \$) are global, and are initialized at the beginning as $BWT \leftarrow' \$'$, $[l, r] \leftarrow [0, 0]$, $len \leftarrow 0$, and $i \leftarrow 0$.

First of all, in line 1 we perform one backward-search step using function LF. The new BWT interval $[l', r']$ is nonempty if and only if the current phrase Wc, $W \in \Sigma^*$, does appear previously in the text. If this is the case (lines 16-19), then we increment the current phrase length (line 17), left-extend the current S suffix (line 18), and update the BWT interval of $c\overleftarrow{W}$ (line 19) by incrementing its right bound r'. This step is always needed since in line 18 the new S suffix (prefixed by $c\overleftarrow{W}$) falls inside interval $[l', r']$.

Otherwise, if Wc does not occur previously and $len = |W| > 0$ (lines 2-8), then Wc is a new LZ factor and interval $[l, r]$ holds all occurrences of $\overleftarrow{W}$ seen until now in the *reversed* text. Notice, however, that $[l, r]$ holds also the *current* occurrence of $\overleftarrow{W}$ (i.e. $i \in [l, r)$) in addition to at least one previous occurrence (i.e. $r - l \geq 2$). We must therefore be careful to output a *previous* occurrence of $\overleftarrow{W}$: in lines 4-8 we locate either l or $r - 1$, depending on which one is different from i. Moreover, we must subtract len from the located text position since *locate* returns an occurrence of $\overleftarrow{W}$ in the *reversed* text, and position 0 is reserved for the terminator character \$. After locating the occurrence, we can extend the BWT with character c (line 12), reset the BWT interval to the full range $[0, |BWT| - 1]$ (line 13), reset phrase length to zero (line 14), and return the factor.

The last case to consider is when Wc does not occur previously and $len = |W| = 0$ (lines 9 and 10). Then, this is the first occurrence of c in the text and we simply output a factor $\langle null, 0, c\rangle$ after extending the BWT with character c and resetting the global variables as described above (lines 13-14).

From the analysis carried out in section 4.2.1 it is clear that, excluding *locate*, all steps in Algorithm 13 take (amortized) $\mathcal{O}(\log n/\log\log n)$ time. Notice that we call *locate* once per phrase. It is known that the number z of LZ77 phrases satisfies $z \in \mathcal{O}(n/\log_\sigma n)$ [69].

Algorithm 13: add_character(c)

input : Character $c \in \Sigma$ (right-extending current T prefix)
output: A factor $\langle pos, len, c \rangle$ if c ends a factor. Nothing otherwise.

```
1  [l', r'] ← BWT.LF([l, r], c);                          /* backward search step */
2  if l' ≥ r' then
3      if len > 0 then
4          if i = l then
5              occ ← r − 1;
6          else
7              occ ← l;
8          P ← BWT.locate(occ) − len;          /* locate a previous occurrence */
9      else
10         P ← null;                                  /* first occurrence of c */
11     L ← len;                          /* length of current phrase (c excluded) */
12     BWT.extend(c);                          /* insert character c in the BWT */
13     [l, r] ← [0, |BWT| − 1];                                  /* reset interval */
14     len ← 0;                                   /* reset phrase length */
15     return ⟨P, L, c⟩;                                  /* return LZ factor */
16 else
17     len ← len + 1;                        /* increase current phrase length */
18     i ← BWT.extend(c);                      /* insert character c in the BWT */
19     [l, r] ← [l', r' + 1];                  /* new suffix falls inside [l', r'] */
```

Since the cost of a single *locate* query is $\mathcal{O}(\log^2 n / \log \sigma)$, in Algorithm 13 *locate* takes $\mathcal{O}(\log n)$ amortized time. As a result, by calling Algorithm 13 on $T[0], ..., T[n-1]$, Theorem 19 follows: we can build the LZ77 factorization of T online in $nH_0 + o(n \log \sigma) + \mathcal{O}(\sigma \log n)$ bits of working space and $\mathcal{O}(n \log n)$ time.

Notice that, if we wish to compute only the LZ phrase boundaries, then we do not need *locate*, and the LZ factorization can be built using a simplified version of Algorithm 13 in $\mathcal{O}(n \log n / \log \log n)$ time.

Implementation

The algorithm described in this section has been implemented in our library `DYNAMIC`, where it takes the name `h0-lz77`. See Section 6.4.3 for a description and theoretical analysis of the implementation. See Chapter 7 for an experimental evaluation of `h0-lz77`.

4.3 Run-Length Compressed Working Space

While entropy compression works well for text with unbalanced character frequencies, it is not able to exploit *long repetitions*. Run-length compression of the Burrows-Wheeler

transform is better suited for this task. In this section we show two algorithms building LZ77 in only $\mathcal{O}(r)$ words of space. The content of this section is based on papers `(iii)` and `(iv)`. Note that papers `(iii)` and `(iv)` describe only offline solutions.

The main obstacle in building LZ77 within $\mathcal{O}(r)$ words of space with a run-length encoded FM-index is the suffix array (SA) sampling: by sampling the SA every $0 < k \leq n$ text positions, this component takes $\mathcal{O}(n/k)$ words of space and supports *locate* queries in time proportional to k. The main contributions of this section are two algorithms that compute LZ77 by combining a (dynamic) run-length BWT with a repetition-aware sparse suffix array sampling. The first algorithm stores only two samples per BWT equal-letter run, while the second stores at most one sample per LZ77 factor. Both algorithms run in $\mathcal{O}(n \log r)$ time and require $\mathcal{O}(r)$ words of working space. We point out that—unfortunately—the suffix array samplings we employ in our algorithms do not represent a general sampling mechanism supporting the retrieval of all occurrences of a string, as they only support the retrieval of *at least one* pattern occurrence. This is however sufficient for locating LZ77 factors, so this strategy is suitable to solve the LZ77 factorization problem in small space.

As a by-product of our results we obtain a $\mathcal{O}(r)$-space algorithm to convert from RLBWT- to LZ77-based compression formats (see Section 5.2.2 for full details). This is one of the first works showing how to *convert* a compression format into another without first decompressing the text; see [2, 3, 106] (grammar compression to/from Lempel-Ziv), and [118] (run-length encoding of the text to LZ78) for similar results. Another important application of our results is related to text indexing. In particular, we obtain that indexes based on combinations of LZ77 and RLBWT compressors—see Section 5.3.3—can be built in asymptotically optimal $\mathcal{O}(z + r)$ words of working space. To the best of our knowledge, the only other repetition-aware index that can be built in asymptotically optimal working space is based on grammar compression and is described in [116].

4.3.1 First Algorithm: SA Sampling Based on BWT Runs

In this section we describe our first algorithm. The main data structures we use are a dynamic RLBWT of the text $\overleftarrow{T}$ (i.e. $RLBWT^+(\overleftarrow{T})$) and σ sets storing the suffix array sampling. The algorithm works in two phases.

In the first phase, we read T from left to right, building $RLBWT^+(\overleftarrow{T})$. This step employs the online BWT construction algorithm described in Section 2.5, which requires a dynamic string data structure D to represent the BWT. The algorithm performs a total amount of $|T|$ `rank` and `insert` operations on D. In our case, D will be designed to be also run-length compressed: we represent it with the data structure described in Section 3.3.

In the second phase (Algorithm 15), we scan T left to right once more, this time using the RLBWT just built—i.e. by repeatedly using the LF mapping on $RLBWT^+(\overleftarrow{T})$ starting from $T[0]$— and output the LZ77 factors.

While reading $T[j]$ for $j > 0$ in the second phase, we must determine whether $T[i, \ldots, j]$, with i first position of the current LZ-phrase, occurs in $T[0, \ldots, j-1]$. If this is not the case, then we output the LZ triple $\langle \pi, j - i, T[j] \rangle$, where π corresponds to the source of the current LZ-phrase (and, hence, $T[\pi, \ldots, \pi + j - i - 1] = T[i, \ldots, j-1]$ and $\pi = \perp$ in case $i = j$). Note that the computation is performed on an index of the entire text (not just of $T[0, \ldots, j]$), thus we need to take special care to ensure that the occurrences of

$T[i, \ldots, j]$ we find are indeed *previous* occurrences. Informally, we need an index of the entire text for the following reason. Our strategy will consist in maintaining this invariant: we keep track, for each BWT run, of the two most external suffix array samples (i.e. text positions) encountered while scanning the text left-to-right. Using an index for $T[0, \ldots, j]$ only, we do not know whether an equal-letter run a^k will later be split in two runs $a^{k'}ca^{k''}$ (with $k' + k'' = k$ and $a \neq c$). In such a case, we would have to sample the last and first a's of the two new runs $a^{k'}$ and $a^{k''}$, respectively, in order to preserve the validity of our invariant. Sampling (i.e. mapping an L-position on the text) is an expensive task as it requires navigating the BWT until a sample is found ($\mathcal{O}(n)$ backwards steps), so this strategy is not feasible. Notice that keeping an index for the entire text solves this problem as we already have access to all runs and therefore we know which L-positions, among the ones we have already visited, are the most external in their run.

In the following we show how to implement our algorithm in $\mathcal{O}(r)$ words of working space, by maintaining σ dynamic sets equipped with a total of $\mathcal{O}(r)$ SA-samples.

Dynamic Suffix Array Sampling

From now on BWT stands for $BWT(\overleftarrow{T})$. Note that, even though we say that we *sample the suffix array*, we actually sample text positions associated with BWT positions, i.e. we sample T-positions on the L-column instead of T-positions on the F-column of the BWT matrix. Moreover, since we enumerate positions in T-order (not $\overleftarrow{T}$-order), k-th BWT-position will correspond to sample $(n - SA[k]) \bmod n$, where $SA[k]$ is the k-th entry in the (standard) suffix array of $\overleftarrow{T}$.

Let j be a T-position and k its corresponding BWT-position: $T[j] = BWT[k]$. We store SA-samples as pairs $\langle j, k\rangle$ and each pair is of one of three types: *singleton*, denoted as $\langle j, k\rangle^\circ$, *open*, denoted as ${}^{\lceil}\langle j, k\rangle$, and *close*, denoted as $\langle j, k\rangle^{\rfloor}$. If the pair type is not relevant for the discussion, we simply write $\langle j, k\rangle$.

Let $\Sigma = \{s_1, \ldots, s_\sigma\}$ be the alphabet. Samples are stored in σ red-black trees $\mathcal{B}_{s_1}, \ldots, \mathcal{B}_{s_\sigma}$ and are ordered by BWT coordinate (i.e. the second component of the pairs). While reading $a = T[j] = BWT[k]$ we first locate the (inclusive) bounds $l \leq k \leq r$ of its associated BWT a-run, then we update the trees according to the following rules:

(A) If for all $\langle j', k'\rangle \in \mathcal{B}_a$, $k' \notin [l, r]$, then we insert the singleton $\langle j, k\rangle^\circ$ in $\mathcal{B}_a$.

(B) If there exists $\langle j', k'\rangle^\circ \in \mathcal{B}_a$ such that $k' \in [l, r]$, then we remove it and:

  (a) If $k < k'$, then we insert in $\mathcal{B}_a$ the pairs ${}^{\lceil}\langle j, k\rangle$ and $\langle j', k'\rangle^{\rfloor}$,

  (b) If $k' < k$, then we insert in $\mathcal{B}_a$ the pairs ${}^{\lceil}\langle j', k'\rangle$ and $\langle j, k\rangle^{\rfloor}$.

(C) If there exist ${}^{\lceil}\langle j', k'\rangle, \langle j'', k''\rangle^{\rfloor} \in \mathcal{B}_a$ such that $k', k'' \in [l, r]$:

  (a) If $k < k' < k''$, then we remove ${}^{\lceil}\langle j', k'\rangle$ from $\mathcal{B}_a$ and insert ${}^{\lceil}\langle j, k\rangle$ in $\mathcal{B}_a$,

  (b) If $k' < k'' < k$, then we remove $\langle j'', k''\rangle^{\rfloor}$ from $\mathcal{B}_a$ and insert $\langle j, k\rangle^{\rfloor}$ in $\mathcal{B}_a$,

  (c) Otherwise ($k' < k < k''$), we leave the trees unchanged.

We say that a BWT a-run $BWT[l, \ldots, r]$ *contains a pair* or, equivalently, *contains a SA-sample*, if there exists some $\langle j, k\rangle \in \mathcal{B}_a$ such that $k \in [l, r]$. It is easy to see that the

following invariants hold for the above three rules: (i) each BWT run contains either no pairs, a singleton pair, or two pairs—one open and one close; (ii) If a BWT run contains an open $^{\lceil}\langle j', k'\rangle$ and a close $\langle j'', k''\rangle^{\rceil}$ pair, then $k' < k''$; (iii) once we add a SA-sample inside a BWT run, that run will always contain at least one SA-sample.

We say that BWT-position k is *marked by SA-sample* $\langle j, k\rangle$, when $a = T[j] = BWT[k]$ and $\langle j, k\rangle \in \mathcal{B}_a$.

Let $BWT[k_\$] = \$$. By saying that T-positions $0, \ldots, j$ have been *processed*, we mean that—starting with all trees empty—we have applied the update rules to the SA-samples $\langle 0, k_\$\rangle, \langle 1, BWT.LF(k_\$)\rangle, \langle 2, BWT.LF^2(k_\$)\rangle, \ldots, \langle j, BWT.LF^j(k_\$)\rangle$, where $BWT.LF^i(k_\$)$ denotes i applications of the LF map starting from BWT-position $k_\$$. We now prove that, after processing $0, \ldots, j$, we can locate at least one occurrence of any string that occurs in $T[0, \ldots, j]$. This property will allow us to locate LZ phrase boundaries and previous occurrences of LZ phrases.

Lemma 3. *If* $0, \ldots, j$ *have been processed and* $[l, r]$ *is the BWT interval associated with* $\overleftarrow{V} \in \Sigma^m$*, with* V right-maximal *in* T*, then*

$$\exists \langle j', k'\rangle \in \mathcal{B}_a \text{ such that } k' \in [l, r] \text{ if and only if } Va \text{ occurs in } T[0, \ldots, j].$$

Proof. ($\Rightarrow$) If $\langle j', k'\rangle \in \mathcal{B}_a$ with $k' \in [l, r]$ exists, then clearly $T[j' - m, \ldots, j'] = Va$. Moreover, since we processed T-positions $0, \ldots, j$ only, it must be the case that $j' \leq j$ and hence Va occurs in $T[0, \ldots, j]$.

($\Leftarrow$) Let $T[t, \ldots, t+m] = Va$, with $t \leq j - m$. Consider the BWT a-run corresponding to $T[t + m] = a$. One of the following cases can hold true:

(1) The BWT a-run is entirely included in $BWT[l, \ldots, r]$ *and* is neither a prefix nor a suffix of $BWT[l, \ldots, r]$, that is $BWT[l, \ldots, r] = Xca^edY$, for some $X, Y \in \Sigma^*,\ c, d \neq a, e > 0$. Then, it follows from invariant (iii) and rule (A) that since we have visited T-position $t + m$, the a-run must contain at least one SA-sample. This is the pair $\langle j', k'\rangle$ we are looking for.

(2) The BWT a-run spans either position l or position r. Since V is right-maximal in T, then $BWT[l, \ldots, r]$ contains also a character $b \neq a$. We therefore have that either (i) $BWT[l, \ldots, r] = a^eXbY$, or (ii) $BWT[l, \ldots, r] = YbXa^e$, where $X, Y \in \Sigma^*,\ e > 0$. The two cases are symmetric hence we discuss only (i).

Consider all T-prefixes $T[0, \ldots, j'']$ such that $j'' \leq j$, Va is a suffix of $T[0, \ldots, j'']$, and the lexicographic rank of $\overleftarrow{T[0, \ldots, j'' - 1]}$ among all $\overleftarrow{T}$-suffixes is $k'' \in [l, l + e - 1]$ (i.e. the suffix lies in $BWT[l, \ldots, l + e - 1] = a^e$). There exists at least one such T-prefix: $T[0, \ldots, t + m]$. Then, the rank k' of the lexicographically largest $\overleftarrow{T}$-suffix with the above properties is such that $\langle j', k'\rangle \in \mathcal{B}_a$ for some $j' \leq j$. This is implied by the three update rules described above. The BWT position k corresponding to T-position $t + m$ lies in the BWT interval $[l, l + e - 1]$, therefore either (i) k is the rightmost position visited in its run (and it is marked with a SA-sample), or (ii) the rightmost visited position $k' > k$ in $[l, l + e - 1]$ is marked with a SA-sample (note that *lexicographically largest* translates to *rightmost* on BWT intervals). □

We can drop the right-maximality requirement from Lemma 3.

Corollary 1. *Once processed* T*-positions* $0, \ldots, j - 1$ *(none if* $j = 0$*), after processing also* $j, \ldots, j + m - 1$*,* $m > 0$*, if a string* $W \in \Sigma^m$ *occurs in* $T[0, \ldots, j + m - 1]$*, then we can locate one of its occurrences.*

Proof. We prove the property by induction on $|W| = m > 0$. Let $W = Va$, $V \in \Sigma^{m-1}$, $a \in \Sigma$. If $m = 1$, then $V = \epsilon$ (empty string). Since T contains at least two distinct characters (a and \$), V is right-maximal. Therefore we can apply Lemma 3 to find an occurrence of $W = a$.

If $m > 1$, then $|V| > 0$ and two cases can occur. If V is right-maximal, then we can again apply Lemma 3 to find an occurrence of $W = Va$ in $T[0, \ldots, j + m - 1]$ (remember that Va occurs in $T[0, \ldots, j + m - 1]$). If, instead, V is not right-maximal, then it is always followed by a in T. By inductive hypothesis we can locate an occurrence π of V in $T[0, \ldots, j + m - 2]$. But then, since all occurrences of V in T are followed by a, π is also an occurrence of $W = Va$ in $T[0, \ldots, j + m - 1]$. □

Corollary 1 gives us a recursive algorithm to locate previous occurrences of phrases. Figures 4.2, 4.3, and 4.4 depict the three cases of the strategy (see next section for a more detailed description). In Figure 4.2 the phrase prefix is right-maximal but the letter that follows is not sampled on the BWT range (we output a LZ factor); in Figure 4.3 the phrase prefix is right-maximal and the letter that follows is sampled on the BWT range (we extend the current LZ factor); in Figure 4.4 the phrase prefix is not right-maximal (we extend the current LZ factor).

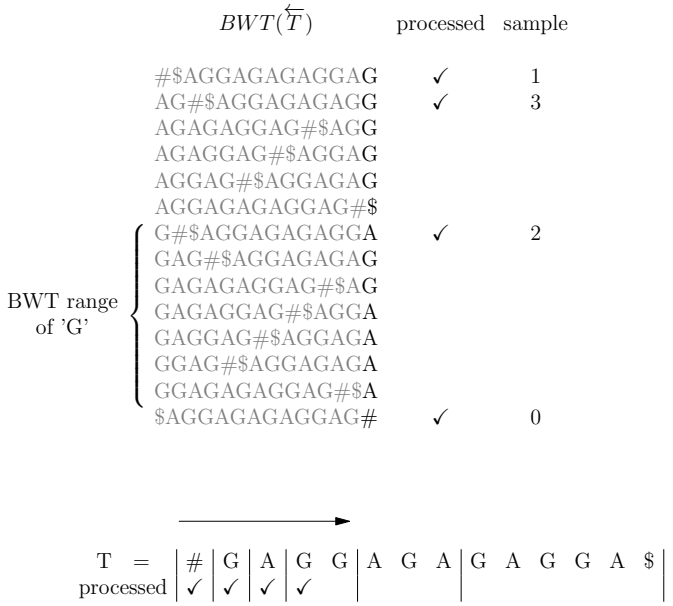


Figure 4.2: Case 1: we are trying to extend the phrase prefix 'G' with a 'G'. The range of 'G' spans more than 1 run ('G' is right-maximal) and there are no sampled 'G' in the range. It follows that 'GG' does not appear before in the text. Note that in this and in the following pictures, the text is already LZ77-factored (vertical bars) for clarity.

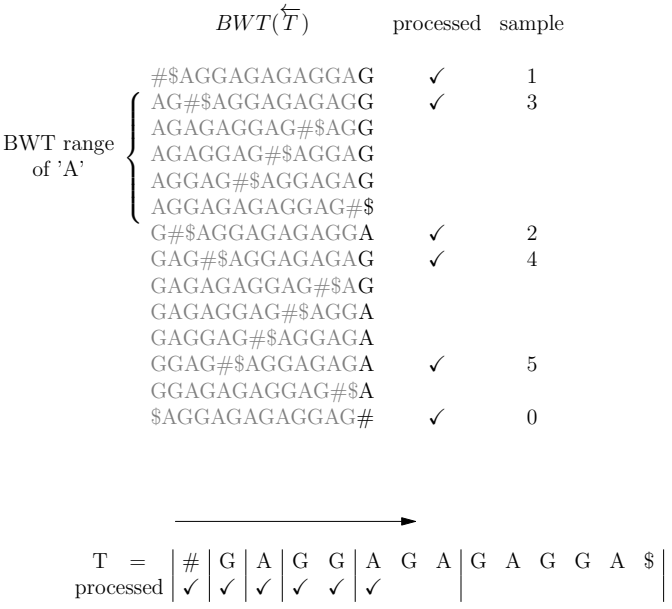


Figure 4.3: Case 2: we are trying to extend the phrase prefix 'A' with a 'G'. The range of 'A' spans more than 1 run ('A' is right-maximal) and there is a sampled 'G' in the range. It follows that 'AG' appears before in the text (at position $sample - length = 3 - 1 = 2$).

Pseudocode

Our complete procedure is reported as Algorithm 14. The algorithm just builds (line 1) $RLBWT^+(\overleftarrow{T})$ using the online algorithm of Section 3.3, and then calls the sub-procedure described as Algorithm 15 to convert $RLBWT^+(\overleftarrow{T})$ to $LZ77(T)$. This is the only step requiring access to the input text, which is read only once from left to right. Since the dynamic string we use is run-length compressed, this step requires $\mathcal{O}(r)$ words of working space.

From this point we describe Algorithm 15. For brevity, RLBWT indicates the data structure $RLBWT^+(\overleftarrow{T})$. From lines 1 to 8 we initialize all variables. In order: the text length n, the current position j in T, the position k in $RLBWT$ corresponding to position j in T (at the beginning, $T[0] = RLBWT[k_\$] = \$$), the current LZ77 phrase prefix length λ (last character $T[j]$ excluded), the T-position $\pi < j$ at which the current phrase prefix $T[j-\lambda, \ldots, j-1]$ occurs ($\pi = \perp$ if $\lambda = 0$), the red-black trees $\mathcal{B}_{s_1}, \ldots, \mathcal{B}_{s_\sigma}$ used to store SA-samples, the current character $c = T[j] = RLBWT[k]$, and the interval $[l, r]$ corresponding to the current reversed LZ phrase prefix $\overleftarrow{T[j-\lambda, \ldots, j-1]}$ in $RLBWT$ (when $\lambda = 0$, $[l, r]$ is the full interval $[0, n-1]$).

The *while* loop at line 9 scans T positions from the first to last. First of all, we have to discover if the current character $T[j] = c$ ends a LZ phrase. In line 10 we count the number u of runs that intersect interval $[l, r]$ on $RLBWT$. If $u = 1$, then the current phrase prefix $T[j-\lambda, \ldots, j-1]$ is always followed by c in T (i.e. it is not right-maximal), and consequently $T[j]$ cannot be the last character of the current LZ phrase. Otherwise,

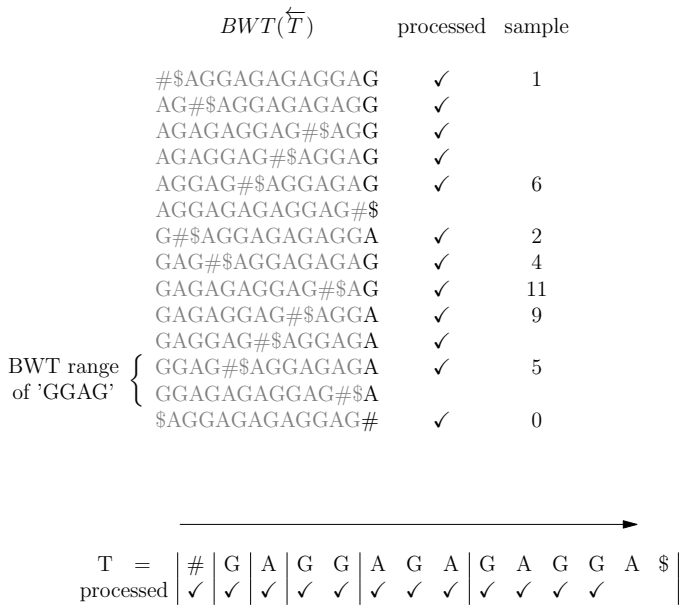


Figure 4.4: Case 3: we are trying to extend the phrase prefix 'GAGG' with an 'A'. The range of 'GGAG' contains only one run ('GAGG' is not right-maximal). Then, all 'GAGG' are followed by 'A' in the text. Since we previously located 'GAGG' (inductive hypothesis) at position 1, it follows that also 'GAGGA' appears at position 1. Note that not all processed positions are marked with a SA sample (only the most external ones in each run).

by Lemma 3 $T[j-\lambda,\ldots,j]$ occurs in $T[0,\ldots,j-1]$ if and only if there exists a SA-sample $\langle j',k'\rangle \in \mathcal{B}_c$ such that $l \leq k' \leq r$. The existence of such pair can be verified with a binary search on the red-black tree $\mathcal{B}_c$. In line 11 we perform these two tests. If at least one of these two conditions holds, then $T[j-\lambda,\ldots,j]$ occurs in $T[0,\ldots,j-1]$ and therefore it is not a LZ phrase. If this is the case, we now have to find $\pi < j-\lambda$ such that $T[\pi,\ldots,\pi+\lambda] = T[j-\lambda,\ldots,j]$ (i.e. a previous occurrence of the current LZ phrase prefix). The implementation of this task follows the inductive proof of Corollary 1. If $u = 1$ (current phrase prefix is not right-maximal) then π is already the value we need. Otherwise (Lines 12-13) we find a SA-sample $\langle j',k'\rangle \in \mathcal{B}_c$ such that $k' \in [l,r]$ (such pair must exist since $u > 1$ and the condition in Line 11 succeeded). Procedure $\mathcal{B}_c.locate(l,r)$ returns such j' (to make the procedure deterministic, one could return the value j' associated with the smallest BWT position $k' \in [l,r]$). Then, we assign to π the value $j'-\lambda$ (Line 13). We can now increment the current LZ phrase prefix length (Line 14) and update the BWT interval $[l,r]$ so that it corresponds to the string $\overleftarrow{T[j-\lambda+1,\ldots,j]}$ (LF mapping in Line 15).

If both the conditions at line 11 fail, then the string $T[j-\lambda,\ldots,j]$ does not occur in $T[0,\ldots,j-1]$ and therefore is a LZ phrase. By the inductive hypothesis of Corollary 1, $\pi < j-\lambda$ is either $\perp$—if $\lambda = 0$—or such that $T[\pi,\ldots,\pi+\lambda-1] = T[j-\lambda,\ldots,j-1]$ otherwise. At line 17 we can therefore output the LZ factor. We now have to open (and start searching in RLBWT) a new LZ phrase: at lines 18-20 we reset the current phrase

prefix length, set π to $\perp$, and reset the interval associated to the current (reversed) phrase prefix to the full interval.

All we are left to do now is to process position j (i.e. apply the update rules to the SA-sample $\langle j, k \rangle$) and proceed to the next text position. At line 21 we locate the (inclusive) borders $[l_{run}, r_{run}]$ of the BWT run containing position k (i.e. $k \in [l_{run}, r_{run}]$). This information is used at line 22 to apply the update rules on $\mathcal{B}_c$ and on the SA-sample $\langle j, k \rangle$. Finally, we increment the current T-position j (line 23), compute the corresponding position k on RLBWT (line 24), and read the next T-character c on the RLBWT.

Algorithm 14: rle_lz77_1(T)

input : A text $T \in \Sigma^n$ beginning with \$ and ending with #
output: LZ77 factors of T in text order.

```
1 RLBWT ← build_rev_RLBWT(T);          /* Build RLBWT+(←T) */
2 rlbwt_to_lz77(RLBWT);                /* convert RLBWT+(←T) to LZ77 */
```

Analysis

`rank`, `access`, and `insert` operations on RLBWT take $\mathcal{O}(\log r)$ time each. Operations $\mathcal{B}_c.exists\_sample(l, r)$ (line 11) and $\mathcal{B}_c.locate(l, r)$ (Line 13) require a binary search on the red-black tree and can also be implemented in $\mathcal{O}(\log r)$ time. $RLBWT.number\_of\_runs(l, r)$ is the number of bits set in $V_{all}[l, \ldots, r]$, plus 1 if $V_{all}[l] = 0$: this operation requires therefore $\mathcal{O}(1)$ `rank`/`access` operations on V_{all} ($\mathcal{O}(\log r)$ time). Similarly, $RLBWT.locate\_run(k)$ requires finding the two bits set preceding and following position k in V_{all} ($\mathcal{O}(\log r)$ time with a constant number of `rank` and `select` operations). We obtain:

Theorem 20. *Algorithm 14 computes the LZ77 factorization of a text $T \in \Sigma^n$ in $\mathcal{O}(r)$ words of working space and $\mathcal{O}(n \log r)$ time, r being the number of runs in the Burrows-Wheeler transform of $\overleftarrow{T}$.*

Implementation

The algorithm described in this section has been implemented in our library `DYNAMIC`, where it takes the name `rle-lz77-1`. See Section 6.4.4 for a description and theoretical analysis of the implementation. See Chapter 7 for an experimental evaluation of `rle-lz77-1`.

4.3.2 Second Algorithm: SA Sampling Based on LZ77 Factors

As we will empirically demonstrate in Chapter 7, storing two suffix array samples per BWT run could be expensive. In practice, z is often smaller than r (see, e.g. [77]) so a suffix array sampling based on LZ77 factors is more desiderable. In this section we show an algorithm achieving this goal.

Here we give an overview of the algorithm, which is described more in detail in the next subsection. The algorithm works in three steps. In the first step, we build online $RLBWT^+(\overleftarrow{T})$ by reading T-characters from left to right and by inserting them in a run-length compressed dynamic string data structure (with the same algorithm used in the

Algorithm 15: rlbwt_to_lz77($RLBWT$)

input : A run-length Burrows-Wheeler transform data structure $RLBWT$ over the reverset text $\overleftarrow{T}$ (i.e. $RLBWT^{+}(\overleftarrow{T})$)
output: LZ77 factors of T in text order.

```
1  n ← |RLBWT|;                          /* T length (equal to RLBWT's length) */
2  j ← 0 ;                    /* Last position (on T) of current LZ phrase prefix */
3  k ← k_$ ;                                  /* Position of $ in RLBWT */
4  λ ← 0;                                /* Length of current LZ phrase prefix */
5  π ← ⊥;                      /* Previous occurrence of current LZ phrase prefix */
6  B_{s_1}, ..., B_{s_σ} ← ∅;          /* Initialize red-black trees of SA-samples */
7  c ← RLBWT[k];                                /* Current T character */
8  [l, r] ← [0, n − 1];          /* Range of current LZ phrase prefix in RLBWT */
9  while j < n do
10     u ← RLBWT.number_of_runs(l, r);          /* Runs intersecting [l, r] */
11     if u = 1 or B_c.exists_sample(l, r) then
12         if u > 1 then
13             π ← B_c.locate(l, r) − λ;          /* Occurrence of phrase prefix */
14         λ ← λ + 1;                  /* Increase length of current LZ phrase */
15         [l, r] ← RLBWT.LF([l, r], c);                /* Backward search step */
16     else
17         Output ⟨π, λ, c⟩;                              /* Output LZ77 factor */
18         λ ← 0;                                /* Reset phrase prefix length */
19         π ← ⊥;                             /* Reset phrase prefix occurrence */
20         [l, r] ← [0, n − 1];          /* Reset range of current LZ phrase prefix */
21     [l_run, r_run] ← RLBWT.locate_run(k) ;          /* run of BWT position k */
22     B_c.update_tree(⟨j, k⟩, [l_run, r_run]);                /* Apply update rules */
23     j ← j + 1;                                    /* Increment T position */
24     k ← RLBWT.LF(k);              /* RLBWT position corresponding to j */
25     c ← RLBWT[k];                                /* Read next T character */
```

previous section). At the same time, we search in the RLBWT the current (reversed) LZ77 phrase with the strategy described in Section 4.2. While doing this, we mark BWT positions corresponding to sources of (reversed) LZ phrases with the corresponding phrase rank: while searching the j-th LZ phrase, as soon as the BWT interval for $\overleftarrow{Wc}$, $W \in \Sigma^{\lambda}$, $c \in \Sigma, \lambda > 0$ becomes empty, we mark one of the F-positions in the BWT interval for $\overleftarrow{W}$ with the integer j, being careful of choosing a position corresponding to a *previous* occurrence of W in the text (not the current one). Note that a F-position can be assigned more than one integer, so we need to maintain *sets* of integers on a *subset* of F-positions. This problem can be solved efficiently with a dynamic sparse bitvector marking with a '1' F-positions with at least one integer, with a dynamic succinct bitvector storing sets

multiplicities in unary (i.e. a size-k set, $k > 0$, corresponds to the sequence of bits 10^{k-1} in this bitvector), and with a dynamic sequence of integers (for this last component we can use the SPSI described in Section 3.3.1).

In the second step, we scan T from left-to-right by using the $RLBWT$ just built (i.e. by applying iteratively the LF function starting from F-position 0) and we use the integers stored in the previous step to locate the sources of LZ phrases. We store such sources in a vector $SOURCES[0, \ldots, z-1]$ initialized with $\perp$ values: while reading text position i, if the position is associated with a set $\{j_1, ..., j_t\}$ of integers, we assign the value i to $SOURCES[j_1], \ldots, SOURCES[j_t]$. Note that i is the *last* position of the source, so in the next algorithm step we will need to subtract λ from it before outputting the LZ77 factor.

In the third and last step we delete all structures except $SOURCES$ and re-build $RLBWT$ by reading T left-to-right. As done in step 1, while building $RLBWT$ we search the current (reversed) LZ phrase. In this way, each time the BWT interval for $\overleftarrow{Wc}$, $W \in \Sigma^{\lambda}$, $c \in \Sigma, \lambda \geq 0$ becomes empty, we output the LZ77 factor $\langle (SOURCES[j] - \lambda) + 1, \lambda, c \rangle$ (or $\langle \perp, 0, c \rangle$ if $\lambda = 0$), $j = 0, \ldots, z-1$ being the rank of the current LZ phrase. Note that $SOURCES[j]$ is $\perp$ if and only if the j-th phrase is a single character. Note moreover that our second and third steps could be integrated in one single step by storing in `SOURCES` also phrase lengths during the second step. Dividing the strategy in three steps allows saving some space in practice (even though it does not make any difference under big-$\mathcal{O}$ notation).

Pseudocode

Algorithm 16 describes steps 1 (lines 1-22) and 2 (lines 23-29) sketched above. In lines 1-5 we initialize the number z of LZ phrases (0 at the beginning: we will count them online), the RLBWT (as an empty run-length encoded string data structure), the BWT interval $[l, r]$ of the current LZ phrase prefix, the current LZ phrase prefix length λ, and the position k of the BWT terminator character \$ on the L column of the BWT. At the beginning, $k = \perp$ (undefined) as the BWT is empty.

We are going to read $T[i]$ for $i = 0, \ldots, n-1$ and build $RLBWT^{+}(\overleftarrow{T})$ while computing phrase boundaries. At lines 7-8 we perform a backward search step to extend the BWT interval $[l, r]$ with the current text character. Two cases can occur.

If the interval $[l', r']$ corresponding to $\overleftarrow{Wc}$, $W \in \Sigma^{\lambda}$, $c = T[i] \in \Sigma, \lambda \geq 0$ is empty (line 9), then Wc is a LZ77 phrase and—in the case $\lambda > 0$—we need to assign the current phrase rank to one of the sources of $\overleftarrow{W}$ on the RLBWT. If $\lambda = 0$, then the phrase is a single character and it has no source. Otherwise (lines 11-14), we need to pick a position inside $[l, r]$ different than the *current* occurrence of W. Since the last character we inserted in the RLBWT is $W[|W| - 1]$, the current occurrence of W appears at the k-th BWT row, k being the position of \$ in the L-column. At lines 12 and 14 we therefore insert the integer z, z being the current LZ phrase rank, in the set associated with either F-position l or r, depending on which one is different than k. In pseudocode 16 we denote the integer set associated with F-position k with $RLBWT.set\_at(k)$. Note that it must be the case that $r > l$ since W occurs at least twice in $T[0, \ldots, i-1]$. We finally update $RLBWT^{+}(\overleftarrow{T[0, \ldots, i-1]})$ to $RLBWT^{+}(\overleftarrow{T[0, \ldots, i]})$ with an extension step (line 15) and, at lines 16-18 we reset the BWT interval to the full interval, reset the phrase length λ to

0, and increase the number z of LZ phrases seen until now.

In the second case, the interval $[l', r']$ corresponding to $\overleftarrow{Wc}$, $W \in \Sigma^{\lambda},\ c = T[i] \in \Sigma, \lambda \geq 0$ is not empty (line 19). Then, we simply increase the length of the current phrase prefix (line 14) and extend the RLBWT with $T[i]$ (line 21). The extension step at line 21 returns the new position k of the \$ character on the L column of the BWT. Note that the (reverse of the) current occurrence of Wc falls inside $[l', r']$, so we need to update this interval by extending its right boundary by 1 (line 22).

We can now scan the RLBWT and assign a source to each phrase. At line 23 we initialize the $SOURCES$ vector with $\perp$ values. From here, k represents the F-position on the BWT corresponding to text position i (starting from $i = 0$). Since we start from $T[0] =$ \$ and \$ appears at position 0 on the F-column, at line 24 we initialize k to 0. For each $i = 0, \ldots, n-1$, we then check if F-position k is associated with a nonempty set $RLBWT.set\_at(k)$ of integers. If this is the case, for each such integer $j \in RLBWT.set\_at(k)$ at line 27 we assign the source i to the j-th LZ phrase. Synchronization between indexes i and k is guaranteed by the execution of the LF step at line 28.

The complete procedure to compute the parse is reported as Algorithm 17. We do not discuss it in detail as it basically repeats the online construction of the RLBWT described above while computing LZ phrase boundaries. While doing this, at line 10 we access the $SOURCES$ vector computed with procedure $find\_sources(T)$ and output LZ77 phrases in text order, being careful to subtract the phrase length from the content of $SOURCES$ since this vector contains the *last* position of each phrase source. To simplify the description, at this line we use the convention that $(SOURCES[j] - \lambda) + 1 = \perp$ if $SOURCES[j] = \perp$.

Analysis

Building the RLBWT in the first and third steps and performing the n backward search steps takes overall $\mathcal{O}(n \log r)$ time (see Section 3.3). We update the sets of integers once per phrase; we remind that such sets are encoded with a dynamic gap-encoded bitvector, a dynamic succinct bitvector, and a dynamic string. The total number of integers is z, so each update operation on the sets takes $\mathcal{O}(\log z)$ time with the structures described in Section 3.3 and the red-black tree implementing the dynamic string. Since $z \in \mathcal{O}(n/\log_\sigma n)$, updating and querying the sets takes therefore $\mathcal{O}(z \log z) \subseteq \mathcal{O}(n \log \sigma) \subseteq \mathcal{O}(n \log r)$ time.

As discussed in Section 3.3, the RLBWT takes $\mathcal{O}(r)$ words of space. Each integer stored in the sets takes $\mathcal{O}(1)$ words (including RBT pointers), so the algorithm uses overall $\mathcal{O}(r+z)$ words of working space. We can reduce the working space to $\mathcal{O}(r)$ words by means of the following theorem:

Lemma 4. *The number z of LZ77 phrases of a text T can be computed with an online algorithm running in $\mathcal{O}(n \log r)$ time and using $\mathcal{O}(r)$ words of working space, r being the number of equal-letter runs in $BWT(\overleftarrow{T})$.*

Proof. Algorithm 17 without the instructions at lines 5 and 10 solves exactly this problem: we just need to return the value z at the end of its execution. □

We can use Lemma 4 and compute z in $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r)$ words of working space before computing the actual parse. If $z \leq 2r$, we execute Algorithm 17, otherwise

Algorithm 16: find_sources(T)

input : A text $T \in \Sigma^n$ beginning with \$ and ending with #
output: A vector $SOURCES[0, \ldots, z-1]$, z being the size of the LZ77 parse, such that $SOURCES[j]$ is the last position of j-th phrase's source.

```
1  z ← 0;                                          /* Initialize size of the parse */
2  RLBWT ← ϵ;                                      /* Initialize RLBWT to empty string */
3  [l, r] ← [0, 0];                                /* Initialize range on RLBWT */
4  λ ← 0;                                          /* Length of current LZ phrase */
5  k ← ⊥;        /* Position of $ in RLBWT (here ⊥ because RLBWT = ϵ) */
6  for i = 0 ... |T| − 1 do
7      c ← T[i];                                   /* read current text character */
8      [l′, r′] ← RLBWT.LF([l, r], c);             /* backward search step */
9      if l′ > r′ then
10         if λ > 0 then
11             if k = l then
12                 RLBWT.set_at(r).insert(z);
13             else
14                 RLBWT.set_at(l).insert(z);
15         RLBWT.extend(c);                        /* insert character c in the BWT */
16         [l, r] ← [0, i];                        /* reset [l, r] to full interval */
17         λ ← 0;                                  /* reset phrase length */
18         z ← z + 1;                              /* increase number of phrases */
19     else
20         λ ← λ + 1;                              /* increase current phrase length */
21         k ← RLBWT.extend(c);  /* extend with c.  Return position of $ */
22         [l, r] ← [l′, r′ + 1]; /* new suffix falls inside [l′, r′]:  increment r′ */
23 SOURCES[0, ..., z − 1] ← ⟨⊥, ..., ⊥⟩;           /* initialize SOURCES */
24 k ← 0;                                          /* position of $ on F column */
25 for i = 0 ... |T| − 1 do
26     for each j ∈ RLBWT.set_at(k) do
27         SOURCES[j] ← i;                         /* assign source to the j-th phrase */
28     k ← RLBWT.LF(k);                            /* LF step:  navigate T forward */
29 return SOURCES;
```

Algorithm 14. Overall, this combined strategy runs therefore in $\mathcal{O}(n \log r)$ time and uses $\mathcal{O}(r)$ words of working space. We obtain:

Theorem 21. *The algorithm above described computes the LZ77 factorization of a text $T \in \Sigma^n$ in $\mathcal{O}(r)$ words of working space and $\mathcal{O}(n \log r)$ time, r being the number of runs*

Algorithm 17: rle_lz77_2(T)

input : A text $T \in \Sigma^n$ beginning with \$ and ending with #
output: LZ77 factors of T in text order.

```
1  z ← 0;                                          /* Initialize size of the parse */
2  RLBWT ← ϵ;                                      /* Initialize RLBWT to empty string */
3  [l, r] ← [0, 0];                                /* Initialize range on RLBWT */
4  λ ← 0;                                          /* Length of current LZ phrase */
5  SOURCES ← find_sources(T);                      /* locate phrase sources */
6  for i = 0 ... |T| − 1 do
7      c ← T[i];                                   /* read current text character */
8      [l′, r′] ← RLBWT.LF([l, r], c);             /* backward search step */
9      if l′ > r′ then
10         Output ⟨(SOURCES[z] − λ) + 1, λ, c⟩;    /* Output LZ77 factor */
11         RLBWT.extend(c);                        /* insert character c in the BWT */
12         [l, r] ← [0, i];                        /* reset interval */
13         λ ← 0;                                  /* reset phrase length */
14         z ← z + 1;                              /* increase number of phrases */
15     else
16         λ ← λ + 1;                              /* increase current phrase length */
17         RLBWT.extend(c);                        /* extend with c */
18         [l, r] ← [l′, r′ + 1]; /* new suffix falls inside [l′, r′]: increment r′ */
```

in the Burrows-Wheeler transform of $\overleftarrow{T}$.

Implementation

The algorithm described in this section has been implemented in our `DYNAMIC` library, where it takes the name `rle-lz77-2`. See Section 6.4.4 for a description and theoretical analysis of the implementation. See Chapter 7 for an experimental evaluation of `rle-lz77-2`.

5

Compressed Computation: Recompression and Indexing

Chapters 3 and 4 dealt with the problem of efficiently compressing text, i.e. turning an uncompressed format into a compressed one. In this chapter, we fully enter the field of compressed computation: the algorithms we describe take as input compressed representations of the text and tackle the problem of manipulating such data and answering queries on it using asymptotically the same working space as the sizes of the input and the output. We focus on two important compression techniques for highly repetitive text collections, LZ77 and RLBWT, tackling two fundamental problems in the field:

(1) can we restructure compressed data (i.e. *re-compress* it, see Section 5.2)?
(2) can we turn a compressed file into a compressed index?

Section 5.2 deals with question (1). We show algorithms to convert between RLBWT and LZ77 formats using a working space proportional to the sizes of the input and the output. Part of these results are unpublished (in particular, the conversion $LZ77 \rightarrow RLBWT$). Sections 5.3.2 and 5.3.3 describe two compressed indexes for repetitive text collections; the content of these sections is based on papers `(v)` and `(vi)`. By combining the results of Sections 3.3, 4.3, and 5.2 we show that such indexes can be either built by reading the text *once* character-by-character (e.g. from a streamed source) or from a compressed-file representation (either based on LZ77 or RLBWT), thus answering affirmatively to question (2). The results of Section 5.2 imply, moreover, the possibility of converting—within compressed working space—between RLBWT- and LZ77-based indexes such as the ones described in [65, 66, 67, 111] and Sections 5.3.2 and 5.3.3.

5.1 Repetitivity Measures: The r-z-g^* Relations

The most successful indexes for repetitive text collections are based on run-length encoding of the Burrows-Wheeler transform, on the Lempel-Ziv (LZ77) factorization, and on straight-line programs, i.e. grammars that generate only one string (the text). Before introducing compressed indexes for repetitive collections based on these techniques, it is interesting to briefly discuss the relationships among these three measures of repetitiveness: the number r of BWT runs, the number z of LZ77 phrases, and the size g^* of the smallest SLP (i.e. number of rules of the smallest grammar generating the text). The relationship between Lempel Ziv parsing and Grammar compression has been settled by Rytter with the following theorem:

Theorem 22. *Lempel Ziv and Grammar compression [106]. The sizes z of the LZ77 factorization and g^* of the smallest grammar satisfy $z \leq g^*$ and $g^* \in \mathcal{O}(z \log(n/z))$.*

Moreover, the bound $g^* \in \mathcal{O}(z \log(n/z))$ is almost tight:

Theorem 23. *[50]. The sizes z of the LZ77 factorization and g^* of the smallest grammar satisfy $g^* \in \Omega(z \log n / \log \log n)$*

These inequalities imply that grammar compression is inherently less powerful than Lempel-Ziv parsing, especially considering the fact that the smallest SLP cannot be approximated in polynomial time within a constant factor unless P=NP [13], and that the LZ77 factorization can be computed in linear time [58].

The exact relationships among r, z, and g^*, however, remain an open problem. Both z and r are at least σ and can be $\Theta(\sigma)$, e.g. in the text $(0\ 1\ \ldots\ \sigma-1)^e$, $e > 0$. Being[1] $g^* \in \Omega(\log n)$, this example shows—on constant-sized alphabets—that the rates g^*/r and g^*/z can be $\Theta(\log n)$.

The following theorem shows the existence of texts that are asymptotically more compressible with LZ77 than with RLBWT:

Theorem 24. *There exists an infinite collection of strings for which $r/z \in \Theta(\log n)$ holds*

Proof. Such family of strings is that of de Bruijn sequences of order $k > 1$, i.e. circular strings of length σ^k having as substrings all the strings in Σ^k. Consider the BWT row-partition induced by length-$(k-1)$ substrings of a de Bruijn sequence in the first $k-1$ columns of the matrix. Each $x \in \Sigma^{k-1}$ appears exactly σ times in the de Bruijn sequence and all such occurrences are preceded by different characters. It follows that each of the above BWT classes contains at least $\sigma - 1$ runs, so the BWT has at least $(\sigma-1)\sigma^{k-1} \in \Theta(\sigma^k) = \Theta(n)$ runs. The number of LZ77 phrases of any text is, on the other hand, always $\mathcal{O}(n/\log_\sigma n)$. The rate r/z is maximized considering a constant-sized alphabet, on which we get $r/z \in \Theta(\log n)$. □

The above theorem may suggest that LZ77 is inherently more powerful than RLBWT. However, as shown in the following theorem, this is not the case:

Theorem 25. *There exists an infinite collection of strings for which $z/r \in \Theta(\log n)$ holds*

Proof. Fibonacci words satisfy this property. Such words are defined recursively as follows: $f_1 = a$, $f_2 = b$, $f_n = f_{n-1}f_{n-2}$. Fibonacci words are a particular case of *standard words*; such words produce a total clustering of the alphabet letters in the BWT [79] (i.e. two runs). On the other hand, the LZ77 factorization of f_n corresponds to the factorization of f_n into *singular words* $\hat{f}_i$, where each $\hat{f}_i$ is obtained by complementing the first letter in the left rotation of the Fibonacci word f_i (see [38] for more details). Since $|f_i|$ is exponential in i, it follows that the Lempel-Ziv factorization of f_n has $\Theta(\log |f_n|)$ factors. □

Finally, Navarro [85] reports an example where—for big enough σ—r can grow by $\mathcal{O}(\sqrt{n})$ for each new edit in the text. Since z only grows by $\mathcal{O}(1)$ after a text edit, this example suggests that there could possibly be texts for which $r = \Theta(z\sqrt{n})$ holds. Considering that $g^* \in \mathcal{O}(z \log(n/z))$, this moreover suggests that there exist cases where grammar compression is more powerful than run-length encoding of the Burrows-Wheeler transform.

[1] since SLP nonterminal rules are of the form $X \to ZW$, there must be at least a logarithmic number of rules.

5.2 Recompression

In this section we address a central point in compressed computation: can we restructure compressed text (i.e. change compression format) without explicit decompression? Being able to perform such task opens the possibility of (i) converting between compressed data structures (e.g. self-indexes) based on different compressors and (ii) using algorithms designed to work on a specific compression format taking as input a different compressed input representation. In our case, recompression will serve as a powerful tool to build in small space a compressed index based on multiple compressors taking as input any of these compressed representations of the file.

5.2.1 Related Work

The term *recompression* was introduced almost simultaneously for the first time by Jeż [53], Goto et al. [46], and Tamakoshi et al. [118]. Recompression indicates the process of restructuring compressed data (e.g. changing compression format) without explicit decompression. In the first case [53], the author uses recompression as a powerful technique to solve *word equations*, i.e. equations involving equalities between strings. The same technique was later used by the same author [53, 54, 55, 56] to solve several problems related to straight-line programs, including pattern matching and approximation of the smallest grammar. These interesting papers concern only grammar compression; the general idea behind this technique is to restructure two straight-line programs in a normalized form such that the resulting compressed representations are equal if and only if the original (uncompressed) strings are equal. In [46, 118], the term is used more broadly to indicate the conversion between different compression formats while using limited resources. In particular, the string should not be fully decompressed during recompression (otherwise the problem becomes trivial). Rytter [106] shows how to convert the LZ77 encoding of a text into a grammar-based encoding, while Bannai et al. [2, 3] consider the opposite direction (though pointing to LZ78 instead of LZ77). In [118] the authors consider conversions between LZ78 and run-length encoding of the text. Note that LZ77 and run-length encoding of the BWT are much more powerful than LZ78 and run-length encoding of the text, respectively, so methods addressing conversion between LZ77 and RLBWT would be of much higher interest.

In this section we show how to efficiently solve this problem in space proportional to the sizes of these two compressed representations. See Definitions 1 and 2 for a formal definition of $RLBWT(T)$ and $LZ77(T)$ as a list of r pairs and z triples, respectively. Let $RLBWT(T) \to LZ77(T)$ denote the computation of the list $LZ77(T)$ using as input the list $RLBWT(T)$ (analogously for the opposite direction). In this section, we describe the following results:

(1) We can compute $RLBWT(T) \to LZ77(T)$ in $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r)$ words of working space
(2) We can compute $LZ77(T) \to RLBWT(T)$ in $\mathcal{O}\big(n(\log r + \log z)\big)$ time and $\mathcal{O}(r + z)$ words of working space

Result (1) is based on the algorithms described in Section 4.3 and requires space proportional to the input *only* (output is streamed to disk). Result (2) requires space proportional to the input *plus* the output, since data structures based on both compressors

are used into main memory. In order to achieve result (2), we show how we can (locally) decompress $LZ77(T)$ while incrementally building a run-length BWT data structure of $\overleftarrow{T}$. Extracting text from LZ77 is a computationally expensive task, as it requires a time proportional to the parse height h (see Definition 11) per extracted character [67], with h as large as n in the worst case. The key ingredient of our solution is to use the run-length BWT data structure itself to efficiently extract text from $LZ77(T)$.

We note that our algorithms perform a number of steps proportional to the size n of the text. Considering that the compressed file could be *exponentially* smaller than the text, a future improvement over our results would be to perform the same tasks in a time proportional to $r + z$.

We recall, from Section 1.4, that $RLBWT^+(T)$ indicates a run-length BWT *data structure* supporting the computation of `LF` and `FL` functions and `extend` queries (as opposed to $RLBWT(T)$, which indicates a list of pairs).

5.2.2 From RLBWT to LZ77

The algorithm we provide to compute $RLBWT(T) \rightarrow LZ77(T)$ is based on the result presented in Section 4.3 (Algorithm 15). We recall that Algorithm 15 computes $LZ77(T)$ using $RLBWT^+(\overleftarrow{T})$. Our strategy to compute $LZ77(T)$ using $RLBWT(T)$ is the following:

1. We convert $RLBWT(T)$ to $RLBWT^+(T)$ (i.e. we add support for `RSA` queries on $RLBWT(T)$)
2. We compute $RLBWT^+(\overleftarrow{T})$ using $RLBWT^+(T)$
3. We run Algorithm 15 and compute $LZ77(T)$ using $RLBWT^+(\overleftarrow{T})$

Let $RLBWT(T) = \langle \lambda_i, c_i \rangle_{i=1,\ldots,r}$ (see Definition 1). Step 1 can be performed by just inserting characters $c_1^{\lambda_1} c_2^{\lambda_2} \ldots c_r^{\lambda_r}$ (in this order) in a dynamic run-length encoded string data structure.

Step 2 is performed by extracting characters $T[0], T[1], \ldots, T[n-1]$ from $RLBWT^+(T)$ and inserting them (in this order) in a dynamic $RLBWT$ data structure with the BWT construction algorithm described in Section 2.5. Since this algorithm builds the $RLBWT$ of the *reversed* text, the final result is $RLBWT^+(\overleftarrow{T})$. Algorithm 18 shows the pseudocode of this procedure. Note that we call the `FL` function on RLBWT, i.e. function mapping F- to L- BWT positions. `FL` requires an `access` and a `rank` on the F column and a `select` on the L column of the BWT. In Line 3, note that we compute the RLBWT position corresponding to $T[0]$ as $RLBWT.FL(RLBWT.FL(0))$. Since \$ appears at F-position 0, $RLBWT.FL(0)$ returns the L-position containing \$, i.e. the F-position containing $T[0]$. With another application of `FL` we obtain the L-position containing $T[0]$. Finally, note that in Line 4 we perform just $|RLBWT| - 1$ steps since we already inserted \$ in $RLBWT^{rev}$.

We can state our first result:

Theorem 26. *Conversion $RLBWT(T) \rightarrow LZ77(T)$ can be performed in $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r)$ words of working space.*

Proof. We use the dynamic RLBWT structure of Section 3.3.3 to implement $RLBWT^+(T)$ and $RLBWT^+(\overleftarrow{T})$. Step 1 requires n `insert` operations in $RLBWT^+(T)$, and terminates

Algorithm 18: invert_rlbwt($RLBWT$)

input : $RLBWT = RLBWT^{+}(T)$
output: $RLBWT^{+}(\overleftarrow{T})$

```
1 RLBWT^rev ←' $'; /* Initialize empty RLBWT of reverse text */
2 j ← 0; /* current text position */
3 i ← RLBWT.FL(RLBWT.FL(0)); /* RLBWT position corresponding to j */
4 while j < |RLBWT| − 1 do
5 |   RLBWT^rev.extend(RLBWT[i]); /* left-extend reversed text */
6 |   j ← j + 1; /* Advance position on text */
7 |   i ← RLBWT.FL(i); /* RLBWT position crresponding to j */
8 return RLBWT^rev;
```

therefore in $\mathcal{O}(n \log r)$ time. Since the string we are building contains r runs, this step uses $\mathcal{O}(r)$ words of working space. Step 2 (Algorithm 18) calls n `extend` and `FL` queries on dynamic RLBWTs. From Section 2.5, `extend` requires a constant number of `rank` and `insert` operations. The FL function requires just an `access` and a `rank` on the F column and a `select` on the L column. It follows that both operations are supported in $\mathcal{O}(\log r)$ time, so also step 2 terminates in $\mathcal{O}(n \log r)$ time. Recall (Section 1.4) that r is defined to be the maximum between the number of runs in $BWT(T)$ and $BWT(\overleftarrow{T})$. Since in this step we are building $RLBWT^{+}(\overleftarrow{T})$ using $RLBWT^{+}(T)$, the overall space is bounded by $\mathcal{O}(r)$ words. Finally, Algorithm 15 terminates in $\mathcal{O}(n \log r)$ time while using $\mathcal{O}(r)$ words of space (Theorem 20). The claimed bounds for our Algorithm to compute $RLBWT(T) \rightarrow LZ77(T)$ follow. □

In Figure 5.1 we contextualize Theorem 26 in our framework of algorithmic tools (Figure 1.1).

5.2.3 From LZ77 to RLBWT

Our strategy to convert $LZ77(T)$ to $RLBWT(T)$ comprises the following steps:

1. We extract $T[0], T[1], \ldots, T[n-1]$ from $LZ77(T)$ and build online $RLBWT^{+}(\overleftarrow{T})$
2. We convert $RLBWT^{+}(\overleftarrow{T})$ to $RLBWT^{+}(T)$
3. We extract equal-letter runs from $RLBWT^{+}(T)$ and stream $RLBWT(T)$ to the output

Step 2 can be performed with Algorithm 18. Step 3 requires reading $RLBWT^{+}(T)[0]$, ..., $RLBWT^{+}(T)[n-1]$ and keeping in memory a character storing last run's head and a counter keeping track of last run's length. Whenever we open a new run, we stream last run's head and length to the output.

The problematic step is number 1. As mentioned in the introduction, extracting a character from $LZ77(T)$ requires to follow a chain of character copies. In the worst case, the length h of this chain—also called the parse height (see Definition 11)—can be as

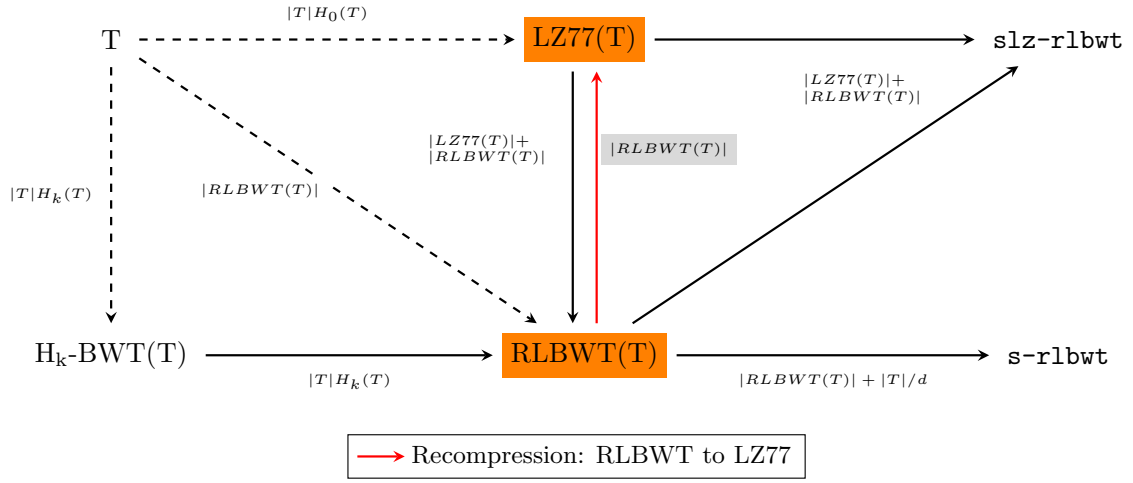


Figure 5.1: Our algorithm to convert $RLBWT(T)$ to $LZ77(T)$ uses a working space proportional to the size of a run-length encoded BWT.

large as n. Our observation is that, since we are building $RLBWT^+(\overleftarrow{T})$, we can use this component to efficiently extract text from $LZ77(T)$: while decoding factor $\langle \pi_v, \lambda_v, c_v \rangle$, we convert π_v to a position on the RLBWT and extract λ_v characters from it. The main challenge in achieving this goal is to convert text positions to RLBWT positions (taking into account that the RLBWT is dynamic and therefore changes in size and content). Considering that $RLBWT^+(\overleftarrow{T})$ is built incrementally, we need a data structure to encode a *dynamic* function $\mathcal{Z} : \{0, ..., n-1\} \to \{0, ..., n-1\}$ mapping text positions to $RLBWT$ positions and supporting the following three operations (see below the list for a description of how these operations will be used):

- `map`: $\mathcal{Z}(i)$. Compute the image of i
- `expand`: $\mathcal{Z}.expand(j)$. Set $\mathcal{Z}(i)$ to $\mathcal{Z}(i) + 1$ for every i such that $\mathcal{Z}(i) \geq j$
- `assign`: $\mathcal{Z}(i) \leftarrow j$. Call $\mathcal{Z}.expand(j)$ and set $\mathcal{Z}(i)$ to j

We say that $\mathcal{Z}(i)$ is *defined* if we executed $\mathcal{Z}(i) \leftarrow j$ at some previous point of the computation, for some j. For simplicity, we restrict our attention to the case where—when calling $\mathcal{Z}(i) \leftarrow j$—argument i is greater than all i' such that $\mathcal{Z}(i')$ is defined. This case will be sufficient to solve our problem. We moreover assume that $\mathcal{Z}(i)$ is always defined when calling $\mathcal{Z}(i)$ (this will be the case in our algorithm).

Function $\mathcal{Z}.expand(j)$ will be used when we insert $T[i]$ at position j in the partial $RLBWT^+(\overleftarrow{T})$ and j is not associated with any phrase source (i.e. $i \neq \pi_v$ for all $v = 1, \ldots, z$). $\mathcal{Z}(i) \leftarrow j$ will instead be used when we insert $T[i]$ at position j in the partial $RLBWT^+(\overleftarrow{T})$ and $i = \pi_v$ for some $v = 1, \ldots, z$ (possibly more than one). In the next section we show how to reduce this task to the elegant problem of finding a data structure for a *dynamic permutation*.

Dynamic functions

We provide a structure $\mathcal{Z}(i)$ satisfying:

Lemma 5. *Letting z be the size of the LZ77 parsing of T, our data structure $\mathcal{Z}(i)$ takes $\mathcal{O}(z)$ words of space and supports* `map`*,* `expand`*, and* `assign` *operations on $\mathcal{Z} : \{0, ..., n-1\} \to \{0, ..., n-1\}$ in $\mathcal{O}(\log z)$ time*

Proof. First of all note that, since $LZ77(T)$ is our input, we know beforehand the domain $\mathcal{D} = \{\pi \mid \langle \pi, \lambda, c\rangle \in LZ77(T) \ \wedge \pi \neq \perp\}$ of $\mathcal{Z}$. We can therefore restrict our attention to functions $\mathcal{Z}' : \{0, ..., d-1\} \to \{0, ..., n-1\}$, $d = |\mathcal{D}| \leq z$. Then, to compute $\mathcal{Z}(i)$ we map $0 \leq i < n$ to a value $0 \leq i' < d$ by binary-searching a precomputed array containing the sorted values of $\mathcal{D}$ and return $\mathcal{Z}'(i')$. Similarly, $\mathcal{Z}(i) \leftarrow j$ is implemented by executing $\mathcal{Z}'(i') \leftarrow j$ (with i' defined as above), and $\mathcal{Z}.expand(j)$ simply as $\mathcal{Z}'.expand(j)$.

We use a dynamic gap-encoded bitvector C (see Section 3.3.2) marking with a bit set positions j such that $j = \mathcal{Z}(i)$ for some i. When initializing $\mathcal{Z}$, C is empty. Let k be the number of bits set in C at some step of the computation. We can furthermore restrict our attention to *surjective* functions $\mathcal{Z}'' : \{0, ..., d-1\} \to \{0, ..., k-1\}$ as follows. $\mathcal{Z}'(i')$ (`map`) returns $C.select_1(\mathcal{Z}''(i'))$. The `assign` operation $\mathcal{Z}'(i') \leftarrow j$ requires an `insert` $C.insert(1, j)$ followed by the execution of $\mathcal{Z}''(i') \leftarrow C.rank_1(j)$. Operation $\mathcal{Z}'.expand(j)$ is implemented with $C.insert(0, j)$.

To conclude, since we restrict our attention to the case where—when calling $\mathcal{Z}(i) \leftarrow j$—argument i is greater than all i' such that $\mathcal{Z}(i')$ is defined, we will execute `assign` operations $\mathcal{Z}''(i') \leftarrow j''$ for increasing values of $i' = 0, 1, \ldots, d-1$, where $i' = k$ is the current domain size. We therefore focus on a new operation, `append`, denoted as $\mathcal{Z}''.append(j'')$ and whose effect is $Z''(k) \leftarrow j''$. We are left with the problem of finding a data structure for a *dynamic permutation* $\mathcal{Z}'' : \{0, ..., k-1\} \to \{0, ..., k-1\}$ with support for `map` and `append` operations. Note that both domain and codomain size (k) are incremented by one after every `append` operation.

Example 19. *Let $k = 5$ and $\mathcal{Z}''$ be the permutation $\langle 3, 1, 0, 4, 2\rangle$. After $\mathcal{Z}''.append(2)$, k increases to 6 and $\mathcal{Z}''$ turns into the permutation $\langle 4, 1, 0, 5, 3, 2\rangle$. Note that $\mathcal{Z}''.append(j'')$ has the following effect on the permutation: all numbers larger than or equal to j'' are incremented by one, and j'' is appended at the end of the permutation.*

To implement the dynamic permutation $\mathcal{Z}''$, we slightly modify the SPSI data structure described at Section 3.3.1. We remind the reader that the SPSI on the sequence $s_1, \ldots, s_m$ is implemented using a red-black tree storing integers $s_1, \ldots, s_m$ in the leaves and partial sums and size counters of the subtrees in internal nodes. Let `PS` be the partial sum structure. We modify operation `PS.insert(i)` so that it returns the new red-black tree leaf x storing the integer that has just been inserted (we remind that we always insert 0). We moreover add a function `PS.locate(x)` taking as input a leaf in the red-black tree and returning the (0-based) position inside the partial sum associated with leaf x: if leaf x stores integer s_i, then `PS.locate(x)` returns $i - 1$. `PS.locate(x)` requires climbing the tree from x to the root and use subtree-size counters to retrieve the desired value, and therefore runs in $\mathcal{O}(\log m)$ time.

At this point, the dynamic permutation $\mathcal{Z}''$ is implemented using the partial sum structure `PS` described above and a vector N of red-black tree leaves supporting `append` operations (i.e. insert at the end of the vector). N can be implemented with a simple

vector of words with initial capacity 1. Every time we need to add an element beyond the capacity of N, we re-allocate $2|N|$ words for the array. N supports therefore constant-time access and amortized constant-time append operations. Starting with empty `PS` and N, we implement operations on $\mathcal{Z}''$ as follows:

- $\mathcal{Z}''.map(i)$ returns $PS.locate(N[i])$
- $\mathcal{Z}''.append(j)$ is implemented by calling $N.append(PS.insert(j))$

Note that the partial sum structure built with this approach contains only zeros: rather than using the *values* $s_1 = \cdots = s_m = 0$, we exploit their *positions* in the partial sum structure. Taking into account all components used to implement our original dynamic function $\mathcal{Z} : \{0, ..., n-1\} \rightarrow \{0, ..., n-1\}$, we get the bounds of our lemma. □

The algorithm

The steps of our algorithm to compute $RLBWT^+(\overleftarrow{T})$ from $LZ77(T)$ are:

1. We sort $\mathcal{D} = \{\pi \mid \langle \pi, \lambda, c \rangle \in LZ77(T) \ \wedge \pi \neq \perp\}$
2. We process $\langle \pi_v, \lambda_v, c_v \rangle_{v=1,...,z}$ from the first to last triple as follows. When processing $\langle \pi_v, \lambda_v, c_v \rangle$:
   (a) we use our dynamic function $\mathcal{Z}$ to convert text position π_v to RLBWT position $j' = \mathcal{Z}(\pi_v)$
   (b) we extract λ_v characters from RLBWT starting from position j' by using the LF function; at the same time, we extend RLBWT with the extracted characters.
   (c) when inserting a character at position j of the RLBWT, if j corresponds to some text position $i \in \mathcal{D}$, then we update $\mathcal{Z}$ accordingly by setting $\mathcal{Z}(i) \leftarrow j$. If, on the other hand, j does not correspond to any text position in $\mathcal{D}$, we execute $\mathcal{Z}.expand(j)$.

Our algorithm is described as Algorithm 19. A detailed description of the pseudocode and the proof of its complexity follow.

In Lines 1-5 we initialize all structures and variables. In order: we compute and sort set $\mathcal{D}$ of phrase sources, we initialize current text position i (i is the position of the character to be read), we initialize an empty RLBWT data structure (we will build $RLBWT^+(\overleftarrow{T})$ online), and we create an empty dynamic function data structure $\mathcal{Z}$. In Line 6 we enter the main loop iterating over LZ77 factors. If the current phrase's source is not empty (i.e. if the phrase copies a previous portion of the text), we need to extract λ_v characters from the RLBWT. First, in Line 8 we retrieve the RLBWT position j' corresponding to text position π_v with a `map` query on $\mathcal{Z}$. Note that, if $\pi_v \neq \perp$, then $i > \pi_v$ and therefore $\mathcal{Z}(\pi_v)$ is defined (see next). We are ready to extract characters from RLBWT. For λ_v times, we repeat the following procedure (Lines 10-19). We read the l-th character from the source of the v-th phrase (Line 10) and insert it in the RLBWT (Line 11). Importantly, the `extend` operation at Line 11 returns the RLBWT position j at which the new character is inserted; RLBWT position j correspond to text position i. We now have to check if i is the source of some LZ77 phrase. If this is the case (Line 12), then we link text position

i to RLBWT position j by calling a `assign` query on $\mathcal{Z}$ (Line 13). If, on the other hand, i is not the source of any phrase, then we call a `expand` query on $\mathcal{Z}$ on the codomain element j. Note that, after the `extend` query at Line 11, RLBWT positions after the j-th are shifted by one. If j' is one of such positions, then we increment it (Line 17). Finally, we increment text position i (Line 19). At this point, we finished copying characters from the v-th phrase's source (or we did not do anything if the v-th phrase consists of only one character). We therefore extend the RLBWT with the v-th trailing character (Line 20), and (as done before) associate text position i to RLBWT position j if i is the source of some phrase (Lines 21-24). We conclude the main loop by incrementing the current position i on the text (Line 25). Once all characters have been extracted from LZ77, RLBWT is a run-length BWT structure on $\overleftarrow{T}$. At Line 26 we convert it to $RLBWT^+(T)$ with Algorithm 18 and return it as a series of pairs $\langle \lambda_v, c_v \rangle_{v=1,\ldots,r}$.

Theorem 27. *Algorithm 19 converts $LZ77(T) \rightarrow RLBWT(T)$ in $\mathcal{O}(n(\log r + \log z))$ time and $\mathcal{O}(r+z)$ words of working space*

Proof. Sorting set $\mathcal{D}$ takes $\mathcal{O}(z \log z) \subseteq \mathcal{O}(n \log z)$ time. Overall, we perform $\mathcal{O}(z)$ `map`/`assign` and n `expand` queries on $\mathcal{Z}$. All these operations take overall $\mathcal{O}(n \log z)$ time. We use the dynamic RLBWT structure of Section 3.3.3 to implement $RLBWT^+(T)$ and $RLBWT^+(\overleftarrow{T})$. We perform n `access`, `extend`, and `LF` queries on $RLBWT^+(\overleftarrow{T})$. This takes overall $\mathcal{O}(n \log r)$ time. Finally, inverting $RLBWT^+(\overleftarrow{T})$ at Line 26 takes $\mathcal{O}(n \log r)$ time and $\mathcal{O}(r)$ words of space (Algorithm 18). We keep in memory the following structures: $\mathcal{D}$, $\mathcal{Z}$, $RLBWT^+(\overleftarrow{T})$, and $RLBWT^+(T)$. The bounds of our theorem easily follow. □

In Figure 5.2 we contextualize Theorem 27 in our framework of algorithmic tools (Figure 1.1).

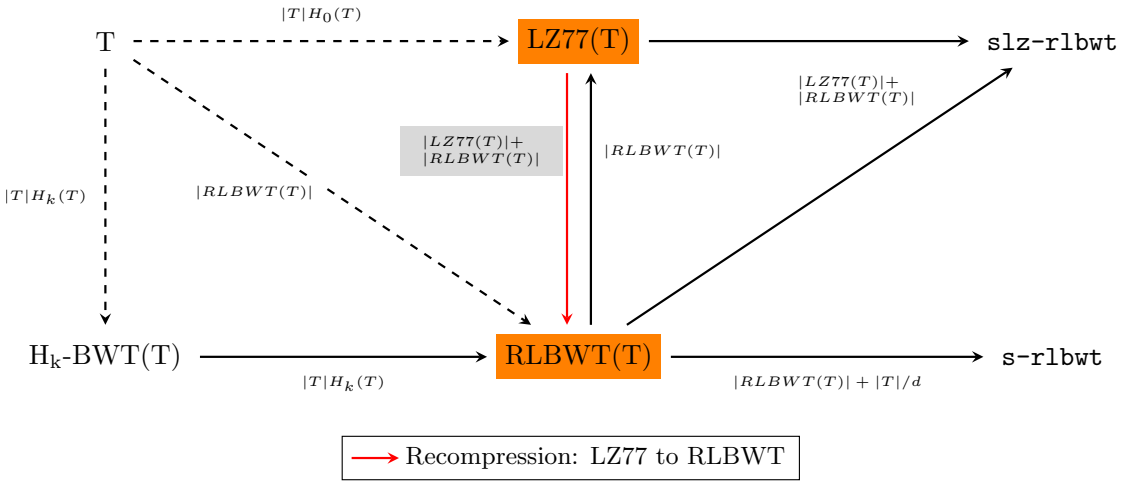


Figure 5.2: Our algorithm to convert $LZ77(T)$ to $RLBWT(T)$ uses a working space proportional to the sizes of a run-length encoded BWT and of the LZ77 parsing of T.

Algorithm 19: lz77_to_rlbwt($\langle \pi_v, \lambda_v, c_v \rangle_{v=1,\ldots,z}$)

input : LZ77 factorization $LZ77(T) = \langle \pi_v, \lambda_v, c_v \rangle_{v=1,\ldots,z}$ of a text T
output: RLBWT representation $\langle \lambda_v, c_v \rangle_{v=1,\ldots,r}$ of T

```
1  D ← {π | ⟨π, λ, c⟩ ∈ LZ77(T) ∧ π ≠ ⊥};                  /* Phrase sources */
2  sort(D);                                                 /* Sort phrase sources */
3  i ← 0;                                                   /* Current position on T */
4  RLBWT ← ϵ;                                               /* Init empty RLBWT of reversed text */
5  Z ← ∅;                                                   /* Init empty dynamic function structure */
6  for v = 1, ..., z do
7      if π_v ≠ ⊥ then
8          j' ← Z(π_v);                                     /* Map text position to RLBWT position */
9          for l = 1, ..., λ_v do
10             c ← RLBWT[j'];                               /* read char from source */
11             j ← RLBWT.extend(c);                         /* left-extend reverse text's RLBWT */
12             if i ∈ D then
13                 Z(i) ← j;                                /* j is the image of i */
14             else
15                 Z.expand(j);                             /* j does not have counter-image */
16             if j ≤ j' then
17                 j' ← j' + 1;                             /* new char falls before j' */
18             j' ← RLBWT.LF(j');
19             i ← i + 1;                                   /* Advance text position */
20     j ← RLBWT.extend(c_v);                               /* Extend with trailing character */
21     if i ∈ D then
22         Z(i) ← j;
23     else
24         Z.expand(j);
25     i ← i + 1;                                           /* Advance text position */
26 return reverse(RLBWT);                                   /* Build and return RLBWT(T) */
```

5.3 Indexes For Highly Repetitive Text Collections

As discussed in the previous sections, entropy compression is not able to capture long repetitions (asymptotically longer than the logarithm of the dataset size), therefore high-order compressed FM indexes represent a poor solution to the problem of compressing highly repetitive texts. Recent works [65, 66] showed that even indexes based on LZ78 are inappropriate to compress repetitive collections (recall, moreover, that LZ78 cannot achieve exponential compression). However, such datasets can—at the same time—be indexed and efficiently compressed with indexes based on LZ77 [65, 66, 67], run-length compression of the Burrows-Wheeler Transform [77, 111], and grammar compression [18,

19, 20, 115].

5.3.1 Related Work: RLBWT-, LZ-, and Grammar- Indexes

The first index tailored for highly repetitive sequences—the `rlcsa`—was proposed by Sirén et al. [77]. The `rlcsa`—discussed in Section 2.4.2—is based on the run-length encoding of the Ψ function and takes space proportional to the number r of equal-letter runs in the Burrows-Wheeler transform, plus the suffix array sampling. The latter component is the main reason why the `rlcsa` cannot achieve exponential compression while at the same time offering polylogarithmic-time locate queries: a sampling factor of k takes n/k words of space and supports locate queries in time proportional to k. A second drawback of the `rlcsa` is that its space is dominated by a $2\log(n/r)$-bits term: twice the space taken by a run-length encoding of the Burrows-Wheeler transform. Despite this (slight) space inefficiency, the `rlcsa` performs very well in practice (being very fast especially on `count` queries).

Kreft and Navarro [65, 66, 67] partially solved these problems by proposing an index—the `LZ77-index`—taking space proportional to the size z of the LZ77-compressed text. The main drawbacks of the `LZ77-index` are that (i) it supports locate queries in time quadratic with respect to the pattern length m and linear with respect to the parse height h (see Definition 11), and (ii) it does not support `count`; the only way to count occurrences is to locate them all. Moreover, in the LZ77 variant they consider (non self-overlapping LZ77 phrases), h can be as large as $\sqrt{n}$ so even `locate` can be inefficient on very repetitive texts. Also in this case however (see Chapter 7), the index performs well in practice on short patterns. This is due to a quite pessimistic worst-case analysis of the extraction cost and to the fact that h is small on average.

Indexes based on Straight-Line Programs [18, 19, 20, 115] achieve exponential compression and polylogarithmic-time queries. However, as discussed in Section 5.1, LZ77 is inherently more powerful than grammar compression, which in practice translates to the fact that grammar indexes are much bigger than LZ77 indexes.

An interesting compromise between LZ and grammar indexes is represented by the self-index based on *block trees* proposed very recently by Navarro [86]. By combining ideas from grammar and Lempel-Ziv compression, block trees achieve space proportional to $z\log(n/z)$, i.e. close to Lempel-Ziv compression. Moreover, they support text extraction in logarithmic time. The index described by Navarro [86] removes the h factor from count/locate times of the `LZ77-index` while occupying slightly more space. One of its drawbacks, however, remains the fact that query times are still quadratic in the pattern length.

In this section, we propose two indexes for highly repetitive text collections solving some of the problems raised above. Our first index—the `s-rlbwt` index—employs a new sparsification technique in the encoding of the run-length Burrows-Wheeler transform and achieves—modulo the suffix array sampling—optimal space with respect to a non-indexed RLBWT representation. Our second index—the `slz-rlbwt` index—solves also the problem related to the suffix array sampling by storing only one sample per LZ77 factor. The `slz-rlbwt` takes space proportional to $r + z$ and supports queries in logarithmic time w.r.t. the dataset size and linear time w.r.t. pattern length.

To improve readability, we drop the $(1 + o(1))$ multiplicative factor present in all space analyses.

5.3.2 The s-rlbwt Index

Recall, from Section 2.4.2, that the `rlcsa` index has a $r(2\log(n/r) + \log\sigma)$-bits leading term in its space usage. However, a run-length encoded Burrows-Wheeler transform can be implemented with a gap-encoded bitvector with r bits set (encoding run lengths) and a string storing run heads. These structures take $r(\log(n/r) + \log\sigma)$ bits of space (plus low-order terms), i.e. $r\log(n/r)$ bits *less* than the above indexes. The question we address in this section is, therefore: can we build an optimal-space (modulo the suffix array sampling) run-length encoded BWT index?

In Theorem 28 we answer affirmatively to this question and present our first index for highly repetitive text collections. Our idea is to apply *sparsification* to the main bitvector storing lengths of all runs in the run-length encoded string of Section 2.3.3. We call our index `s-rlbwt` (sparse RLBWT):

Theorem 28. *The `s-rlbwt` (sparse RLBWT) is a full-text self-index taking*

$$r\log(n/r) + r\log\sigma + \mathcal{O}(r) + (n/k)\log n$$

bits of space and supporting:

- *`Count` in* $\mathcal{O}\left(m(\log\frac{n}{r} + \log\sigma)\log^{\epsilon} n\right)$ *time*
- *`Locate` in* $\mathcal{O}\left((m + occ\cdot k)(\log\frac{n}{r} + \log\sigma)\log^{\epsilon} n\right)$ *time, and*
- *`Extract` in* $\mathcal{O}\left((m+k)(\log\frac{n}{r} + \log\sigma)\log^{\epsilon} n\right)$ *time*

For any constant $\epsilon > 0$.

Proof. we build the `rlbwt` index of Theorem 9 using the Elias-Fano indexable dictionary described in Theorem 2.1 of [94] to encode gap lengths. We recall that this indexable dictionary encodes a length-n bitvector with r bits set in $r\log(n/r) + \mathcal{O}(r)$ bits and supports `rank`, $\texttt{select}_1$, and `access` queries in $\mathcal{O}(\log(n/r))$ time. We moreover apply *sparsification* to the main bitvector V_{all} storing lengths of all runs (refer to Section 2.3.3 for a definition of V_{all}), i.e. we store just one out of $1/\delta$ bits set, where $0 < \delta \leq 1$. With this strategy, we are still able to answer all queries on the RLBWT by using the vectors V_c to reconstruct the positions of the missing ones in V_{all}, using $r\big(\log(n/r) + \delta\log(\frac{n}{r\delta}) + \log\sigma\big)$ bits of space, but query times are multiplied by a factor $1/\delta$. [2]

We choose $\delta = \log^{-\epsilon} n$ for any constant $\epsilon > 0$. As a result, $\delta\cdot r\log(\frac{n}{r\delta}) = o(r\log(n/r))$. The bounds of the theorem follow. □

In our implementation [101] we fix $\delta = 1/8$. We used `sdsl`'s `sd_vector` [43] for the Elias-Fano gap-encoded bitvectors. `sd_vector` implements the results described by

[2] The idea is the following. Let V'_{all} be the bitvector obtained from V_{all} by keeping only one out of $1/\delta$ bits set. To support $\texttt{rank}_1(i)$ on the original V_{all}, we first perform $\texttt{rank}_1(i)$ on V'_{all}. The result is multiplied by $1/\delta$ to obtain the rank j of the run (among all equal-letter runs) marked on V'_{all} with a bit set that immediately precedes position i. We then read at most $1/\delta$ run heads in H starting from position j and compute their lengths by performing a $\texttt{select}_1$ on bitvectors $V_{H[j]}, V_{H[j+1]}, ...$ using as arguments of $\texttt{select}_1$ the values $H.rank_{H[j]}(j), H.rank_{H[j+1]}(j+1), ...$, respectively. We accumulate these lengths on top of $V'_{all}.select_1(j)$, stopping as soon as the accumulated length reaches position i. Finally, we return j plus the number of run heads $H[j], H[j+1], ...$ seen until that point. The solution for answering $\texttt{select}_1$ on V_{all} follows the same logic and we do not report it here.

Okanohara and Sadakane [92] and uses $r \log(n/r) + 2r$ bits of space to encode a length-n bitvector with r bits set. As a result, in our implementation the hidden constant multiplying r in Theorem 28 is equal to 2. Theoretical query times are slightly higher than those of [94] for big values of r, but in practice are very fast (see [92] for full details). Excluding the suffix array sampling, the size of our index turns out to be almost always much smaller than that of the `rlcsa` [111] index (see Chapter 7).

5.3.3 The slz-rlbwt Index

There is a potential problem with the `s-rlbwt` index described in the previous section, namely, the suffix array sampling. Note that both query times and space of the index depend on the sampling rate k. For small k, if the text is very repetitive then the suffix array sampling could turn out to be exponentially bigger than the run-length encoded BWT. In order to make the index size always proportional to r, one could choose $k = n/r$. With this solution, the index could achieve exponential compression with small values of r; however, in such cases running times would become prohibitive (being the term $k = n/r$ close to n).

We need to find a better way to compress the suffix array sampling. The idea behind the result discussed in this section is to combine techniques from FM- and LZ-indexes in a single index in order to achieve this goal. We combine a RLBWT with a suffix array sampling based on LZ77; the result is an index whose size depends on both r and z, and whose query times are always polylogarithmic with respect to the dataset size, regardless of the compression rate. Three variants of this idea have been proposed in papers `(v)` and `(vi)`: `flz-rlbwt` (full index), `blz-rlbwt` (bidirectional index), and `slz-rlbwt` (sparse index). In this thesis we focus only on the third variant, which is by far the most space-efficient while still supporting very fast queries. In this section we first present our `slz-rlbwt` index and then—for completeness—give a quick overview of the other two variants.

We start by introducing a technique to *sparsify* the LZ77 parsing. Intuitively, the LZ77 parsing of a repetitive text collection $T_1T_2...T_k$ ($T_2, ..., T_k$ being variants of T_1) is much denser (i.e. presents many small phrases) inside T_1 than it is in $T_2, ..., T_k$. This suggests that excluding (big enough) contiguous regions from the parse (i.e. not outputting phrases inside these regions) should reduce the number of phrases in the denser regions of the collection. More formally:

Definition 18. ***LZ77-d parsing.*** *Let $d \geq 0$. Consider the following generalization of LZ77, denoted here as* LZ77-d. *We factor the text T as $X_1Y_1X_2Y_2...X_{z_d}Y_{z_d}$, where z_d is the* size *of the parse, $Y_1, ..., Y_{z_d} \in \Sigma^d$, and X_i is the longest prefix of $X_iY_i...X_{z_d}Y_{z_d}$ that appears at least twice in $X_1Y_1X_2Y_2...X_i$. In the case $d = 0$, we assume the text is (virtually) preceded by all characters in Σ*

Note that the version of LZ77 defined in Definition 2 corresponds to LZ77-1. Many works in the literature use, instead, the variant LZ77-0. On texts of practical interest, the LZ77-d parse produces a *dramatically* smaller parse than LZ77-0: in our experiments on twelve repetitive text collections (see Chapter 7), z_{512} was on average more than *ten times smaller* than z_0. We can now present our second index for highly repetitive text collections:

Theorem 29. *The* `slz-rlbwt` *(sparse Lempel-Ziv RLBWT) is a full-text self-index taking*

$$3z_d \log n + z_d \log(n/z_d) + r\left(\log(n/r) + \log\sigma + \mathcal{O}(1)\right)$$

bits of space and supporting:

- `Count` *in* $\mathcal{O}(m(\log(n/r) + \log\sigma)\log^{\epsilon} n)$ *time,*
- `Locate` *in* $\mathcal{O}\left((occ+1)\cdot(m+d)\cdot(\log(n/r)+\log\sigma)\log^{\epsilon} n + (occ+1)\log n\right)$ *time, and*
- `Extract` *in* $\mathcal{O}((m+d)\cdot(\log(n/r)+\log\sigma)\log^{\epsilon} n + \log z_d + (h+1)\cdot\log(n/z_d))$ *time*

For any constant $\epsilon > 0$ *and integer* $d \geq 0$. z_d *is the number of phrases of the LZ77-d parse.* h *is the parse height.*

Proof. Let us first define the index using the LZ77-0 parse. We keep $RLBWT^{+}(\overleftarrow{T})$ (the sparse variant used in the `s-rlbwt` index) and use it to find the RLBWT range of the (reversed) query pattern $P \in \Sigma^m$. We moreover sample the suffix array at the end of LZ77 factors. By definition, primary pattern occurrences cross a LZ77 factor. To locate primary occurrences, we can therefore forward-extract at most m characters from each position in the RLBWT range (forward extraction is implemented with *select* queries on RLBWT, see Section 2.3.3), stopping as soon as we find a suffix array sample (if there is one). Extracting m characters per occurrence takes overall $\mathcal{O}((occ+1)\cdot m\cdot(\log(n/r)+\log\sigma)\log^{\epsilon} n)$ time. In this phase, we need a gap-encoded bitvector END_T of $z\log(n/z)+\mathcal{O}(z)$ bits marking the last character of each LZ phrase on the *text*, a gap-encoded bitvector END_F of $z\log(n/z)+\mathcal{O}(z)$ bits marking the last character of each LZ phrase on the F column of $RLBWT(\overleftarrow{T})$, and a permutation $\mathcal{P}$ of the set $\{0,\ldots,z-1\}$ connecting corresponding bits set in END_F and END_T. $\mathcal{P}$ can be implemented with a wavelet tree (Section 2.3.4) so that it takes $z\log z(1+o(1))$ bits and supports operations (apply the permutation/inverse permutation to a $i \in \{0,\ldots,z-1\}$) in $\mathcal{O}(\log z)$ time. When finding a bit set on the F column of the BWT, we use $\mathcal{P}$ to find the corresponding bit set on END_T, therefore mapping the F position on the text ($\mathcal{O}(\log z)$ time for each primary occurrence). Finding primary occurrences takes, therefore, $\mathcal{O}((occ+1)\cdot m\cdot(\log(n/r)+\log\sigma)\log^{\epsilon} n+(occ+1)\log z)$ time.

After locating primary occurrences, we use a 2-sided range data structure to find all secondary occurrences as described in Section 2.4.3. Recall (from Section 2.4.3) that to implement this component we need a gap-encoded (Elias-Fano) bitvector $SOURCES$ of $z\log(n/z)+\mathcal{O}(z)$ bits of space marking with a bit set the first position of phrases sources on the text (i.e. locations from where phrases are copied), a succinct bitvector of z bits to mark duplicate sources, and a wavelet tree of $z\log n$ bits to store the actual 2D points (supporting operations in $\mathcal{O}(\log n)$ time). Overall, finding secondary occurrences takes $\mathcal{O}((occ+1)\log n)$ time.

It is easy to see that the above techniques still work if we replace LZ77-0 with LZ77-d, $d > 0$, provided that (i) we sample the suffix array at the end of each X_i, (ii) we build the 2-sided geometric structure on the (sources of the) phrases $X_1, ..., X_{z_d}$, and (iii) during locate we extract—in addition to the m pattern characters—also d characters before each pattern occurrence in order to locate phrases that start inside one of the strings Y_i. The bounds for `count`/`locate` of our theorem follow.

To conclude, we need to show how to support `extract` queries. To extract $T[i, \ldots, i+m-1]$, we distinguish three cases. (i) if $T[i, \ldots, i+m-1]$ overlaps a bit set on END_T, then we find the corresponding bit set in END_F using the permutation $\mathcal{P}$ ($\mathcal{O}(\log z_d)$ time) and extract the m characters from $RLBWT^+(\overleftarrow{T})$ using `LF` and `FL` queries to move forward/backwards in the text, respectively. (ii) If $T[i, \ldots, i+m-1]$ starts inside one of the strings Y_j of the LZ77-d parse (see Definition 18), then we find the bit set in END_T that immediately precedes string Y_j. Note that this bit set is within d positions from $T[i]$. We find the corresponding bit set in END_F using the permutation $\mathcal{P}$ and extract at most $m+d$ characters from the $RLBWT$ using `LF` queries to move forward in the text. (iii) $T[i, \ldots, i+m-1]$ is contained in one of the strings X_j, i.e. is entirely copied from a previous text location. Then—using bitvector END_T—we locate the rank and starting location in T of phrase X_j. We keep z_d integers of $\log z_d$ bits each to map each phrase rank to a bit set in $SOURCES$, indicating from where the phrase is copied in the text. By using these informations, in $\mathcal{O}(\log(n/z_d))$ time (required to answer queries on END_T) we can jump from $T[i, \ldots, i+m-1]$ to the location $T[i', \ldots, i'+m-1]$, with $i' < i$, from where $T[i, \ldots, i+m-1]$ is copied. Note that we need to jump at most h times—h being the parse height—until we fall into one of the cases (i) or (ii). The complexity of `extract` follows. □

As for the `s-rlbwt` index, in our implementation of the `slz-rlbwt` index [102] we fix $\delta = 1/8$. The hidden constant multiplying r in Theorem 29 is equal to 2 (we use the same RLBWT implementation of the `s-rlbwt` index). In our experiments (see Chapter 7), the size of the `slz-rlbwt` index was—on average—1.1 times larger than that of the most space-efficient variant of the LZ77 index described by Kreft and Navarro [67]: this proves that RLBWT indexes can be as space-efficient as LZ77 indexes on repetitive text collections. Our index, however, supports much (orders of magnitude) faster `count` queries (but slower `locate`) than the LZ77 index.

Enhancing memory usage during locate Note that, in order to locate occurrences, we need to recursively follow substring-copy chains starting from primary occurrences. This process naturally defines a forest of trees rooted in primary occurrences whose leaves are pattern occurrences that are no more copied forward in the text. Depending on how we visit these trees, the memory usage of our index during `locate` queries can vary dramatically. If the trees are BFS-visited, then the memory usage (on top of the index) is $\mathcal{O}(occ)$ words. If, instead, the trees are DFS-visited (streaming to disk found occurrences), the memory usage is $\mathcal{O}(h)$ words, h being the LZ77 parse height. On very repetitive texts occ can be very large, so the second strategy should be preferred. Our `slz-index` implements DFS-visit of pattern occurrences. In Chapter 7 we confirm that this strategy is very effective also in practice.

Variants We give a quick overview of the other two indexes combining LZ77 and RLBWT and described in papers `(v)` and `(vi)`. In the first index, named `flz-rlbwt` (full) and described in paper `(v)`, we consider all ways of breaking the pattern in a prefix and a suffix and (i) use $RLBWT^+(\overleftarrow{T})$ to find the lexicographic order the reversed pattern prefix among the suffixes of the reversed text that begin at LZ phrase boundaries, and (ii) use $RLBWT^+(T)$ plus a structure encoding the subset of suffix tree nodes corresponding to LZ phrases to find the lexicographic order the pattern suffix among all (forward) LZ77

factors. This strategy is equivalent to considering only the last phrase border in a pattern occurrence (in the case the occurrence contains more than one phrase borders); as a result, we report each pattern occurrence just once. These ranges are used—as described in Section 2.4.3—to retrieve all pattern primary occurrences (i.e. occurrences spanning a phrase border) by querying a 4-sided geometric range data structure. Secondary occurrences (i.e. occurrences entirely contained in a phrase) are retrieved—as described in Section 2.4.3—by querying a 2-sided geometric range data structure storing all phrases sources and end-points. In the second variant, dubbed `blz-rlbwt` (bidirectional), we drop $RLBWT^+(\overleftarrow{T})$ and simulate it using $RLBWT^+(T)$ and bidirectional search [68, 108].

In practice (see paper `(vi)`), the `flz-rlbwt` index is fast but too memory consuming. The `blz-rlbwt` index partially solves this problem by dropping $RLBWT^+(\overleftarrow{T})$, but still has to keep in memory heavy geometric data structures (the constant factors of these structures are not negligible). Moreover, bidirectional search increases `locate` query times, which become quadratic in the pattern length. As a result, in practice the `blz-rlbwt` index is prohibitively slow on `locate` queries while not improving by much the space occupancy of the full index.

Construction from a compressed file In this paragraph we prove that we can build our `slz-rlbwt` index in asymptotically optimal working space:

Theorem 30. *We can build the* `slz-rlbwt` *index in $\mathcal{O}(n(\log r + \log z))$ time and $\mathcal{O}(r+z)$ words of space taking as input any of the following: $LZ77(T)$, $RLBWT(T)$, $RLBWT(\overleftarrow{T})$, or the streamed text.*

Proof. Using the algorithms of Section 5.2, we can retrieve $RLBWT^+(\overleftarrow{T})$ in $\mathcal{O}(n(\log r + \log z))$ time and $\mathcal{O}(r+z)$ words of space starting from any of the above text representations. To compute LZ77-d(T), we can easily adapt the strategy described in Section 4.3: we just need—in Algorithm 14—to skip d characters after a phrase ends before starting searching the next phrase in the RLBWT index. Building bitvector END_F requires navigating $RLBWT^+(\overleftarrow{T})$ (with LF queries) and inserting phrase endpoints in a dynamic gap-encoded bitvector ($\mathcal{O}(n \log r + z_d \log z_d)$ time). Analogously, END_T can be built by scanning LZ77-d factors in text order. Permutation $\mathcal{P}$ can be built by scanning one more time $RLBWT^+(\overleftarrow{T})$ with LF queries, keeping track (in a vector of pairs) of corresponding bits set in END_F and END_T, and finally building the wavelet tree ($\mathcal{O}(n(\log r + log z_d) + z_d \log z_d)$ time). Finally, building the 2-sided geometric range structure requires first sorting LZ77-d factors with respect to their source field, computing the gap-encoded bitvector mapping text positions to $[0, z_d)$, computing the bitvector marking duplicate positions, and then inserting the computed points in the wavelet tree geometric structure ($\mathcal{O}(z_d \log n)$ time). In the end, if needed we can turn $RLBWT^+(\overleftarrow{T})$ and the gap-encoded bitvectors to static (and more space-efficient) data structures. □

In Figure 5.3 we contextualize Theorem 30 in our framework of algorithmic tools (Figure 1.1).

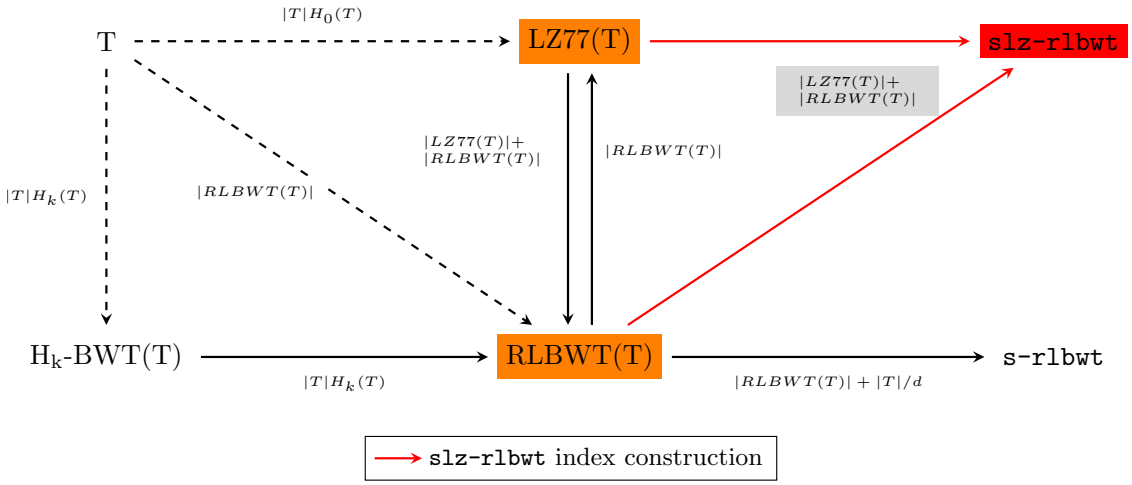


Figure 5.3: Our `slz-rlbwt` index can be constructed taking as input LZ77(T) and RLBWT(T). In turn, these two compressed representations can be computed in repetition-aware space with the algorithms of Sections 3.3 and 4.3.

6

From Theory to Practice: the DYNAMIC library

Many algorithms described in the literature work within provably-optimal running times and working space, but in practice they are based on too complicated data structures which prevent them to be competitive in practice. In the experimental world, asymptotic analysis loses some of its descriptive power and, quite often, is not able to accurately predict what the real performance of an algorithm will be. In this sense, it is not uncommon to see implementations of $\mathcal{O}(n \log n)$-time algorithms running faster than their linear-time counterparts. This is due to several factors that in practice play an important role but in theory are often poorly modeled: cache locality, branch prediction, disk accesses, context switches, memory fragmentation[1], and so on. A good implementation must take into account all these factors (e.g. through code profiling) in order to be practical.

6.1 Related Work

In recent years, several libraries implementing static data structures have been proposed, namely: `sdsl` [43] (probably the most used, comprehensive, and tested), `pizza&chili` [37] (compressed indexes), `sux` [120], `succinct` [95], `BWTIL` [96], `libcds` [15]. These libraries proved that *static* succinct data structures can be very practical in addition to being theoretically appealing. On the *dynamic* side, little work has been done. Dynamic data structures represent a challenge in practice, being based on components that are often cache-inefficient and memory-consuming (e.g. self-balancing trees) and cause severe memory fragmentation due to the numerous allocations and de-allocations introduced by dynamism (this effect can however be alleviated through appositely designed memory allocators). An interesting and promising (but still under development) step in this direction is represented by `Memoria` [113], a C++14 framework providing general purpose dynamic data structures. Other libraries are also still under development (`ds-vector` [25]) or have been published but the code is not available [21, 62]. To the best of our knowledge, the only working implementation of a succinct dynamic bitvector is [42]: a C++ container-like succinct data structure for storing a vector of bits with fast appending on both sides and

[1] *Memory fragmentation* is a scenario arising when data is allocated in multiple non-contiguous blocks interleaved by small empty segments of memory. Such small segments are unallocated but unusable for most typical scenarios (e.g. because they are too small), so are—to all effects—unusable. As we will see in Chapter 7, with dynamic data structures memory fragmentation can introduce overheads amounting to 24% of the allocated memory in the worst case.

fast insertion in the middle.

To sum up, the experimental community still lacks a stable and open-source library providing the most useful succinct dynamic data structures needed to implement several space-efficient indexing and compression algorithms (such as the ones presented in Chapters 3 and 4): dynamic partial sums, gap-encoded/succinct bitvectors, compressed wavelet trees, run-length encoded strings. In view of this gap between theoretical and practical advances in the field, in this chapter we present `DYNAMIC` [97]: a C++11 library providing implementations of the above mentioned succinct dynamic data structures. Our library has been extensively profiled and tested, and offers structures whose performance are provably close to the theoretical lower bounds (in particular, they approach succinctness and logarithmic queries). The core of the library is a searchable partial sum data structure with inserts (SPSI). This structure is used as building block for all other components: gap-encoded bitvectors (gaps are encoded as SPSI integers), succinct bitvectors (a binary SPSI), wavelet trees (fixed-length/gamma/Huffman encoding), run-length strings, and FM-indexes (run-length/entropy-compressed) supporting left-extension of the text. We conclude the chapter by describing how the algorithms of Chapters 3 and 4 have been implemented with `DYNAMIC`, including a detailed analysis of their space usage.

6.2 The Core: Searchable Partial Sums with Inserts

Recall, from Section 3.3.1, that the Searchable Partial Sums With Inserts (SPSI) problem asks for a data structure PS to maintain a sequence $s_1, \ldots, s_m$ of non-negative k-bits integers supporting the following operations:

- `PS.sum`$(i) = \sum_{j=1}^{i} s_j$;
- `PS.search`(x) is the smallest i such that $\sum_{j=1}^{i} s_j > x$;
- `PS.update`(i, δ): update s_i to $s_i + \delta$. δ can be negative as long as $s_i + \delta \geq 0$;
- `PS.insert`(i): insert 0 between s_{i-1} and s_i (if $i = 0$, insert in first position).

In this section we describe a practical variant of the partial sum data structure described in Section 3.3.1. Our practical solution offers space-time tradeoffs provably close to those of the best theoretical solutions described in the literature (which are often too complicated to be implemented in practice).

6.2.1 Data Structure

We employ B-trees (instead of red-black trees as done in Section 3.3.1). This choice improves cache-efficiency in that B-trees allow using a bigger fanout with respect to red-black trees (bigger internal nodes reduce tree height and can fit in a cache line). We use a leaf size l (i.e. number of integers stored in each leaf) always bounded by

$$0.5 \log m \leq l \leq \log m$$

and a node fanout $f \in \mathcal{O}(1)$. f has to be chosen accordingly with the cache line size; bigger f reduces cache misses and tree height. See Section 6.3 for a discussion on the maximum

leaf size and f values used in practice in our implementation. Letting $l = c \cdot \log m$ being the size of a particular leaf, we call the coefficient $0.5 \leq c \leq 1$ the *leaf load*.

In order to improve space usage even further while still guaranteeing very fast operations, integers in the leaves are packed contiguously in a word array and, inside each leaf $\mathcal{L}$, we assign to each integer the bit-size of the largest integer stored in $\mathcal{L}$. Whenever an integer overflows the maximum size associated to its leaf (after an `update` operation), we re-allocate space for all integers in the leaf. This operation takes $\mathcal{O}(\log m)$ time, so it does not asymptotically increase the cost of `update` operations. Crucially, in each leaf we allocate space only for the integers actually stored inside it, and re-allocate space for the whole leaf whenever we insert a new integer or we split the leaf. With this strategy, we do not waste space for half-full leaves[2]. Note moreover that, since the size of each leaf is bounded by $\Theta(\log m)$, re-allocating space for the whole leaf at each insertion does not asymptotically slow down `insert` operations.

6.2.2 Theoretical Guarantees

Let us denote with $m/\log m \leq L \leq 2m/\log m$ the total number of leaves, with $\mathcal{L}_j$, $0 \leq j < L$, the j-th leaf of the B-tree (using any leaf order), and with $I \in \mathcal{L}_j$ an integer belonging to the j-th leaf. The total number of bits stored in the leaves of the tree is

$$\sum_{0 \leq j < L} \sum_{I \in \mathcal{L}_j} max\_bitsize(\mathcal{L}_j)$$

where $max\_bitsize(\mathcal{L}_j) = \max_{I \in \mathcal{L}_j}(bitsize(I))$ is the bit-size of the largest $I \in \mathcal{L}_j$, and $bitsize(x) = \lfloor \log_2 x \rfloor + 1$ is the number of bits required to write number x in binary. The above quantity is equal to

$$\sum_{0 \leq j < L} c_j \cdot \log m \cdot max\_bitsize(\mathcal{L}_j)$$

where $0.5 \leq c_j \leq 1$ is the j-th leaf load. Since leaves loads are always upper-bounded by 1, the above quantity is upper-bounded by

$$\log m \sum_{0 \leq j < L} max\_bitsize(\mathcal{L}_j)$$

which, in turn, is upper-bounded by

$$\log m \sum_{0 \leq j < L} bitsize\left(\sum_{I \in \mathcal{L}_j} I\right) \leq \log m \sum_{0 \leq j < L} 1 + \log_2\left(1 + \sum_{I \in \mathcal{L}_j} I\right)$$

In the above inequality, we use the upper-bound $bitsize(x) \leq 1 + \log_2(1 + x)$ to deal with the case $x = 0$. Let $M = m + \sum_{i=1}^{m} s_i = m + \sum_{0 \leq j < L} \sum_{I \in \mathcal{L}_j} I$ be the sum of all integers stored in the structure, plus m. From the concavity of log and from $L \leq 2m/\log m$, it can be derived that the above quantity is upper-bounded by

$$2m \cdot (\log(M/m) + \log\log m + 1) \tag{6.1}$$

To conclude, we store $\mathcal{O}(1)$ pointers/counters of $\mathcal{O}(\log M)$ bits each per leaf and internal node. We obtain:

[2] In practice, to speed up operations we allow a small fraction of the leaf to be empty

Theorem 31. *Let $s_1, \ldots, s_m$ be a sequence of m non-negative integers and $M = m + \sum_{i=1}^{m} s_i$. The partial sum data structure implemented in* `DYNAMIC` *takes at most*

$$2 \cdot m \left(\log(M/m) + \log\log m + \mathcal{O}(\log M / \log m)\right)$$

bits of space and supports `sum`, `search`, `update`, *and* `insert` *operations on the sequence $s_1, \ldots, s_m$ in $\mathcal{O}(\log m)$ time.*

In our experiments we observed that—even taking into account memory fragmentation—the bit-size of our dynamic partial sum structure is well approximated by function $1.19 \cdot m \left(\log(M/m) + \log\log m + \log M / \log m\right)$. See Chapter 7 for full details.

6.3 Plug and Play with Dynamic Structures

The SPSI structure described in the previous section can be used as building block to obtain all dynamic structures used in our algorithms. In our library, the SPSI structure's type name is `spsi` and is parametrized on 3 template arguments: the leaf type (here, the type `packed_vector` is always used[3]), the leaf size and the node fanout. `DYNAMIC` defines two SPSI types with two different combinations of these parameters:

```
typedef spsi<packed_vector,256,16> packed_spsi;
typedef spsi<packed_vector,8192,16> succinct_spsi;
```

The reasons for the particular values chosen for the leaf size and node fanout will be explained later. We use these two data types as basic component in the definition our structures. To improve readability, in the following we drop the $(1 + o(1))$ multiplicative term present in all space analyses.

6.3.1 Gap-Encoded Bitvectors

`DYNAMIC` implements gap-encoded bitvectors using a SPSI to encode gap lengths. Recall, from Section 3.3.2, that this solution permits to support also $\texttt{delete}_0$ operations (which require just to decrement one of the SPSI's counters). `DYNAMIC`'s name for the dynamic gap-encoded bitvector class is `gap_bitvector`. The class is a template on the SPSI type. We plug `packed_spsi` in `gap_bitvector` as follows:

```
typedef gap_bitvector<packed_spsi> gap_bv;
```

and obtain:

Theorem 32. *Let $B \in \{0,1\}^n$ be a bit-sequence with b bits set. The dynamic gap-encoded bitvector* `gap_bv` *implemented in* `DYNAMIC` *takes at most*

$$2 \cdot b \left(\log(n/b) + \log\log b + \mathcal{O}(\log n / \log b)\right)$$

bits of space and supports `rank`, `select`, `access`, `insert`, *and* $\texttt{delete}_0$ *operations on B in $\mathcal{O}(\log b)$ time.*

[3] `packed_vector` is simply a packed vector of k-bits integers supporting all SPSI operations in linear time

In our experiments, the optimal node fanout for the SPSI stucture employed in this component turned out to be 16, while the optimal leaf size 256 (these values represented a good compromise between query times and space usage). In our experiments we observed that the bit-size of our dynamic gap-encoded bitvector is well approximated by function $1.19 \cdot b\,(\log(n/b) + \log\log b + \log n/\log b)$. See Chapter 7 for full details.

6.3.2 Succinct Bitvectors and Compressed Strings

Let n be the bitvector length. Dynamic succinct bitvectors can be implemented using a SPSI where all $m = n$ stored integers are either 0 or 1. At this point, `rank` operations on the bitvector correspond to `sum` on the partial sum structure, and `select` operations on the bitvector can be implemented with `search` on the partial sum structure[4]. `access` and `insert` operations on the bitvector correspond to exactly the same operations on the partial sum structure. Note that in this case we can accelerate operations in the leaves by a factor of $\log n$ by using constant-time built-in bitwise operations such as `popcount`, masks and shifts. This allows us to use bigger leaves containing $\Theta(\log^2 n)$ bits, which results in a total number of internal nodes bounded by $\mathcal{O}(n/\log^2 n)$. The overhead for storing internal nodes is therefore of $o(n)$ bits. Moreover, since in the leaves we allocate only the *necessary* space to store the bitvector's content (i.e. we do not allow empty space in the leaves), it easily follows that the dynamic bitvector structure implemented in `DYNAMIC` takes n bits of space and supports all operations in $\mathcal{O}(\log n)$ time.

In our experiments, the optimal node fanout for the SPSI stucture employed in the succinct bitvector structure turned out to be 16, while the optimal leaf size 8192. `DYNAMIC`'s name for the dynamic succinct bitvector is `succinct_bitvector`. The class is a template on the SPSI type. `DYNAMIC` defines its dynamic succinct bitvector type as:

```
typedef succinct_bitvector<succinct_spsi> suc_bv;
```

We obtain:

Theorem 33. *Let $B \in \{0,1\}^n$ be a bit-sequence. The dynamic succinct bitvector data structure `suc_bv` implemented in `DYNAMIC` takes n bits of space and supports `rank`, `select`, `access`, and `insert` operations on B in $\mathcal{O}(\log n)$ time.*

In our experiments we observed (see Chapter 7) that the size of our dynamic succinct bitvector is always upper-bounded by $1.23 \cdot n$ bits. The 23% overhead on top of the optimal size comes mostly from memory fragmentation (16%). The remaining 7% comes from succinct structures on top of the bit-sequence.

Dynamic compressed strings are implemented with a wavelet tree built upon dynamic succinct bitvectors (see Section 2.3.2). We explicitly store the topology of the tree ($\mathcal{O}(|\Sigma|\log n)$ bits) instead of encoding it implicitly in a single bitvector. This choice is space-inefficient for very large alphabets, but reduces the number of `rank/select` operations on the bitvector(s). `DYNAMIC`'s compressed strings (wavelet trees) are a template on the bitvector type. `DYNAMIC` defines its dynamic string type as:

```
typedef wt_string<suc_bv> wt_str;
```

When using Huffman topology, the implementation satisfies:

[4] Actually, `search` permits to implement only $\texttt{select}_1$. $\texttt{select}_0$ can however be easily simulated with the same solution used for `search` by replacing each integer $x \in \{0,1\}$ with $1-x$ at run time. This solution does not increase space usage.

Theorem 34. *Let $S \in \Sigma^n$ be a string with zero-order entropy equal to H_0. The Huffman-compressed dynamic string data structure* `wt_str` *implemented in* `DYNAMIC` *takes*

$$n(H_0 + 1) + \mathcal{O}(|\Sigma| \log n)$$

bits of space and supports `rank`, `select`, `access`, *and* `insert` *operations on S in average $\mathcal{O}((H_0 + 1) \log n)$ time.*

The user can choose at construction time whether to use a Huffman, fixed-size, or gamma encoding for the alphabet. Gamma encoding is useful when the alphabet size is unknown at construction time. In the case a fixed-size encoding is used (i.e. $\lceil \log_2 |\Sigma| \rceil$ bits per character), the structure takes $n \log |\Sigma| + \mathcal{O}(|\Sigma| \log n)$ bits and supports all operations in $\mathcal{O}(\log |\Sigma| \cdot \log n)$ time.

Run-Length Encoded Strings

In our library, run-length compressed strings are a template on a gap-encoded bitvector type (encoding run lengths) and on a dynamic string type (encoding run heads). We plug the structures of the previous sections in an implementation of the run-length encoded dynamic string described in Section 2.3.3 (template class `rle_string`) as follows:

```
typedef rle_string<gap_bv, wt_str> rle_str;
```

and obtain

Theorem 35. *Let $S \in \Sigma^n$ be a string with r_S equal-letter runs. The dynamic run-length encoded string data structure* `rle_str` *implemented in* `DYNAMIC` *takes*

$$r_S \cdot (4 \log(n/r_S) + \log |\Sigma| + 4 \log \log r_S + \mathcal{O}(\log n / \log r_S)) + \mathcal{O}(|\Sigma| \log n)$$

bits of space and supports `rank`, `select`, `access`, *and* `insert` *operations on S in $\mathcal{O}(\log |\Sigma| \cdot \log r_S)$ time.*

6.3.3 Dynamic FM-Indexes

We can obtain a dynamic FM-index by encoding the Burrows-Wheeler transform with any dynamic string data structure and adding a sparse dynamic vector storing the suffix array sampling on top of it. The dynamic sparse vector can be implemented with a dynamic bitvector and a dynamic vector of integers (e.g. a SPSI). In `DYNAMIC`, the BWT is a template class parametrized on the L-column and F-column types. For the F column, a run-length encoded string is always used. `DYNAMIC` defines two types of dynamic Burrows-Wheeler transform structures (wavelet-tree/run-length encoded):

```
typedef bwt<wt_str, rle_str> wt_bwt;
typedef bwt<rle_str, rle_str> rle_bwt;
```

Dynamic sparse vectors are implemented inside the FM index class using a dynamic bitvector marking sampled BWT positions and a dynamic sequence of integers (a SPSI) storing non-null values. We combine a Huffman-compressed BWT with a succinct bitvector and a SPSI:

```
typedef fm_index<wt_bwt, suc_bv, packed_spsi> wt_fmi;
```

and obtain

Theorem 36. *Let $S \in \Sigma^n$ be a string with zero-order entropy equal to H_0, $P \in \Sigma^m$ a pattern occurring occ times in T, and k the suffix array sampling rate. The dynamic Huffman-compressed FM-index* `wt_fmi` *implemented in* `DYNAMIC` *takes*

$$n(H_0 + 2) + \mathcal{O}(|\Sigma| \log n) + (n/k) \log n$$

bits of space and supports:

- `access` *to BWT characters in average* $\mathcal{O}((H_0 + 1) \log n)$ *time*
- `count` *in average* $\mathcal{O}(m(H_0 + 1) \log n)$ *time*
- `locate` *in average* $\mathcal{O}((m + occ \cdot k)(H_0 + 1) \log n)$ *time*
- *text* `left-extension` *in average* $\mathcal{O}((H_0 + 1) \log n)$ *time*

If a plain alphabet encoding is used, all (H_0+1) *terms are replaced by* $\log |\Sigma|$ *and times become worst-case.*

If, instead, we combine a run-length compressed BWT with a gap-encoded bitvector and a SPSI as follows:

```
typedef fm_index<rle_bwt, gap_bv, packed_spsi> rle_fmi;
```

we obtain

Theorem 37. *Let $S \in \Sigma^n$ be a string whose BWT has r runs, $P \in \Sigma^m$ a pattern occurring occ times in T, and k the suffix array sampling rate. The dynamic run-length compressed FM-index* `rle_fmi` *implemented in* `DYNAMIC` *takes*

$$r \cdot (4 \log(n/r) + \log |\Sigma| + 4 \log \log r + \mathcal{O}(\log n / \log r)) + \mathcal{O}(|\Sigma| \log n) + (n/k) \log n$$

bits of space and supports:

- `access` *to BWT characters in* $\mathcal{O}(\log |\Sigma| \cdot \log r)$ *time*
- `count` *in* $\mathcal{O}(m \cdot \log |\Sigma| \cdot \log r)$ *time*
- `locate` *in* $\mathcal{O}((m + occ \cdot k)(\log |\Sigma| \cdot \log r))$ *time*
- *text* `left-extension` *in* $\mathcal{O}(\log |\Sigma| \cdot \log r)$ *time*

The suffix array sample rate k can be chosen at construction time.

6.4 Compression Algorithms, in Practice

In this section we describe how the algorithms of Chapters 3 an 4 have been implemented using `DYNAMIC`, including a detailed analysis of their space usage. As done in the previous section, to improve readability we omit the $(1 + o(1))$ multiplicative term present in all space analyses.

6.4.1 cw-bwt: High-Order Compressed BWT

Recall that `cw-bwt` maintains a de Bruijn automaton of degree k and stores, for each automaton's state, a partial sum structure and a dynamic zero-order compressed string. In `DYNAMIC`, the automaton is implemented with two vectors of size σ^k storing, respectively, partial sum structures and dynamic zero-order compressed strings. The length-k strings corresponding to each automaton's state are encoded implicitly in the indexes—seen as integers in base σ— of the two arrays. The i-th partial sum structure is implemented—as described in Section 3.2—with a packed B-tree storing σ_i integers whose maximum size is that of context i, where σ_i is the effective alphabet of the i-th context. Since we use packed computation, this strategy speeds up operations considerably if the context size is small (because more integers fit in a machine word). Overall, all partial sums structures take $\mathcal{O}(\sigma^{k+1} \log n)$ bits of space.

Zero-order compressed strings are implemented with the structure of Theorem 34. Since each string represents a BWT context, we achieve high-order compression: overall, the dynamic strings take $(nH_k + n)$ bits of space. Note that in `DYNAMIC` we do not implement the dynamic bitvector—guaranteeing constant-time operations on strings of size $w^{\mathcal{O}(1)}$—described in Section 3.2 (this bitvector has however been implemented in the library [42]). This choice gives—in theory—a better worst-case time (w.r.t the algorithm described in the original paper), but a worse average-case time (see below). In practice, the solution using `DYNAMIC`'s data structures is faster than the original one, which has also been implemented and can be found in the `BWTIL` library [96].

In order to find the optimal context size k (which should be close to $k = \log_\sigma(n/\log^2 n) - 1$, according to Section 3.2), we scan multiple times a (small enough) prefix of the text and estimate the overhead introduced by the partial sums for $k = 0, 1, 2, \ldots$. This procedure stops as soon as this overhead exceeds $0.1 \cdot n \log \sigma$ bits. We obtain:

Theorem 38. *The `cw-bwt` algorithm implemented in `DYNAMIC` computes the Burrows-Wheeler transform of a text $T \in \Sigma^n$ within the following bounds:*

- $n(H_k + 1) + o(n \log |\Sigma|) + \mathcal{O}(|\Sigma| \log n)$ *bits of space*
- $\mathcal{O}(n(H_k + 1) \log \log n)$ *average-case time*
- $\mathcal{O}(n(H_k + 1) \log n)$ *worst-case time*

Where H_k is the k-th order entropy of T and $k = \min\{1, \log_\sigma(n/\log^2 n) - 1\}$

In our implementation, the context size k must be at least 1 for technical reasons. The average-case time comes from the observation that, in the average case, each context's size is $\log^{\mathcal{O}(1)} n \leq w^{\mathcal{O}(1)}$. While in this case the dynamic bitvector of Section 3.2 (implemented in [42]) guarantees constant-time operations, `DYNAMIC`'s bitvector supports operations in logarithmic time w.r.t. the bitvector length, i.e. $\mathcal{O}(\log \log^{\mathcal{O}(1)} n) = \mathcal{O}(\log \log n)$ time in the average case.

6.4.2 rle-bwt: Run-Length Compressed BWT

`DYNAMIC`'s algorithm building the BWT in run-compressed space implements the BWT construction algorithm of Section 2.5 using a run-length BWT implemented with the structure of Theorem 35. We obtain:

Theorem 39. *The* `rle-bwt` *algorithm implemented in* `DYNAMIC` *builds the Burrows-Wheeler transform of a text $T \in \Sigma^n$ within the following bounds:*

- $r \cdot (4\log(n/r) + \log|\Sigma| + 4\log\log r + \mathcal{O}(\log n/\log r)) + \mathcal{O}(|\Sigma|\log n)$ *bits of space*
- $\mathcal{O}(n \log r \log|\Sigma|)$ *worst-case time*

Where r is the number of equal-letter runs in the Burrows-Wheeler transform of T.

6.4.3 h0-lz77: Zero-Order Compressed LZ77

Recall that our algorithm computing LZ77 in zero-order compressed working space (see Section 4.2) requires just a zero-order compressed FM-index supporting left-extension of the text. To this end, in `DYNAMIC` we use the FM index of Theorem 36 with default sampling rate $k = 256$ (which can be considered $\Theta(\log_\sigma n \log\log n)$—i.e. the rate chosen in Section 4.2—for small σ and all practical values of n). This yields:

Theorem 40. *The* `h0-lz77` *algorithm implemented in* `DYNAMIC` *computes the LZ77 factorization of a text $T \in \Sigma^n$ within the following bounds:*

- $n(H_0 + 2) + o(n\log|\Sigma|) + \mathcal{O}(|\Sigma|\log n)$ *bits of space*
- $\mathcal{O}(n(H_0+1)\log n \cdot \log\log n)$ *worst-case time*

Where H_0 is the zero-order entropy of T.

The multiplicative factor of $(H_0+1)\log\log n$ in running times comes from the fact that our dynamic string is slower than the one (not implemented) used in Section 4.2 (which supports all operations in $\mathcal{O}(\log n/\log\log n)$ time).

6.4.4 rle-lz77: Run-Length Compressed LZ77

Our two algorithms to compute LZ77 with a run-length compressed BWT (Section 4.3) share a RLBWT data structure over $\overleftarrow{T}$ implemented with the run-length string of Theorem 35.

Our first algorithm `rle-lz77-1` augments the RLBWT with a map SA associating a (sparse) dynamic vector $SA[c]$ of suffix array samples to each $c \in \Sigma$. Each $SA[c]$ is implemented with a gap-encoded bitvector (Theorem 32) marking BWT positions containing character c *and* a suffix array sample, plus a SPSI (Theorem 31) storing the actual samples. Overall, the $|\Sigma|$ gap-encoded bitvectors span $\sigma \cdot n$ positions and contain at most $2r$ bits set. This yields an overall space occupancy of $4r\cdot(\log(n/r) + \log\sigma + \log\log r + \mathcal{O}(\log n/\log r))$ bits for these components. The SPSI structures store overall at most $2r$ integers in the range $[0, n)$ (total sum: at most $M = 2rn$). Plugging these values in the structure of Theorem 31, we obtain that the overall size of the SPSI structures is at most $4r \cdot (\log n + \log\log r + \mathcal{O}(\log n/\log r))$ bits. Table 6.1 recapitulates the space of all data structures used by `rle-lz77-1`.

Adding up all these terms, we obtain:

Theorem 41. *The* `rle-lz77-1` *algorithm implemented in* `DYNAMIC` *computes the LZ77 factorization of a text $T \in \Sigma^n$ within the following bounds:*

	$r \log n$	$r \log(n/r)$	$r \log \lvert\Sigma\rvert$	$r \log \log r$	$r \log n / \log r$	$\lvert\Sigma\rvert \log n$
RLBWT	0	4	1	4	$\mathcal{O}(1)$	$\mathcal{O}(1)$
bitvectors	0	4	4	4	$\mathcal{O}(1)$	0
SPSI	4	0	0	4	$\mathcal{O}(1)$	0
TOTAL	4	8	5	12	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Table 6.1: Space occupancy of the data structures used by `rle-lz77-1`

- $r \cdot \left(4 \log n + 8 \log \frac{n}{r} + 5 \log |\Sigma| + 12 \log \log r + \mathcal{O}(\frac{\log n}{\log r})\right) + \mathcal{O}(\Sigma \log n)$ *bits of space*
- $\mathcal{O}(n \log r \log |\Sigma|)$ *worst-case time*

Where r is the number of equal-letter runs in the Burrows-Wheeler transform of $\overleftarrow{T}$.

Note that the above space bound is about $12|RLBWT|$, i.e. 12 times the size of a run-length encoded BWT.

The idea behind our second algorithm `rle-lz77-2` is to mark RLBWT positions associated with the source of a LZ77 factor during a first scan of the Burrows-Wheeler transform, and then outputting the LZ77 factors during a second pass. To save time, in our implementation we do not delete and re-build the RLBWT as done in our original algorithm (this in practice would take too much time). This slightly increases space usage (read below) as we need to store lengths and trailing characters of LZ77 factors during the first scan. We keep the following data structures. (1) A gap-encoded bitvector $FPOS[0, \ldots, n-1]$ marking with a bit set F-positions that are the source of at least a LZ77 factor. We use the structure of Theorem 32 for this component, for a total space occupancy of $2z \cdot (\log(n/z) + \log \log z + \mathcal{O}(\log n / \log z))$ bits. (2) a succinct bitvector $REP[0, \ldots, z-1]$ taking z bits of space and keeping track of factors sharing the same source: a bit set in $FPOS$ representing the source of k distinct LZ77 factors is marked in this bitvector with the pattern 10^{k-1}. (3) a dynamic SPSI $PTR[0, \ldots, z-1]$ associating ranks of LZ77 factors to their corresponding bit in REP (this component stores therefore a permutation of $[0, z)$). We use the structure of Theorem 31 for this component, for a total space occupancy of $2z \cdot (\log z + \log \log z + \mathcal{O}(1))$ bits. Finally, we need three vectors $\pi[0, \ldots, z-1]$, $\lambda[0, \ldots, z-1]$, and $trail[0, \ldots, z-1]$ storing phrase sources, lengths, and trailing characters of each factor, respectively. These vectors can be implemented with three SPSI using overall $z(\log n + \log \log z) + z(\log \sigma + \log \log z) + \mathcal{O}(z)$ bits of space while supporting $\mathcal{O}(\log z)$-time operations. For simplicity, we consider $\epsilon = 0$. Table 6.2 recapitulates the space of all data structures—except RLBWT—used by `rle-lz77-2`.

	$z \log n$	$z \log(n/z)$	$z \log \lvert\Sigma\rvert$	$z \log \log z$	z	$z \log n / \log z$	$z \log z$
FPOS	0	2	0	2	0	$\mathcal{O}(1)$	0
REP	0	0	0	0	1	0	0
PTR	0	0	0	2	$\mathcal{O}(1)$	0	2
$\pi, \lambda, trail$	2	0	1	2	$\mathcal{O}(1)$	0	0
TOTAL	2	2	1	6	$\mathcal{O}(1)$	$\mathcal{O}(1)$	2

Table 6.2: Space occupancy of the data structures (excluding RLBWT) used by `rle-lz77-2`

Adding up all these terms (we group $2\log(n/z) + 2\log z = 2\log n$ and hide the $\mathcal{O}(z)$ term inside $\mathcal{O}(z\frac{\log n}{\log z})$), we obtain:

Theorem 42. *The* `rle-lz77-2` *algorithm implemented in* `DYNAMIC` *computes the LZ77 factorization of a text $T \in \Sigma^n$ within the following bounds:*

- $r \cdot \left(4\log\frac{n}{r} + \log|\Sigma| + 4\log\log r + \mathcal{O}(\frac{\log n}{\log r})\right) + z \cdot \left(4\log n + \log|\Sigma| + 6\log\log z + \mathcal{O}(\frac{\log n}{\log z})\right) + \mathcal{O}(|\Sigma|\log n)$ *bits of space*
- $\mathcal{O}\left(n(\log r \log|\Sigma| + \log z)\right)$ *worst-case time*

Where r is the number of equal-letter runs in the Burrows-Wheeler transform of $\overleftarrow{T}$ and z is the number of LZ77 factors.

Note that the above space bound is about $4|RLBWT| + 4|LZ77|$, i.e. 4 times the size of a run-length encoded BWT plus 4 times the size of the LZ77 parsing. This is much less than the space—$12|RLBWT|$—taken by `rle-lz77-1`, especially considering that, in practice, z is often much smaller than r. It should therefore not be surprising that—see Chapter 7—`rle-lz77-2` outperforms `rle-lz77-1` in both working space and running times.

7

Experimental Results

This chapter is devoted to experimental results. We first present (Section 7.1) an experimental evaluation of `DYNAMIC`'s succinct and gap-encoded bitvectors (standing at the core of all other library's data structures). In Section 7.2 we compare our compression algorithms and indexes with the state of the art on repetitive text collections. All experiments have been performed on a `intel core i7` machine with 12 GB of RAM running Linux Ubuntu 16.04.

7.1 DYNAMIC: Benchmarks

We built 34 gap-encoded (`gap_bv`) and 34 succinct (`suc_bv`) bitvectors containing $n = 500 \cdot 10^6$ bits, varying the frequency b/n of bits set in the interval $[0.0001, 0.99]$. In each experiment, we first built the bitvector by performing n $\texttt{insert}_b$ queries, b being equal to 1 with probability b/n, at uniform random positions. After building the bitvector, we executed n $\texttt{rank}_0$, n $\texttt{rank}_1$, n $\texttt{select}_0$, n $\texttt{select}_1$, and n `access` queries at uniform random positions. Running times of each query were averaged over the n repetitions. We measured memory usage in two ways: (i) internally by counting the total number of bits allocated by our procedures—this value is denoted as *allocated* memory in our plots—, and (ii) externally using the tool `/usr/bin/time`—this value is denoted as *RSS* in our plots (Resident Set Size).

7.1.1 Working Space

We fitted measured RSS memory with the theoretical predictions of Section 6.2.2 using a linear regression model. Parameters of the model were inferred using the statistical tool `R` (function `lm`). In detail, we fitted RSS memory in the range $b/n \in [0, 0.1]$[1] with function $k \cdot f(n, b) + c$, where: $f(n, b) = b \cdot (\log(n/b) + \log\log b + \log n / \log b)$ is our theoretical prediction (recall that memory occupancy of our gap-encoded bitvector should never exceed $2f(n, b)$), k is a scaling factor accounting for memory fragmentation and average load distribution in the B-tree, and c is a constant accounting for the weight of loaded C++ libraries (this component cannot be excluded from the measurements of the tool `/usr/bin/time`). Function `lm` provided us with parameters $k = 1.19$ and $c = 28758196$ bits $\approx 3.4MB$. The value for c was consistent with the space measured with b/n close to 0.

[1] For $b/n \geq 0.1$ it becomes more convenient—see below—to use our succinct bitvector, so we considered it more useful to fit memory usage in $b \in [0, 0.1]$. In any case—see plot 7.1—the inferred model well fits experimental data in the (more wide) interval $b/n \in [0, 0.7]$.

All plots were generated using `R`. Figures 7.1, 7.2, and 7.3 show memory occupancy of `DYNAMIC`'s bitvectors as a function of the frequency b/n of bits set. In Figure 7.1 we compare both bitvectors. In Figures 7.2 and 7.3 we show separately our gap-encoded and succinct bitvectors (in the first case, we focus on the interval $b/n \in [0, 0.1]$). In Figures 7.1 and 7.2 we moreover show the growth of function $1.19 \cdot f(n, b) + 28758196$. Plot in Figure 7.1 shows that our theoretical prediction fits almost perfectly the memory usage of our gap-encoded bitvector for $b/n \leq 0.7$. The plot suggests moreover that for $b/n \geq 0.1$ it is preferable to use our succinct bitvector rather than the gap-encoded one. As far as the gap-encoded bitvector is concerned, memory fragmentation[2] amounts to approximately 15% of the allocated memory for $b/n \leq 0.5$. This fraction increases to 24% for b/n close to 1. Plot in Figure 7.3 shows memory usage of our succinct bitvector. As expected, memory usage is independent of b/n (except small oscillations due to load distribution in the B-tree and memory fragmentation). Note that RSS memory never exceeds $1.29n$ bits: the overhead of $0.29n$ bits is distributed among (1) `rank/select` succinct structures ($\approx 0.07n$ bits) (2) loaded C++ libraries (a constant amounting to approximately 3.4 MB, i.e. $\approx 0.06n$ bits in this case), and memory fragmentation ($\approx 0.16n$ bits). Excluding the size of C++ libraries (which is constant), our bitvector's size never exceeds $1.23n$ bits (being $1.20n$ bits on average).

7.1.2 Running Times

Plots in Figures 7.4-7.9 show running times of our bitvectors on all queries. We used a linear regression model (inferred using `R`'s function `lm`) to fit function $c+k \cdot \log b$ with query times of our gap-encoded bitvector. Query times of our succinct bitvector were interpolated with a constant (being n fixed). These plots show interesting results. First of all, our succinct bitvector supports extremely fast ($0.01\mu s$ on average) `access` queries. `rank` and `select` queries are, on average, 15 times slower than `access` queries. As expected, `insert` queries are very slow, requiring—on average—390 times the time of `access` queries and 26 times that of `rank/select` queries. On all except `access` queries, running times of our gap-encoded bitvector are faster than (or comparable to) those of our succinct bitvector for $b/n \leq 0.1$. Combined with the results in Plots 7.1, 7.2, and 7.3, these considerations confirm that for $b/n \leq 0.1$ our gap-encoded bitvector should be preferred to the succinct one. `access`, `rank`, and `select` queries are all supported in comparable times on our gap-encoded bitvector ($\approx 0.05 \cdot \log b\ \mu s$), and are one order of magnitude faster than `insert` queries. Finally, one might wonder why `gap-bv` is much slower than `suc-bv` for $b/n \geq 0.1$, given that they use the same machinery (i.e. a SPSI). This is due to two main reasons. First of all, in `suc-bv`'s SPSI, all integers are either 0 or 1 so we use built-in constant-time operations such as `popcount` and masks to speed-up queries. This is not true in `gap-bv`: even for very dense bitvectors, a single gap length greater than 1 would force all integers in its leaf to be assigned more than 1 bit (thus preventing the use of fast built-in operations on words). The second reason for this behavior is that we use a larger leaf size (of $\mathcal{O}(\log^2 n)$) in `suc-bv`. This leads to a smaller tree height with respect to `gap-bv` (whose height is $\mathcal{O}(\log b)$) for large b/n.

[2] we estimated the impact of memory fragmentation by comparing RSS and allocated memory, after subtracting from RSS the estimated weight—approximately 3.4 MB—of loaded C++ libraries

7.2 Repetitive Text Collections

We organized experiments as follows. We generated and downloaded twelve highly repetitive text collections from three domains: genomes, software, and wikipedia web pages. Then, we ran six BWT construction algorithms, six LZ77 factorization algorithms, and built five compressed indexes on these datasets. Compression algorithms were assessed on their speed and memory footprint during execution, while indexes were assessed on their disk size and on the time and RAM space taken to answer `count` and `locate` queries. Running times and memory usage (Resident Set Size) were measured with the tool `/usr/bin/time`. We generated patterns in `pizza&chili` format [37]. On our repetitive datasets, a large number of patterns extracted with the `genpatterns` tool of [37] occurred millions of times in the text; to limit computation times while still being able to test the indexes on a sufficiently large number of patterns, we created our own version of `genpatterns` (available at [98]). Our tool employs an FM-index to count the number of pattern occurrences and uses this information to output only patterns occurring a fixed maximum number of times in the text. Using our tool, we extracted from each dataset 5000 patterns of length 2^i, for $i = 2, \ldots, 10$, occurring at most 1000 times each in the text.

7.2.1 Datasets

We used two custom scripts to generate the repetitive datasets. Our scripts download and concatenate all versions of a Wikipedia web page [103] and source code from all revisions of a GitHub repository [99]. In addition, we downloaded four repetitive DNA datasets from the `pizza&chili` repetitive corpus [37]. Our aim was to compare all tools (compression algorithms and indexes) on a common ground, so we chose the maximum input file size according to the most memory-consuming tool (the `LZ77-index` construction tool—see below—, which requires approximately $13n$ bytes of RAM during execution). As a result, we truncated all files to $5 \cdot 10^8$ Bytes (when bigger). The datasets are:

- `DNA` (from `pizza&chili` repetitive corpus):
  - `cere`: 37 sequences of *Saccharomyces Cerevisiae*
  - `para`: 36 sequences of *Saccharomyces Paradoxus*
  - `influenzae`: 78041 sequences of *Haemophilus Influenzae*
  - `escherichia`: 23 sequences of *Escherichia Coli*
- `Git repositories`. Concatenation of source files from the last revisions of:
  - `sdsl`. `github.com/simongog/sdsl-lite`
  - `samtools`. `github.com/samtools/samtools`
  - `boost`. `github.com/boostorg/boost`
  - `bwa`. `github.com/lh3/bwa`
- `wikipedia`. Concatenation of all versions of:
  - `einstein`. `en.wikipedia.org/wiki/Albert_Einstein`
  - `earth`. `en.wikipedia.org/wiki/Earth`

– `bush`. `en.wikipedia.org/wiki/George_W._Bush`

– `wikipedia`. `en.wikipedia.org/wiki/Wikipedia`

Table 7.1 reports the sizes of the above files before and after compression with 7-Zip (`www.7-zip.org`, indicated as `7z` in the following), followed by the compression rate (uncompressed size/compressed size).

File	Size (MB)	7z-compressed size (MB)	Compression rate
cere	439.92	8.01	54.90
para	409.38	9.80	41.78
influenzae	147.63	2.45	60.29
escherichia	107.47	7.06	15.23
sdsl	476.84	0.34	1385.94
samtools	476.84	0.70	677.01
boost	476.84	0.12	4031.41
bwa	418.38	0.38	1112.23
einstein	476.84	0.81	589.84
earth	476.84	0.97	489.99
bush	476.84	1.15	413.78
wikipedia	476.84	1.24	385.73

Table 7.1: Size of the datasets before and after `7z`-compression. Last column is the rate between columns 2 and 3, and represents by how many times `7z` compresses the dataset. Note that software repositories are extremely repetitive: in particular, the `boost` C++ library is compressed by over 4000 times with `7z`.

7.2.2 Tested Algorithms and Indexes

BWT Table 7.2 shows all tested BWT construction algorithms. We include the space used by the tools in RAM and on disk in theory and practice. Disk space for input/output is excluded.

LZ77 Table 7.3 shows all tested LZ77 factorization algorithms. These tools work into main memory, so we do not show disk usage. As far as `lzscan` is concerned, we chose d in such a way that the term $\mathcal{O}(n/d)$ was always around 50% text's size (the tool requires n/d to be an integer number of MB). The tool `bwte` uses a constant (user-defined) amount of RAM during execution; we fixed this constant to the default (256 MB) in all experiments. We included `isa6r` as it is specialized for repetitive inputs.

Indexes The tested indexes for repetitive text collections are reported in Table 7.4. `fmi-rrr` is `sdsl`'s FM-index implementation [43,98] using RRR-bitvectors for the wavelet trees. We used the same suffix array sparsification factor of 512 for the following indexes: `fmi-rrr`, `rlcsa`, `s-rlbwt`, `slz-rlbwt` (in the latter index, this value represents the sparsification degree d of the LZ77-d parse, see Section 5.3.3). We chose to build the most space-efficient version of the LZ77 index [64] (`lzi` in the table) by calling the index construction tool as `build_lz77 <input> <output> bsst brev`.

tool	type	RAM th	disk th	RAM pr	disk pr
`se-sais` [8, 43]	SA	$\mathcal{O}(n \log \sigma)$	$\mathcal{O}(n \log n)$	$1.2n$ B	$5n$ B
`divsufsort` [43, 81]	SA	$n \log n$	0	$5n$ B	0
`bwte` [29]	BWT	$\mathcal{O}(1)$	$\mathcal{O}(n \log \sigma)$	256 MB	$1.3n$ B
`dbwt` [107]	BWT	$\mathcal{O}(n \log \sigma)$	0	$2.4n$ B	0
`cw-bwt` 3.2, 6.4.1	BWT	$n(H_k + 1) + o(n \log \sigma)$	0	$n(H_k + 1)$ bits	0
`rle-bwt` 3.3, 6.4.2	BWT	$\mathcal{O}(r)$ words	0	$11r$ B	0

Table 7.2: Tested BWT construction tools. Some tools build only the suffix array (note that with SA + text we can access the BWT, so this is conceptually equivalent to building the BWT). *RAM/disk th/pr* stands for space used in RAM/disk in theory and in practice (some tools work in external memory). Theoretical spaces are in bits if not otherwise specified.

tool	RAM th	RAM pr
`isa6r` [61, 73]	$\mathcal{O}(n \log n)$	$6n$ B
`kkp1s` [58, 73]	$\mathcal{O}(n \log n)$	$5n$ B
`lzscan` [57, 73]	$\mathcal{O}(n \log \sigma)$	$n + \mathcal{O}(n/d)$ B
`h0-lz77` 4.2, 6.4.3	$n(H_0 + 1) + o(n \log \sigma)$	$1.1n(H_0 + 1)$ bits
`rle-lz77-1` 4.3.1, 6.4.4	$\mathcal{O}(r)$ words	$48r$ B
`rle-lz77-2` 4.3.2, 6.4.4	$\mathcal{O}(r + z)$ words	$16r + 16z$ B

Table 7.3: Tested LZ77 factorization tools. All tools work in RAM; we show both theoretical (RAM th) and practical (RAM pr) space bounds. Theoretical spaces are in bits if not otherwise specified.

index name	notes	ref
`fmi-rrr`	`sdsl`'s FM index with RRR-compressed wavelet trees	[43, 98]
`rlcsa`	Run-length compressed suffix array	[110, 111]
`lzi`	Most compressed variant of the LZ77 index	[64, 65]
`s-rlbwt`	Our sparse rlbwt index	5.3.2, [101]
`slz-rlbwt`	Our `slz-rlbwt` index	5.3.3, [100]

Table 7.4: Tested compressed indexes

7.2.3 Results

Compression Tools

We start by discussing the behavior of our BWT construction algorithm `cw-bwt`. Recall that we tested two `cw-bwt` implementations. The former, originally presented in paper `(i)`, discussed in Section 3.2, and implemented in [96], employs a dynamic bitvector (implemented in [42]) supporting constant-time operations on bit-sequences of size $w^{\mathcal{O}(1)}$. This implementation was originally developed to prove that our algorithm runs in linear time on constant-sized alphabets and near-uniform text distributions. The latter `cw-bwt` version has been described in Section 6.4.1 and is implemented in `DYNAMIC`. This alternative version runs in $\mathcal{O}(n \log \log n)$ time on constant-sized alphabets and near-uniform text distributions. We ran the linear-time version of `cw-bwt` on several prefixes of the Human genome (alphabet $\Sigma_{DNA} = \{A, C, G, T, N\}$, size $n \approx 3 \cdot 10^9$). Plot in Figure 7.10 shows the results. Vertical dashed lines show how the entropy order k automatically increases with the input sequence size. The plot confirms that—on constant-sized alphabets and

near-uniform text distributions—`cw-bwt` runs in linear time. `cw-bwt` completed the BWT of the entire Human genome in 4 hours and 37 minutes using only 994 MB of RAM (about 2.6 bits per input symbol, less than a plain encoding of the alphabet Σ_{DNA}). As discussed below, the $\mathcal{O}(n \log\log n)$-time version of `cw-bwt` has been compared with all other compression tools on repetitive datasets.

Plots in Figures 7.11-7.16 show running times and memory usage of tested BWT/LZ77 compression algorithms on all datasets. Solid and a dashed horizontal lines show the datasets' sizes before and after compression with `7z`, respectively. Our tools are highlighted in red. We can infer some general trends from the plots. Our tools (except `rle-lz77-1` in some cases) use always less space than the plain text, but from one to three orders of magnitude more space than the `7z`-compressed text. `h0-lz77` and `cw-bwt` use always a working space very close to (and always smaller than) the plain text, with `cw-bwt` (k-th order compression) being more space-efficient than `h0-lz77` (0-order compression). On the other hand, our more space-efficient tools—`rle-bwt` and `rle-lz77-2`—are up to two orders of magnitude more space-efficient than `h0-lz77` and `cw-bwt` in most of the cases. This behavior confirms—as expected—that entropy-compression is not able to exploit long text repetitions to improve compression; LZ77 and run-length encoding of the BWT are by far more appropriate in such cases. As predicted by theory (Section 6.4.4), `rle-lz77-1` is slower and less space-efficient than `rle-lz77-2` in all cases.

`bwte` represents a good trade-off in both running times and working space between tools working in compressed and uncompressed working space. `kkp1s` is the fastest tool, but uses a working space that is one order of magnitude larger than the uncompressed text and—in all except DNA datasets—three orders of magnitude larger than that of `rle-bwt` and `rle-lz77-2`. As predicted by theory, tools working in compact working space (`lzscan`, `se-sais`, `dbwt`) use always slightly more space than the uncompressed text, and one order of magnitude less space than tools working in $\mathcal{O}(n)$ words.

To conclude, all plots show that the price to pay for working in small space is high running times. Our tools are up to three orders of magnitude slower than tools working in $\mathcal{O}(n)$ words of space. As predicted by theory (Sections 3.2 and 4.2 and Theorems 38 and 40), `cw-bwt` is faster than `h0-lz77`. This is due to two main reasons: (i) `cw-bwt` breaks the BWT in contexts, and therefore works on much smaller dynamic strings, and (ii) the average wavelet tree height in `cw-bwt` is H_k, while in `h0-lz77` is H_0. In some cases (e.g. `boost`), these factors make `cw-bwt` almost one order of magnitude faster than `h0-lz77`.

Indexes

Disk space In Figures 7.17 and 7.18 we compare the disk memory footprint of our `s-rlbwt` index with that of the `rlcsa` index. We show space usage in terms of bits per character. We moreover separate GitHub and Wikipedia files from DNA files, being the firsts one order of magnitude more compressible than the seconds. These plots show only the space taken by the compressed suffix arrays; we excluded the suffix array sampling from the measurements. On DNA datasets, the `s-rlbwt` index is much smaller than the `rlcsa`. This trend is still present—even if with smaller differences—On GitHub datasets. On Wikipedia datasets the trend is inverted: the `rlcsa` is slightly more space efficient than the `s-rlbwt` index. This behavior might reflect a higher dependency on the alphabet size in our index, rather than on the dataset compressibility. To test this statement, we generated a uniform random text of length $n = 10^8$ on the alphabet $\{1, \ldots, 255\}$ and built

the `rlcsa` and the `s-rlbwt` indexes on this file. The `rlcsa`'s size was of 203511448 Bytes, while the `s-rlbwt`'s size 197956842 Bytes (i.e. 97% `rlcsa`'s size). This indicates that, while the alphabet size plays an important role on the size of our index (being the two indexes almost equal in size), it is not the only factor that determines its compression efficiency (the text's structure also has some impact). In Figure 7.19 we compare the disk space of all indexes, taking into account all their components (i.e. suffix array sampling included, when present). As expected, `slz-rlbwt` and `lzi` are by far the most space-efficient indexes. `slz-rlbwt` is—on average—1.1 times larger than `lzi`, with a peak on the `boost` dataset (the most compressible one, see Table 7.1), where `slz-rlbwt` is 1.38 times larger than `lzi`. On one dataset (`escherichia`), `slz-rlbwt` is more space-efficient than `lzi`. As Table 7.5 shows, the space-efficiency of the `slz-rlbwt` index derives from the fact that—on average—z_{512} is just 7.3% of z_0. This explains why `slz-rlbwt` and `lzi` are comparable in size, despite `slz-rlbwt` including almost all components of the `lzi` (plus a RLBWT). The `rlcsa` is—on average—1.16 times larger than `s-rlbwt`. The two indexes are comparable in size on all except DNA datasets; on these datasets, `rlcsa` takes as much as 1.7 times the space of `s-rlbwt`. Finally, as expected `fmi` is by far the least space-efficient tool: this confirms, again, that entropy compression is not suitable for compressing repetitive datasets.

File	z_{512}	z_0	z_{512}/z_0 (%)
boost	795	22680	3.5
bush	21643	358035	6.0
bwa	6643	106655	6.2
cere	208695	1700630	12.3
earth	15882	314681	5.0
einstein	16291	251450	6.5
escherichia	113700	2075822	5.5
influenzae	146072	765934	19.1
para	215915	2332657	9.3
samtools	6991	150988	4.6
sdsl	4411	113591	3.9
wikipedia	23761	390170	6.1

Table 7.5: The table shows the numbers z_{512} and z_0 of phrases of the LZ77-512 and LZ77-0 factorizations, respectively. Remarkably, z_{512} is—on average—just 7.3% of z_0. z_{512} and z_0 were computed using the tools `slz-rlbwt` and `kkp1s`, respectively

Count queries Figures 7.20-7.25 show running times of all indexes on `count` queries. Recall that `lzi` does not support `count` directly: in order to count the number of pattern occurrences, this index has to locate them all. As expected, `lzi` is three orders of magnitude slower than all other indexes on all datasets and all pattern lengths. The other indexes run in comparable times, with `rlcsa` being the fastest and our `s-rlbwt` and `slz-rlbwt` indexes the slowest (up to 5 times slower than `rlcsa` on the `boost` dataset). Also this is an expected result, since our indexes are asymptotically slower than `rlcsa` (by a factor of $\log^{\epsilon} n$, see Section 5.3.2). `fmi` is—in general—faster than our indexes and

slower than `rlcsa`.

Locate queries Figures 7.26-7.31 show running times of all indexes on `locate` queries. Again, `lzi` running times show a sharp dependency (quadratic) on the pattern length. The other tools do not show this behavior; however, on short patterns `lzi` running times are—almost always—smaller than those of other indexes. Running times of all tools are affected by the number of pattern occurrences, which tend to decrease with the length of the pattern. This phenomenon is responsible for the running times of all indexes (except `lzi` and `slz-rlbwt`) to decrease with the pattern length. Interestingly, running times of the `slz-rlbwt` index show a sinusoidal behavior; this is due to the contrasting effects of decreasing number *occ* of occurrences and increasing pattern length m (recall that this index has a $occ \cdot m$ term in its `locate` running times). The plots show that our `slz-rlbwt` index is much faster than `lzi` on DNA datasets with patterns longer than, approximately, 2^6. This trend is inverted on GitHub and Wikipedia datasets (characterized by a larger alphabet), where `lzi` is up to two orders of magnitude faster than `slz-rlbwt` for short patterns. However, on long patterns the running times of the two indexes meet. To conclude, `rlcsa` is by far the fastest index, being two orders of magnitude faster than all other indexes on almost all datasets and pattern lengths. The only exception in this case is represented by `lzi`, which in some cases (DNA and Wikipedia datasets) is faster than `rlcsa` on short patterns (shorter than, approximately 2^3).

Considering that—on `locate` queries—`rlcsa` is faster and more memory-consuming than `slz-rlbwt`, we also compared these two indexes choosing the sample rate for `rlcsa` in such a way that its final size matched that of `slz-rlbwt`. Note that—on most datasets—`rlcsa` without suffix array sampling is already larger than the `slz-rlbwt` (see Figure 7.17); for this reason, we conducted the experiment on the `wikipedia` dataset, being it one of the few that was more compressible with `rlcsa` than with our RLBWT. With a sample rate of 2200 for `rlcsa` and 512 for `slz-rlbwt`, both indexes' sizes were approximately 4 MB. In Figure 7.44 we report our results. `rlcsa` is still the fastest index, supporting `locate` queries from 2^5 to 2^{10} times faster than `slz-rlbwt`. This is due to two reasons: (i) `slz-rlbwt` is (also in theory) slower than `rlcsa` by a factor of $\log^{\epsilon} n$, and (ii) the leading terms in `slz-rlbwt`'s and `rlcsa`'s `locate` times are $occ \cdot m$ and *occ*, respectively. Considering that—on our dataset—*occ* decreases with the pattern length, reason (ii) also explains why `rlcsa`'s `locate` times decrease with the pattern length while `slz-rlbwt`'s ones increase: this phenomenon is due to the fact that m grows at a faster rate than occ^{-1} does (so the product $occ \cdot m$ increases with m). We note that this is not necessarily a negative result for our index. `rlcsa` still suffers from the space/time trade-off introduced by the suffix array sampling; this means that, on very (in particular, exponentially) compressible datasets, `rlcsa` can achieve a memory footprint comparable to that of `slz-rlbwt` only storing very few (possibly, a constant number of) samples. This clearly affects `rlcsa`'s running times, which can become linear in the dataset size n. This situation already happens with the `boost` dataset: on this input, `rlcsa`'s compressed suffix array (without the suffix array sampling) takes 369 KB of memory. On the other hand, the *whole* `slz-rlbwt` index's size on this dataset is only 264 KB. It follows that the only hope for `rlcsa` to approach the same space usage of `slz-rlbwt` is to store a very small number of samples (possibly constant), thus incurring in a $\Omega(n)$ cost for each `locate` query.

RSS memory - count Figures 7.32-7.37 show RSS memory usage of all indexes on `count` queries. Memory usage of all indexes (as measured by `/usr/bin/time`) during `count` queries has the same trend among all datasets. `lzi` is the most space-efficient index, followed by `slz-rlbwt`, `s-rlbwt`, `rlcsa`, and `fmi`. The memory usage of `lzi` is always very close to that of `slz-rlbwt`, and `s-rlbwt` is always more space-efficient than `rlcsa`. Consistently with Figure 7.19 (disk space), `fmi` uses up to one order of magnitude more space than all other indexes.

RSS memory - locate Figures 7.38-7.43 show RSS memory usage of all indexes on `locate` queries. These plots show clearly that the `lzi` index is very sensitive to the number *occ* of pattern occurrences. On short patterns (having more occurreces), `lzi` is one of the least space-efficient tools. This effect is alleviated on long patterns, but—except on DNA datasets—`lzi` uses more working space than all other indexes except `fmi` (which is always the least space-efficient tool). As noted above, `rlcsa`'s working space is always higher than that of `s-rlbwt`. With the only exceptions of datasets `cere` and `para`, our `slz-rlbwt` index is by far the most space-efficient. These plots show that our DFS-traversal of the tree of pattern occurrences (see Section 5.3.3) is much more convenient than what implemented in the `lzi` index: in some cases (e.g. short `samtools` patterns), `lzi` uses more than 4 times the space of `slz-rlbwt`.

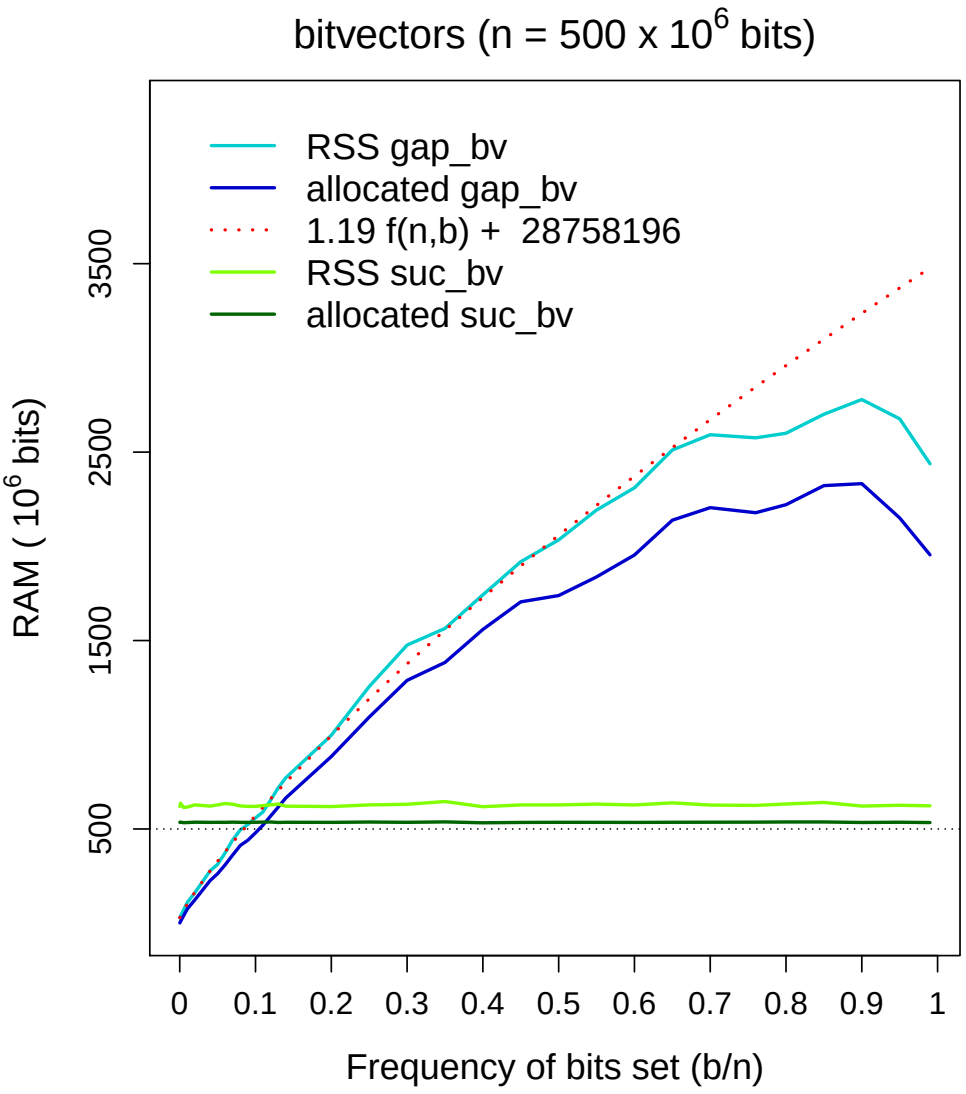


Figure 7.1: Memory occupancy of `DYNAMIC`'s bitvectors. n is the number of inserted bits and b the number of bits set. The frequency of bits set (x axis) is b/n. We show memory usage as measured internally (*allocated*: total number of bits allocated by our procedures) and externally by `/usr/bin/time` (*RSS*: resident set size). We moreover show the growth of function $f(n,b) = b(\log(n/b) + \log\log b + \log n/\log b)$ opportunely scaled to take into account memory fragmentation and the weight of loaded C++ libraries.

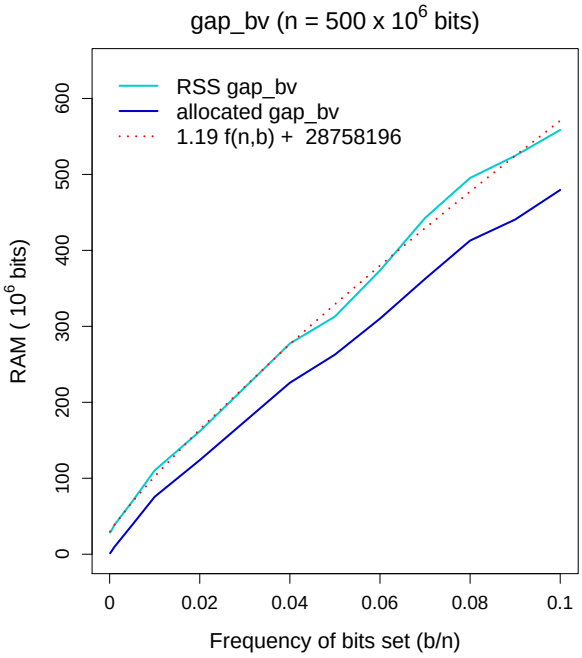


Figure 7.2: Memory occupancy of DYNAMIC's gap-encoded bitvector in the interval $b/n \in [0, 0.1]$. Here, $f(n, b) = b(\log(n/b) + \log\log b + \log n/\log b)$.

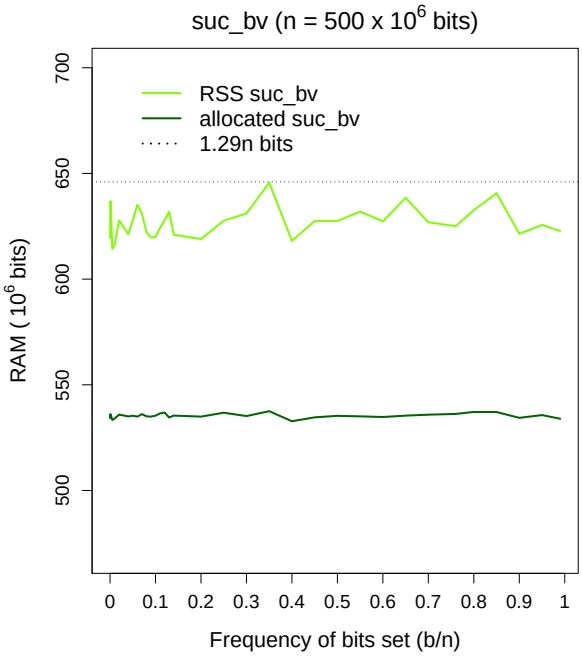


Figure 7.3: Memory occupancy of DYNAMIC's succinct bitvector

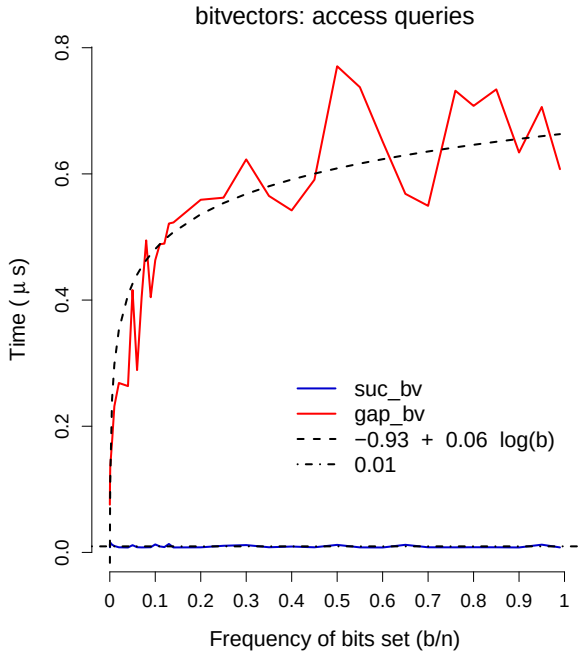


Figure 7.4: Running times of `DYNAMIC`'s bitvectors on `access` queries

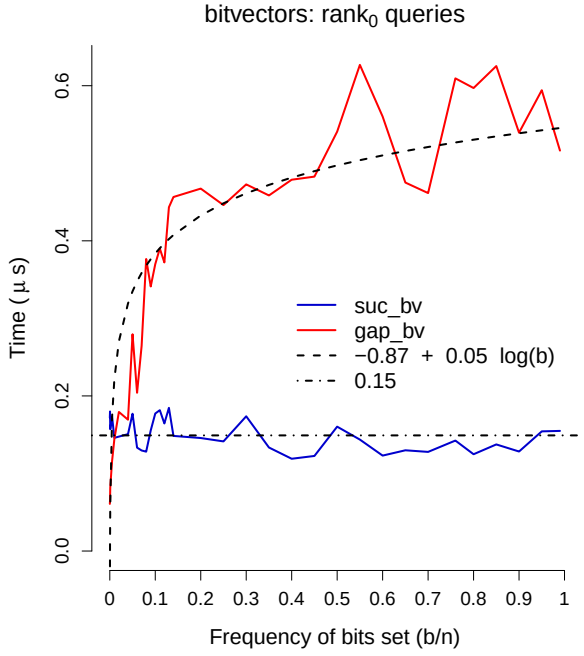


Figure 7.5: Running times of `DYNAMIC`'s bitvectors on $\texttt{rank}_0$ queries

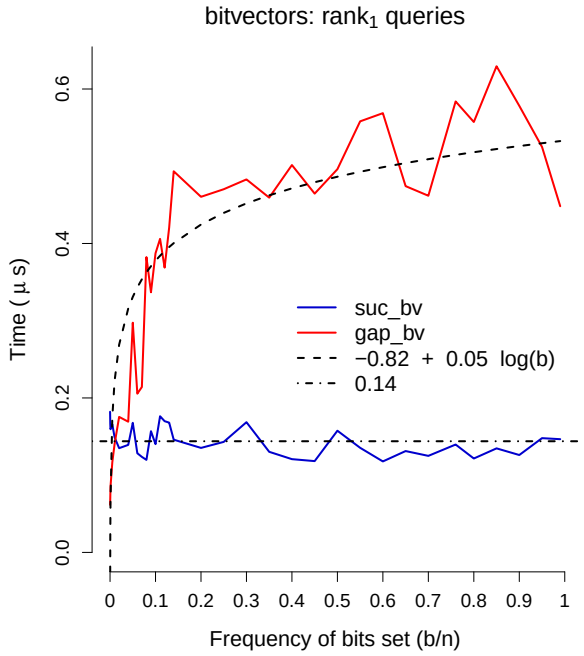


Figure 7.6: Running times of DYNAMIC's bitvectors on $\texttt{rank}_1$ queries

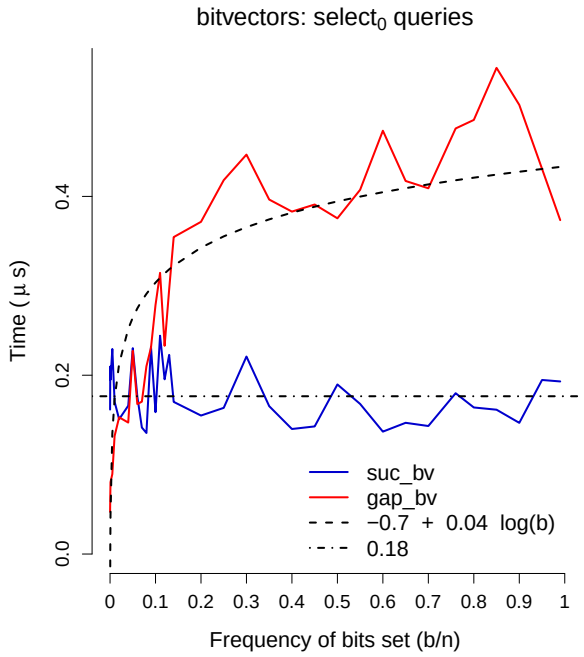


Figure 7.7: Running times of DYNAMIC's bitvectors on $\texttt{select}_0$ queries

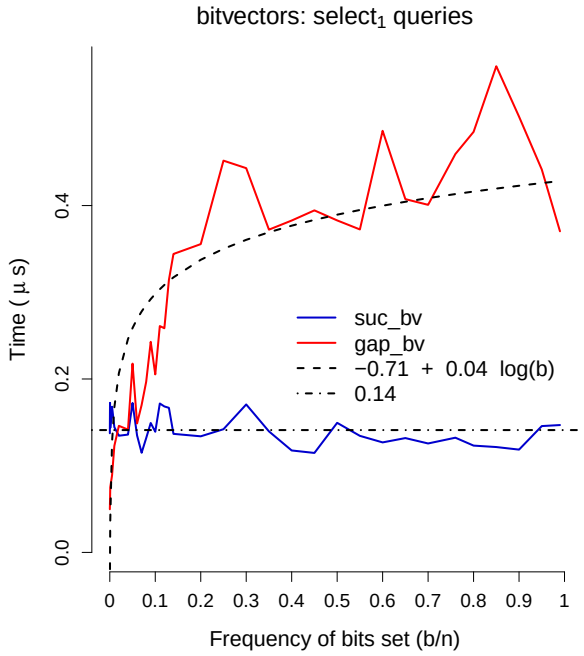


Figure 7.8: Running times of `DYNAMIC`'s bitvectors on `select`$_1$ queries

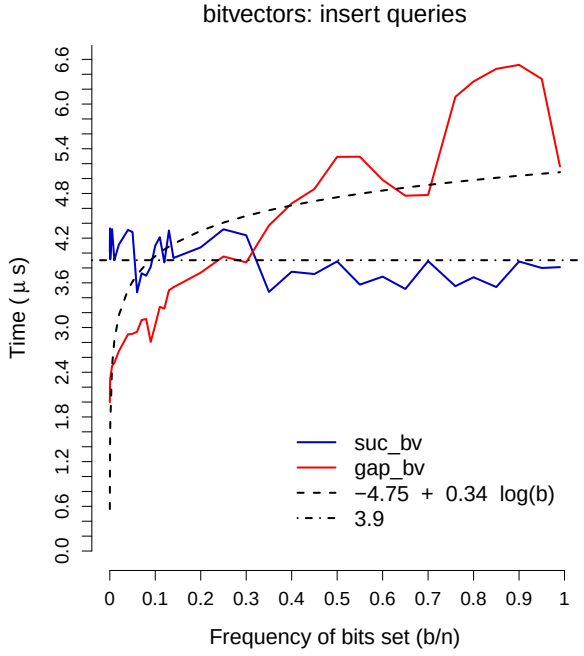


Figure 7.9: Running times of `DYNAMIC`'s bitvectors on `insert` queries

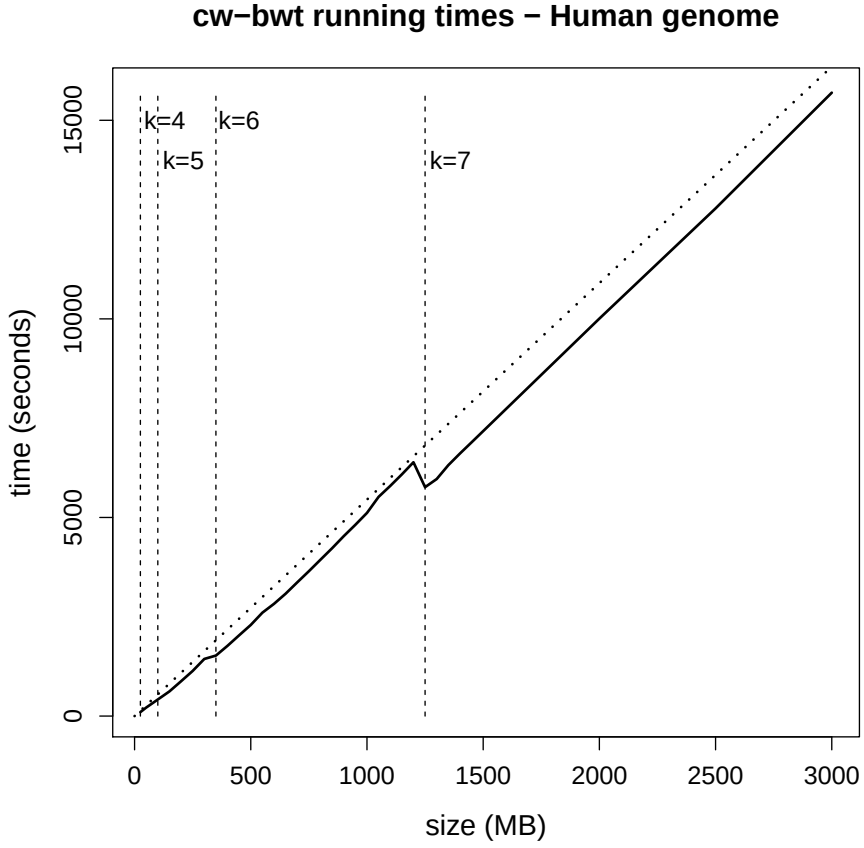


Figure 7.10: `cw-bwt` running times on prefixes of the Human genome. To generate this plot we ran an implementation [96] of `cw-bwt` using the bitvector [42] described in Section 3.2 (i.e. supporting constant-time operations on bit-sequences of size $w^{\mathcal{O}(1)}$) on several prefixes of the Human genome (alphabet $\Sigma_{DNA} = \{A, C, G, T\}$). Vertical dashed lines show how the entropy order k automatically increases with the input sequence size. The plot confirms that—on constant-sized alphabets and near-uniform text distributions—`cw-bwt` runs in linear time with respect to n.

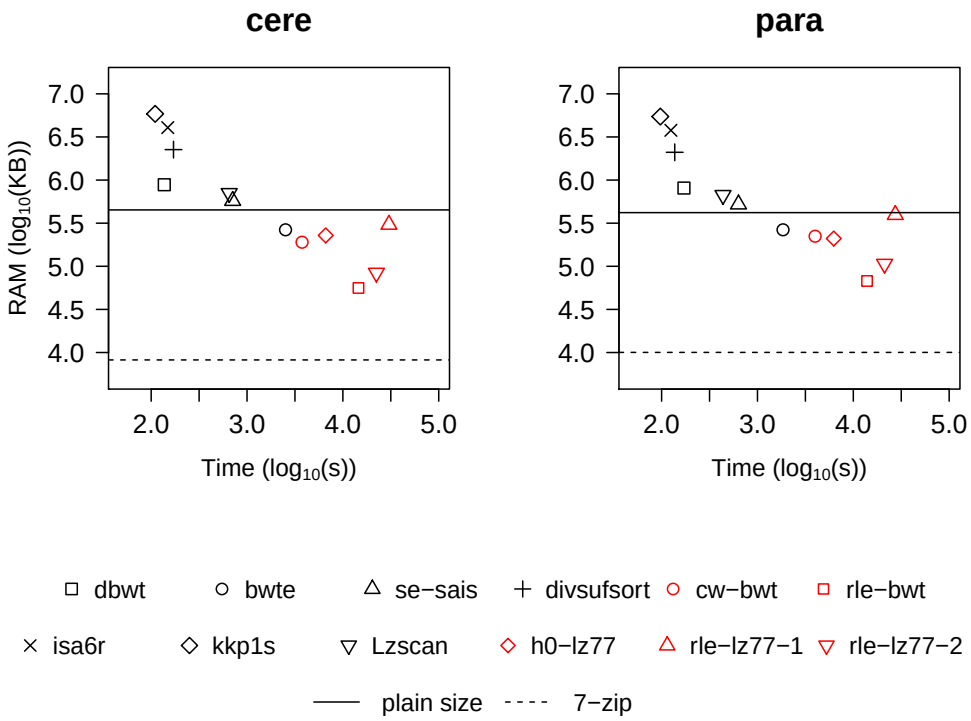


Figure 7.11: Compression tools on the datasets `cere` and `para`

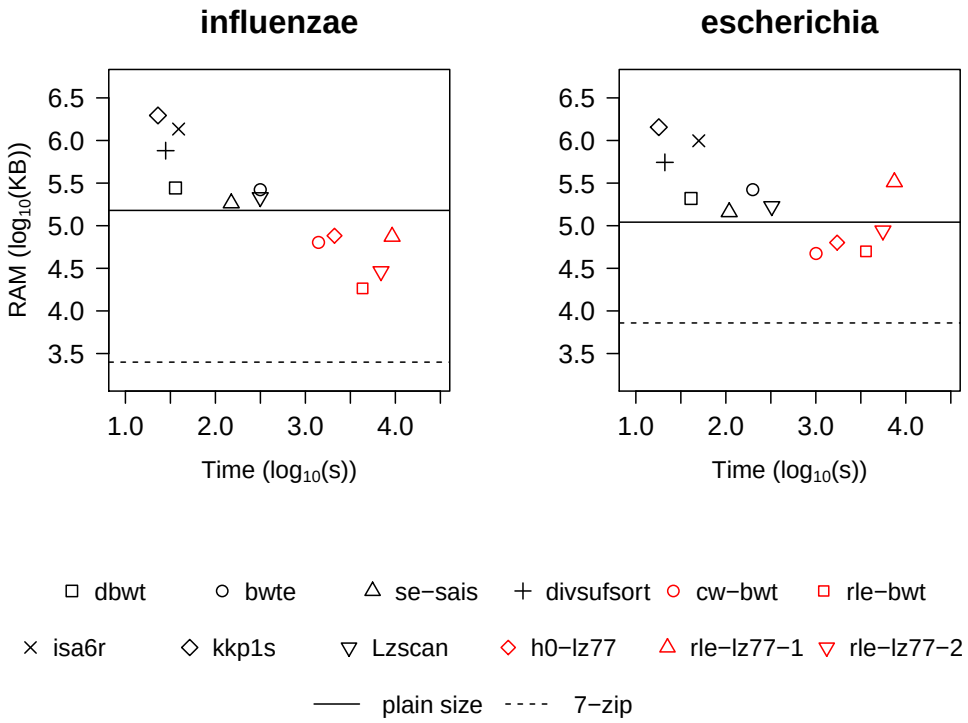


Figure 7.12: Compression tools on the datasets `influenzae` and `escherichia`

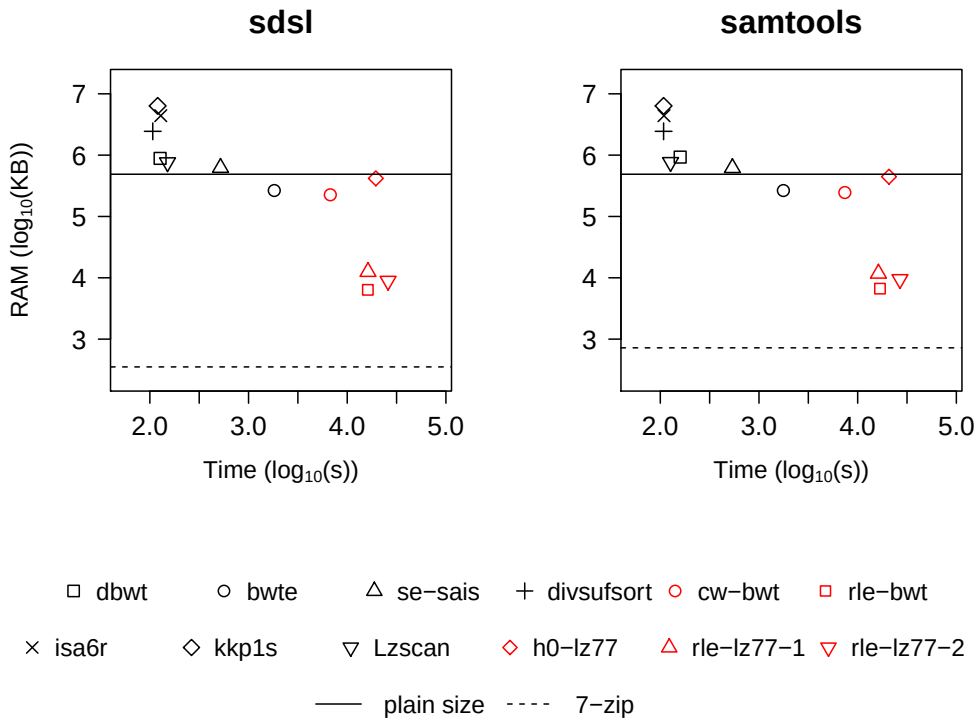


Figure 7.13: Compression tools on the datasets `sdsl` and `samtools`

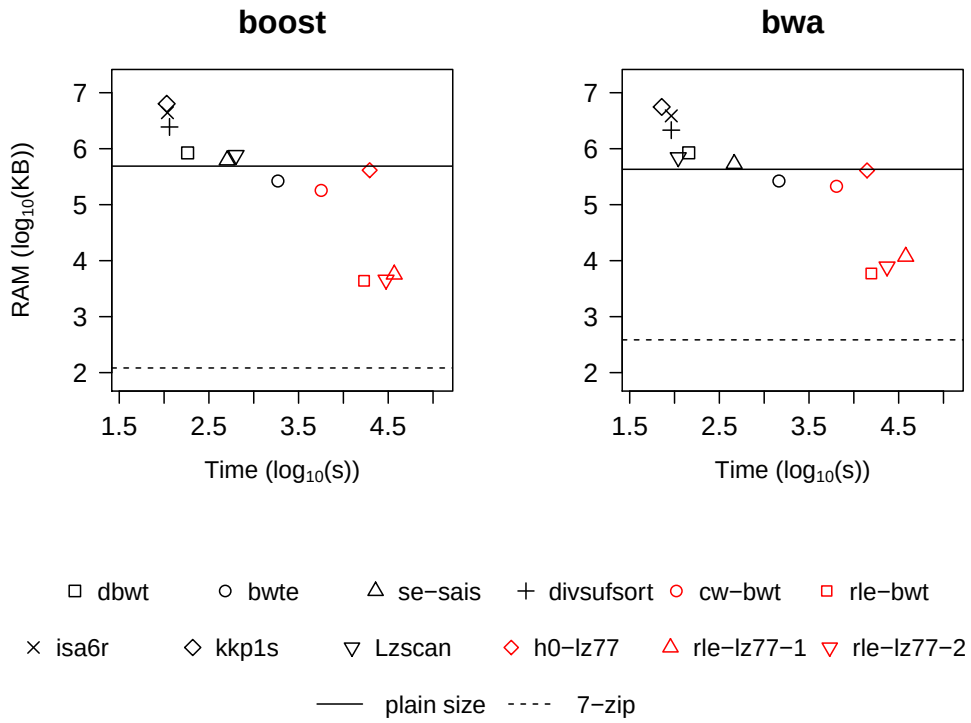


Figure 7.14: Compression tools on the datasets `boost` and `bwa`

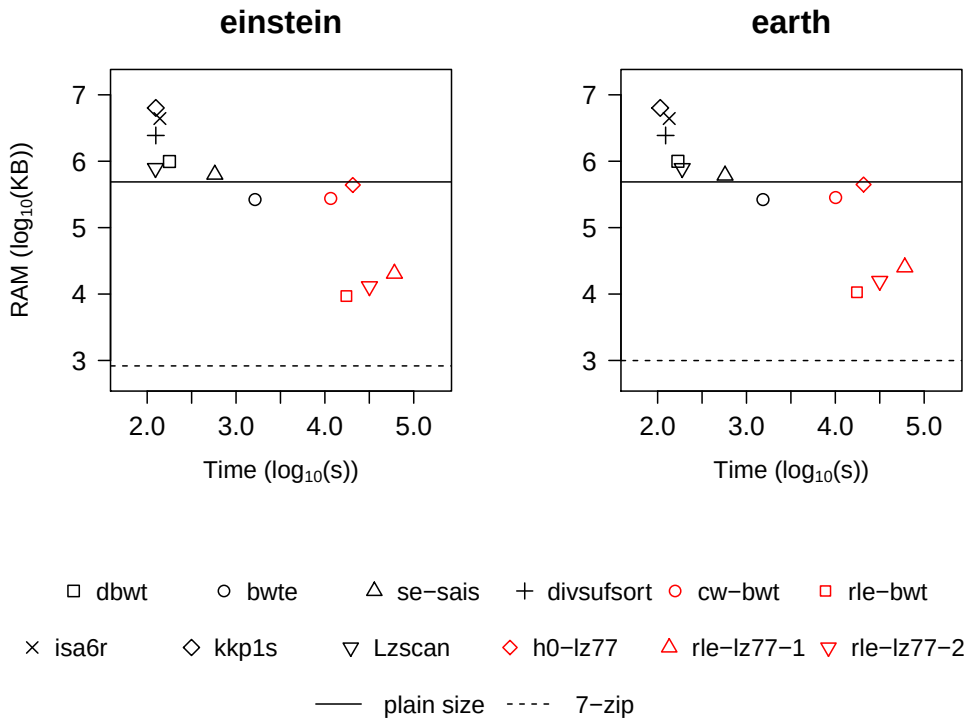


Figure 7.15: Compression tools on the datasets `einstein` and `earth`

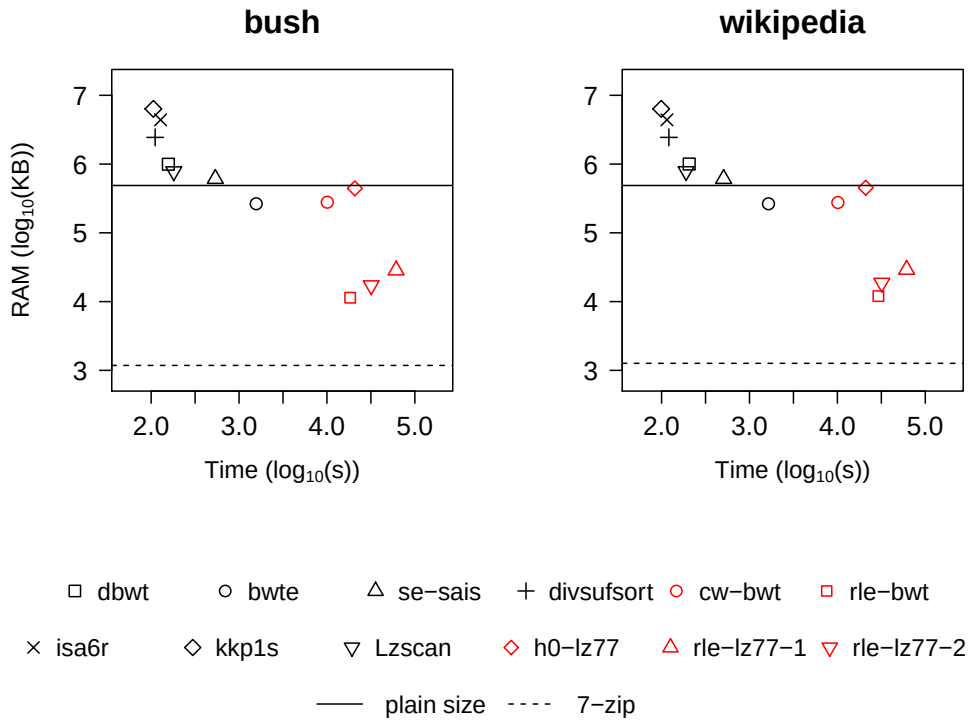


Figure 7.16: Compression tools on the datasets `bush` and `wikipedia`

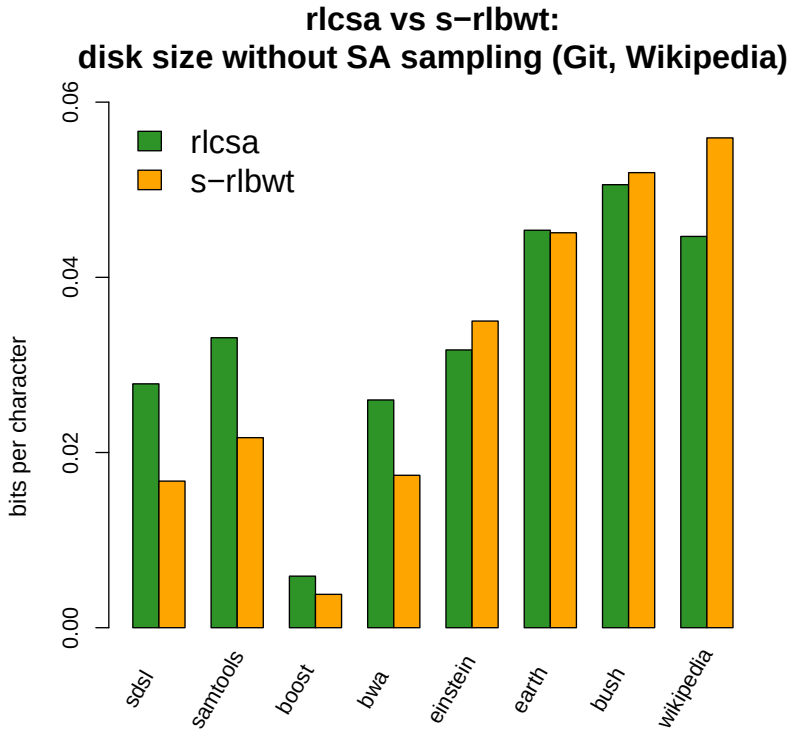


Figure 7.17: Disk size of the Run-length compressed suffix array (`rlcsa`) and of the sparse run-length BWT (`s-rlbwt`) excluding the suffix array sampling on GitHub and Wikipedia datasets. Our compressed suffix array improves upon the space of the `rlcsa` on GitHub datasets. On all Wikipedia datasets except `earth`, the trend is inverted.

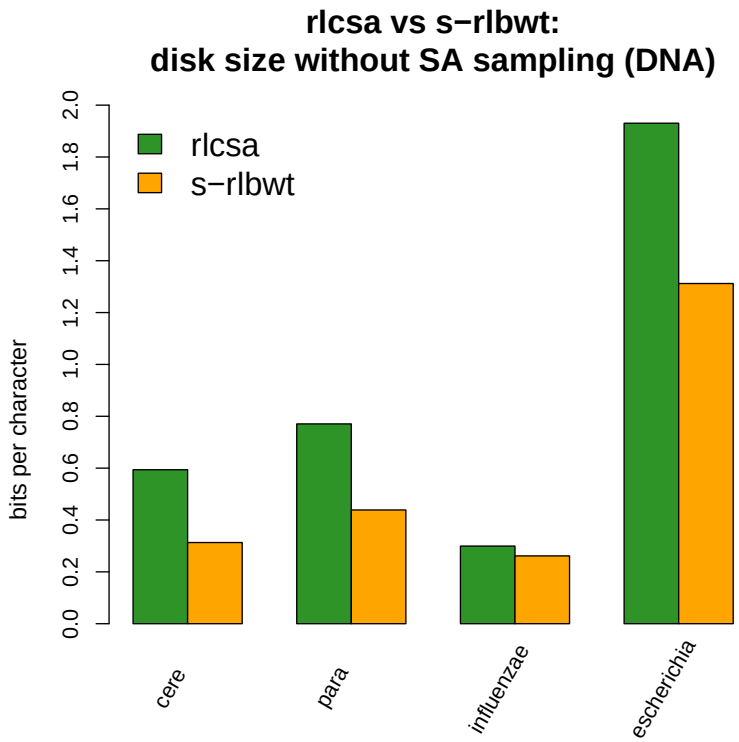


Figure 7.18: Disk size of the Run-length compressed suffix array (`rlcsa`) and of the sparse run-length BWT (`s-rlbwt`) excluding the suffix array sampling on DNA datasets. In this case, our compressed suffix array significantly improves upon the space of the `rlcsa`. Note that `rlcsa` is not able to compress `escherichia` (as it uses almost 2 bits per character).

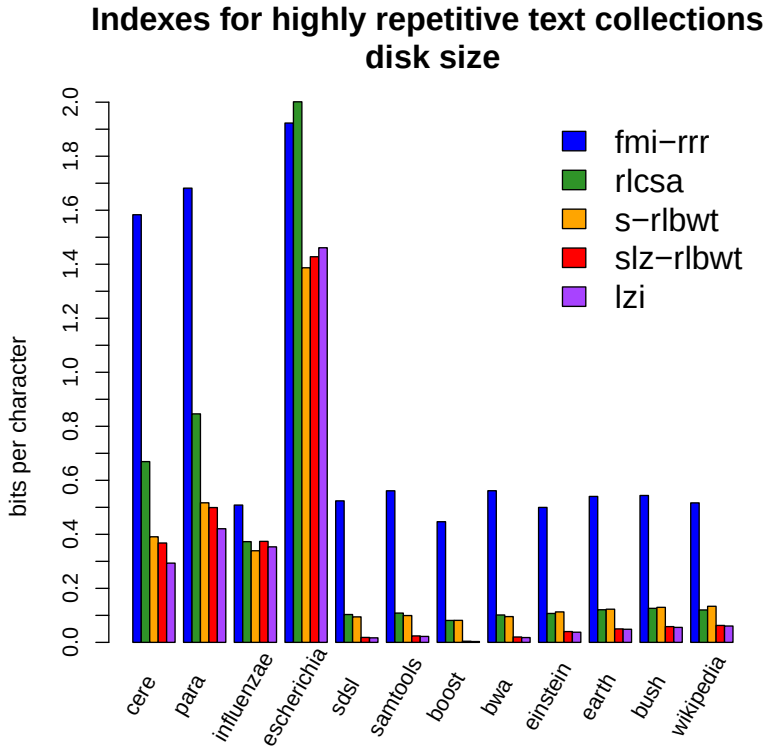


Figure 7.19: Disk space of the tested indexes. Note that indexes relying on a uniform sampling of the suffix array (`fmi-rrr`, `rlcsa`, `s-rlbwt`) are the most space-consuming. The `lzi` and `slz-rlbwt` indexes, on the other hand, compress all their structures in a data-aware manner and improve by several times the space of the other indexes on all datasets. It is worth to notice that the size of our `slz-rlbwt` is on average only 1.1 times larger than (never exceeding 1.38 times) that of `lzi`.

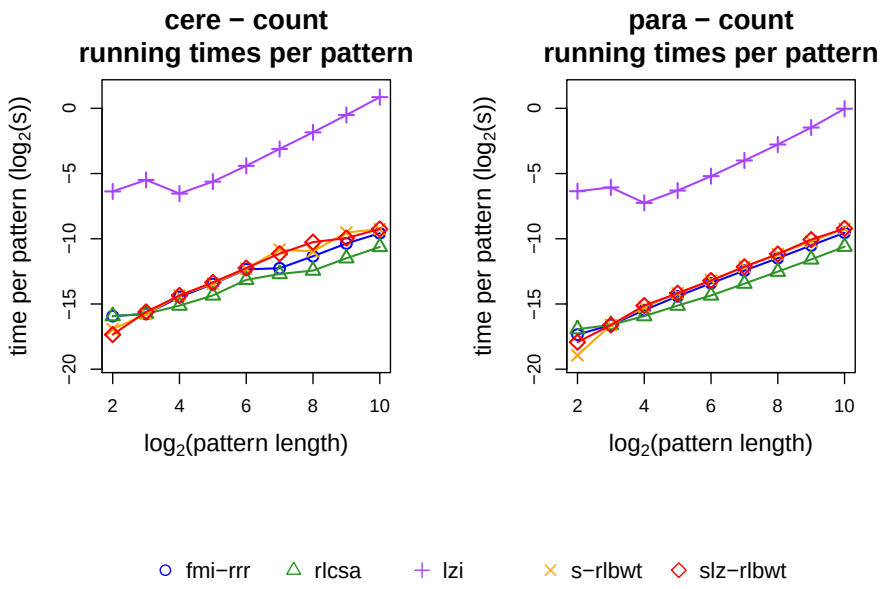


Figure 7.20: Count times of the indexes on `cere` and `para`

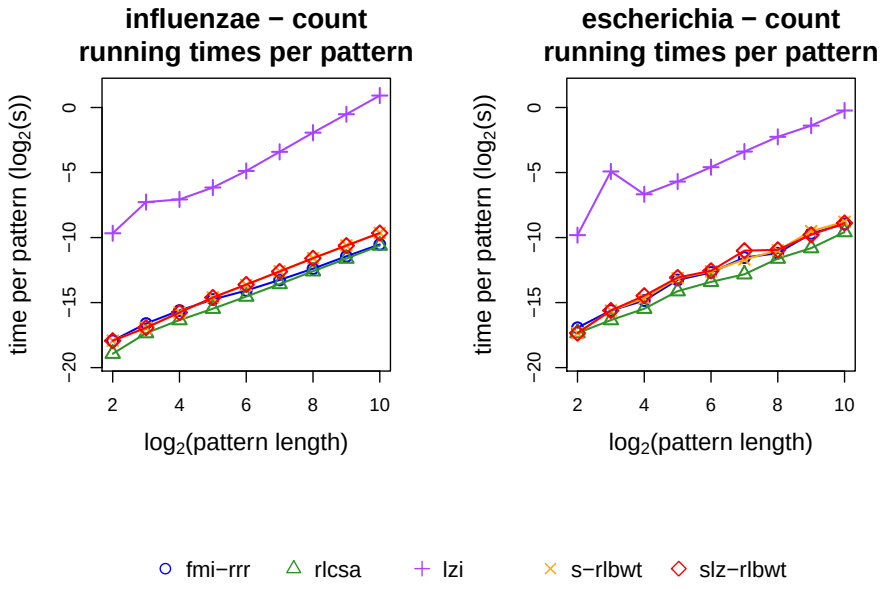


Figure 7.21: Count times of the indexes on `influenzae` and `escherichia`

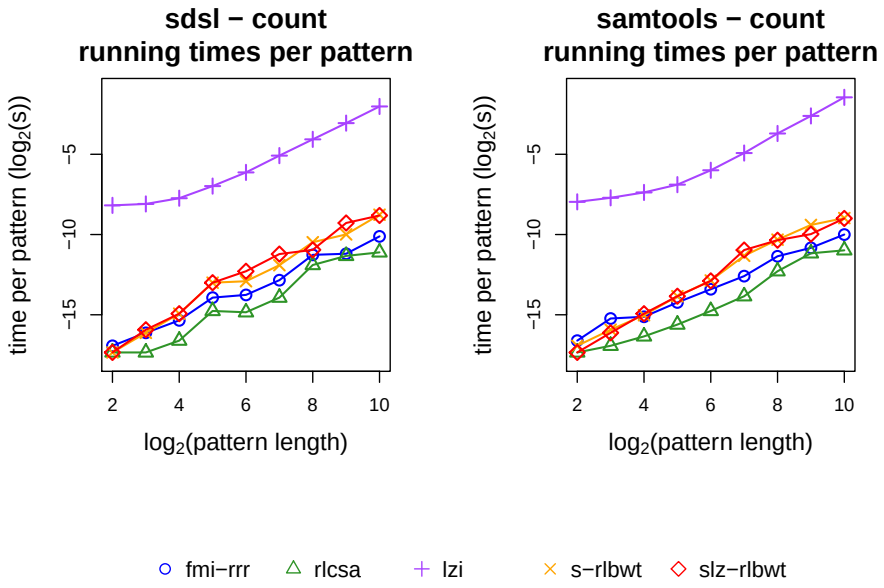


Figure 7.22: Count times of the indexes on `sdsl` and `samtools`

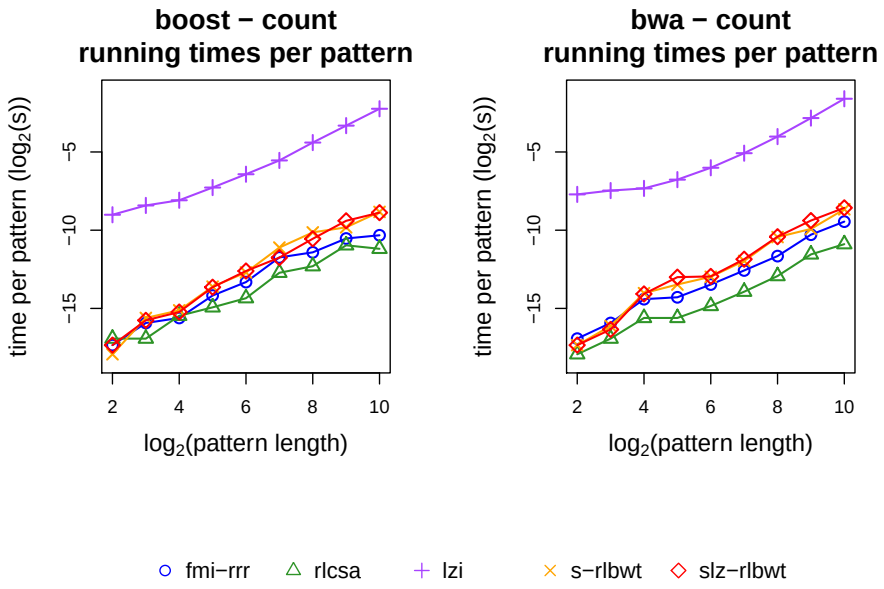


Figure 7.23: Count times of the indexes on `boost` and `bwa`

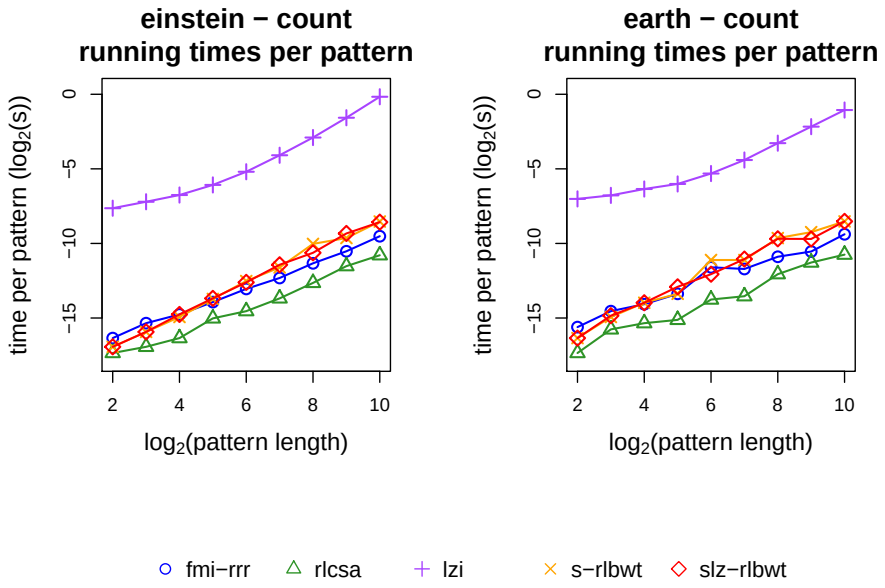


Figure 7.24: Count times of the indexes on `einstein` and `earth`

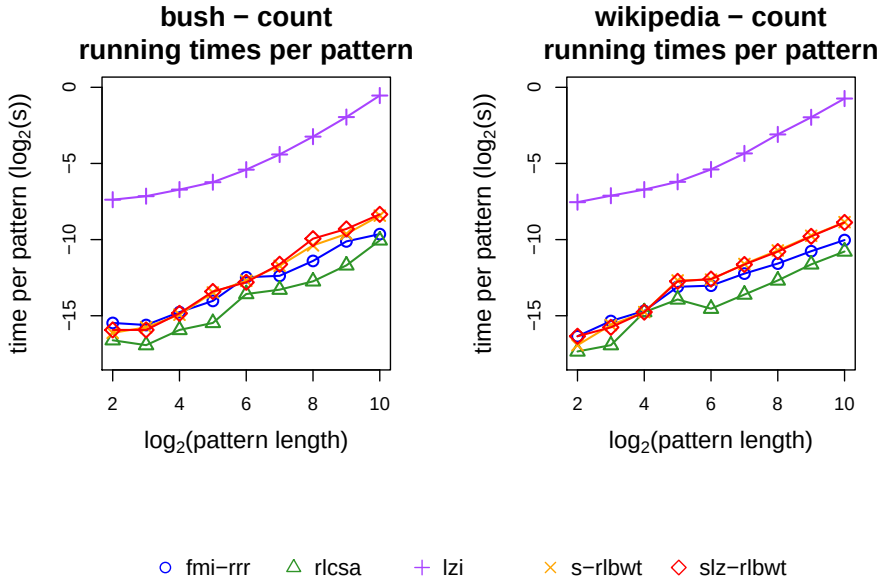


Figure 7.25: Count times of the indexes on `bush` and `wikipedia`

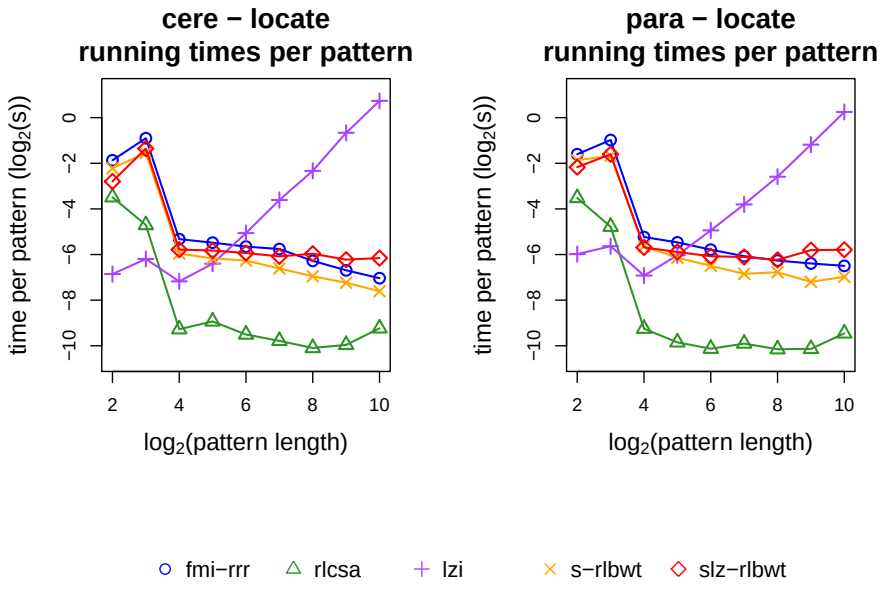


Figure 7.26: Locate times of the indexes on `cere` and `para`

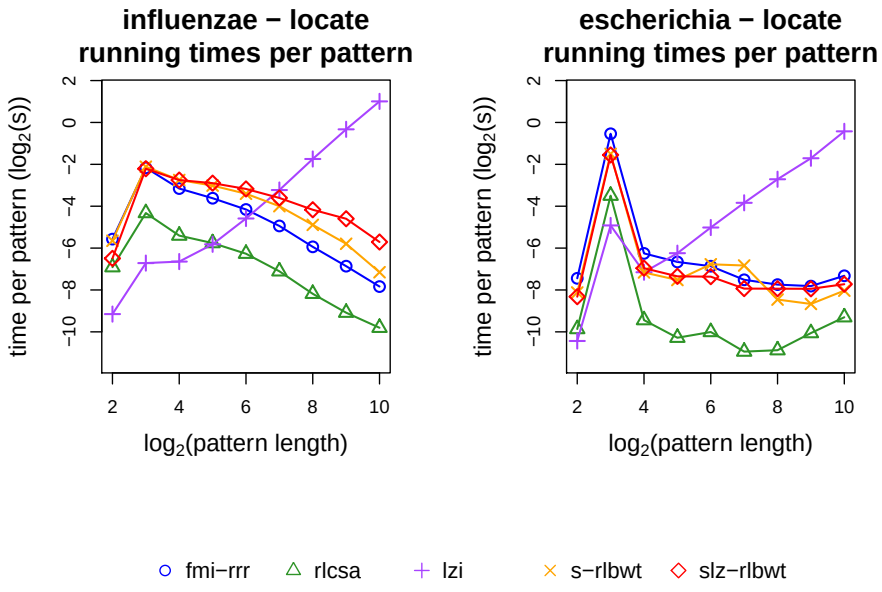


Figure 7.27: Locate times of the indexes on `influenzae` and `escherichia`

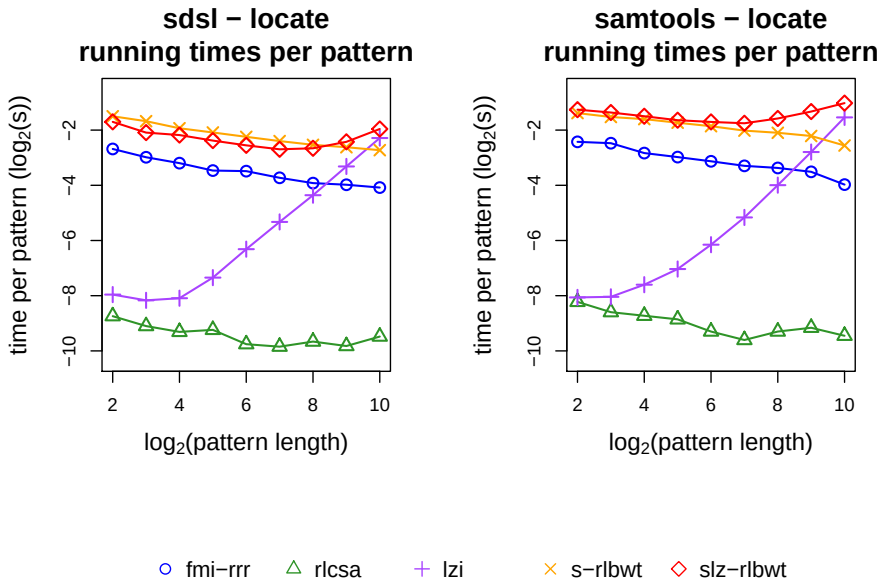


Figure 7.28: Locate times of the indexes on `sdsl` and `samtools`

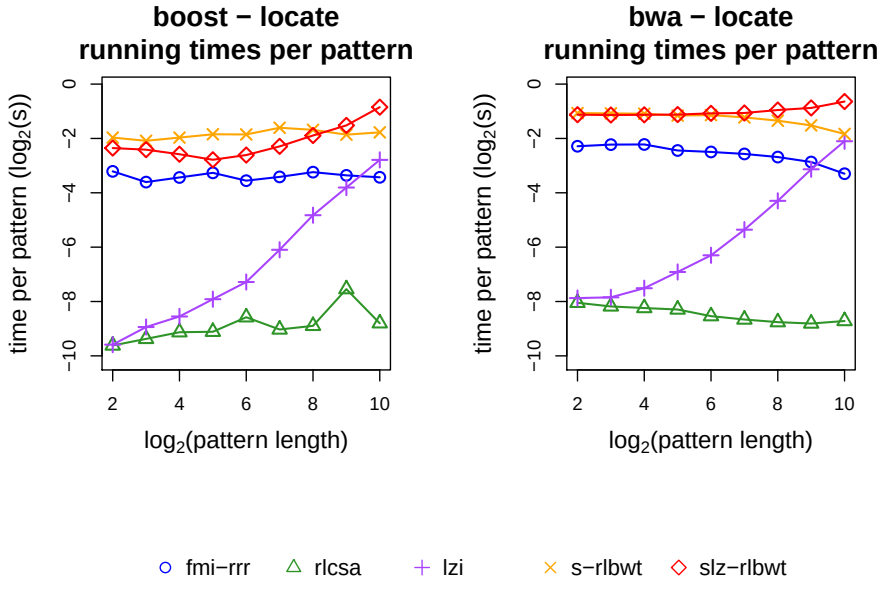


Figure 7.29: Locate times of the indexes on `boost` and `bwa`

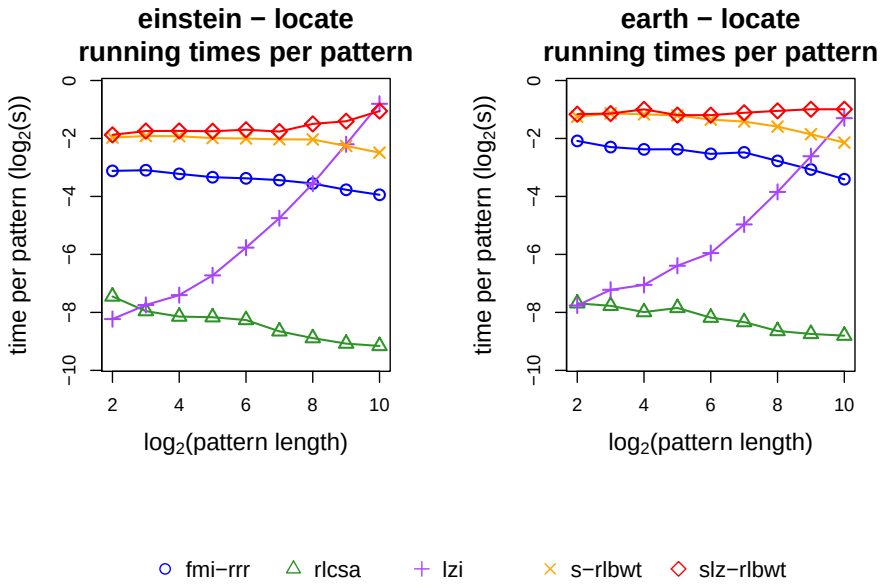


Figure 7.30: Locate times of the indexes on `einstein` and `earth`

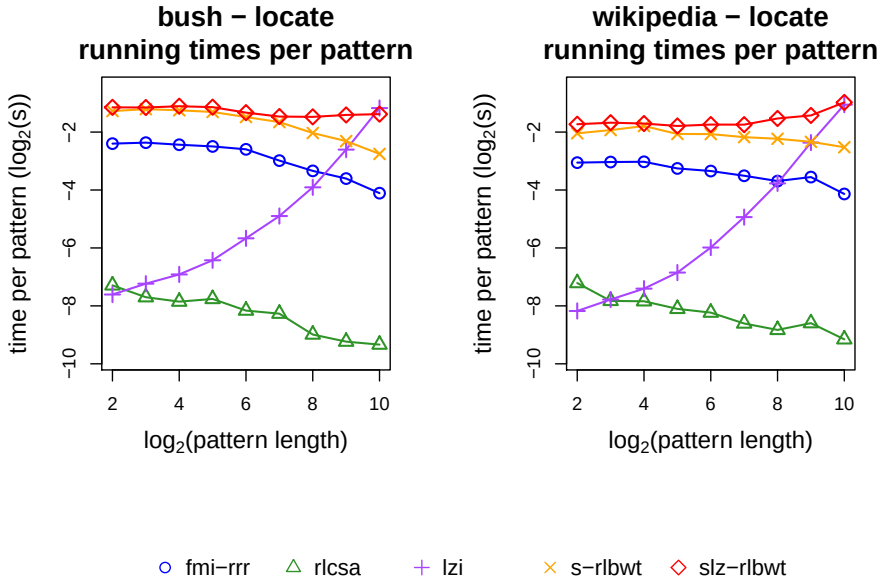


Figure 7.31: Locate times of the indexes on `bush` and `wikipedia`

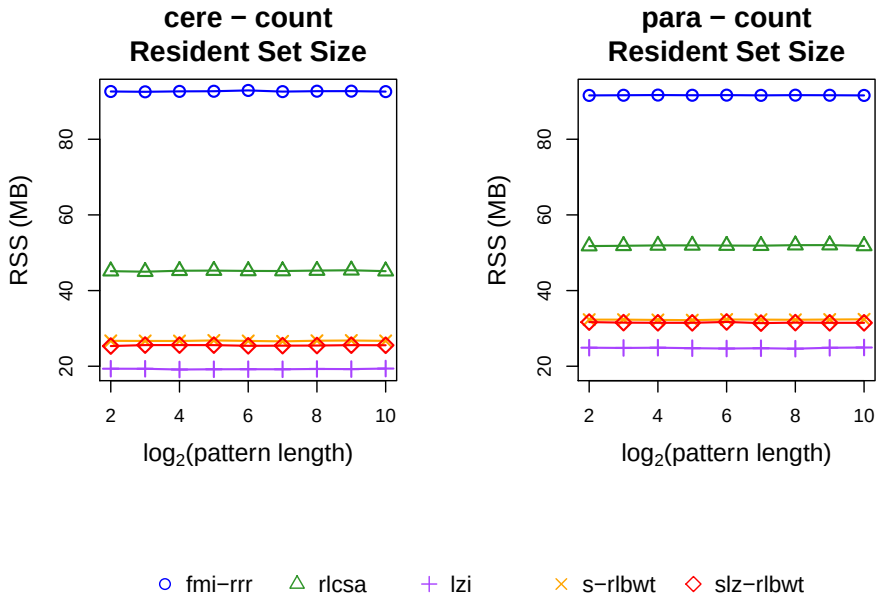


Figure 7.32: Count - Resident Set Size of the indexes on `cere` and `para`

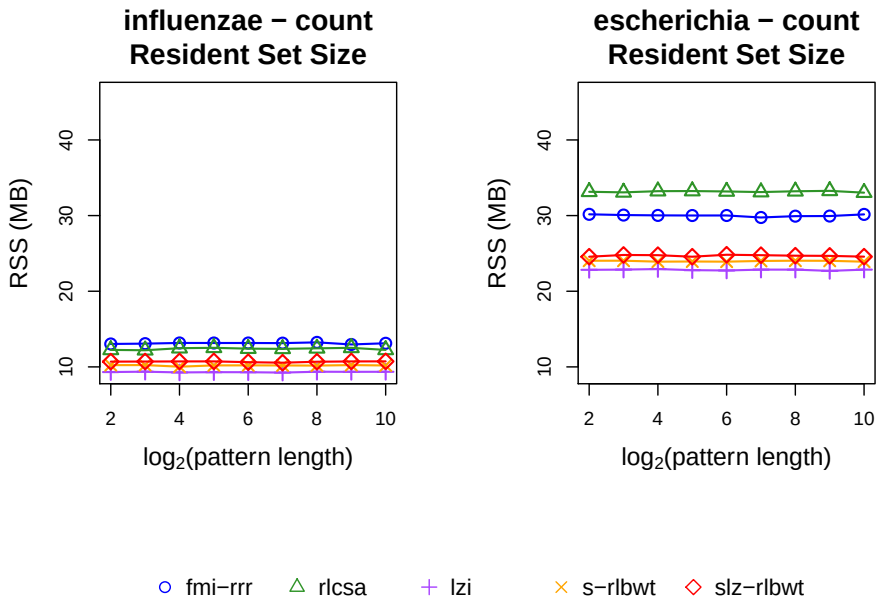


Figure 7.33: Count - Resident Set Size of the indexes on `influenzae` and `escherichia`

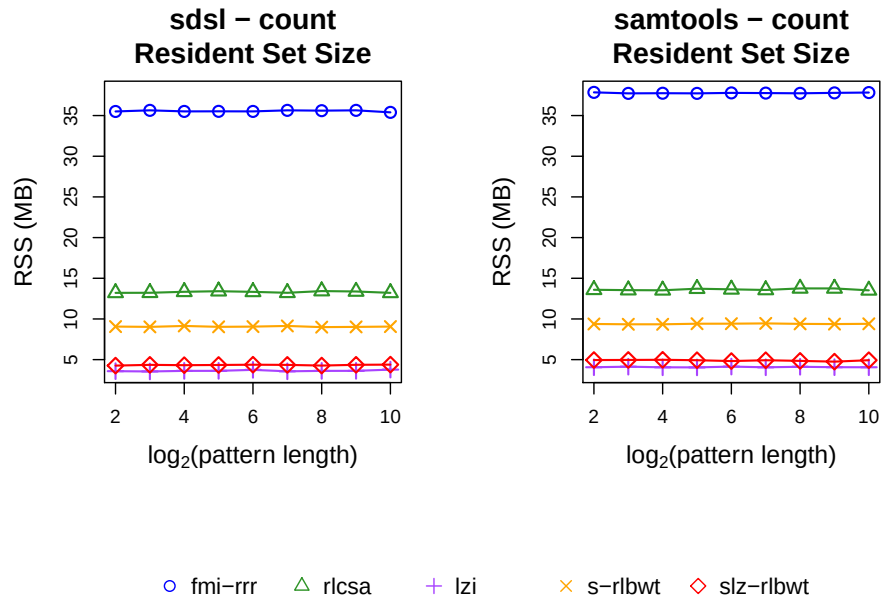


Figure 7.34: Count - Resident Set Size of the indexes on `sdsl` and `samtools`

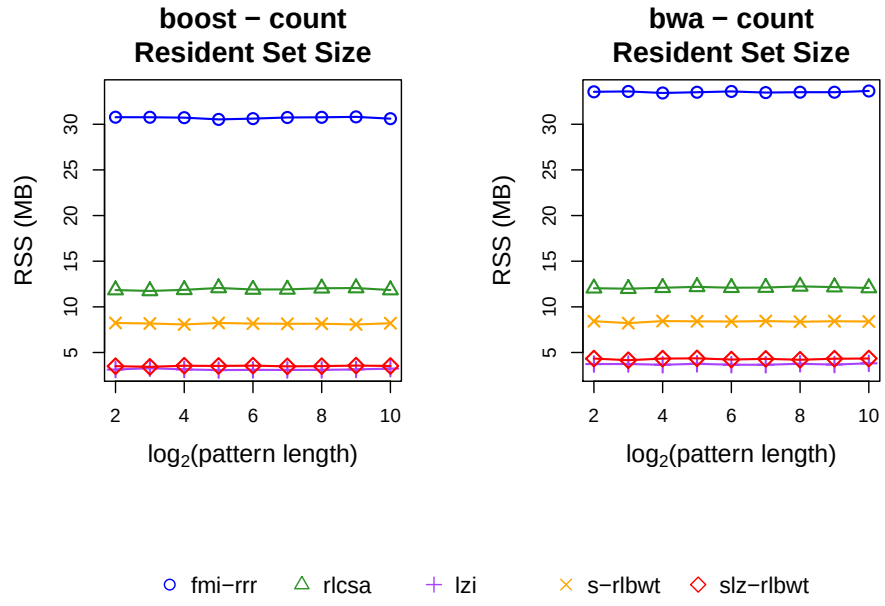


Figure 7.35: Count - Resident Set Size of the indexes on `boost` and `bwa`

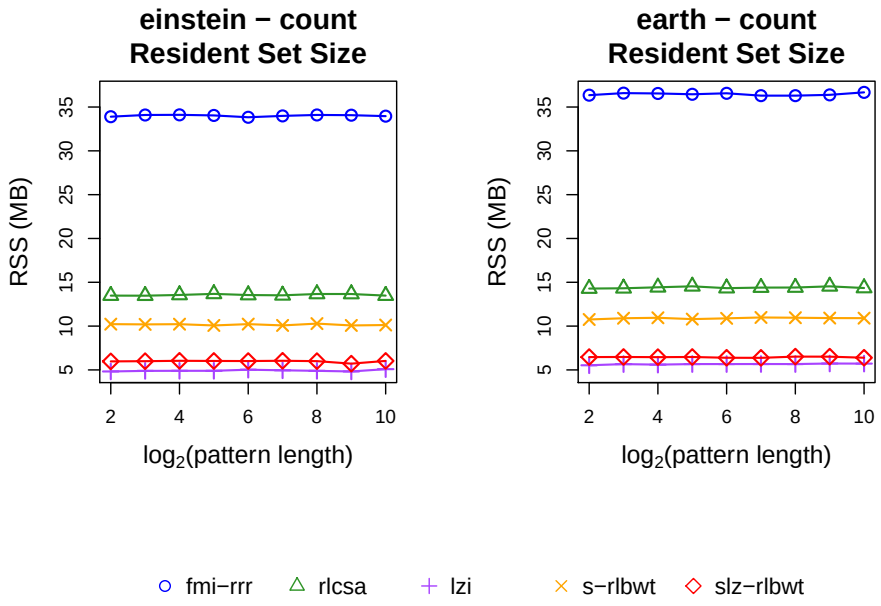


Figure 7.36: Count - Resident Set Size of the indexes on `einstein` and `earth`

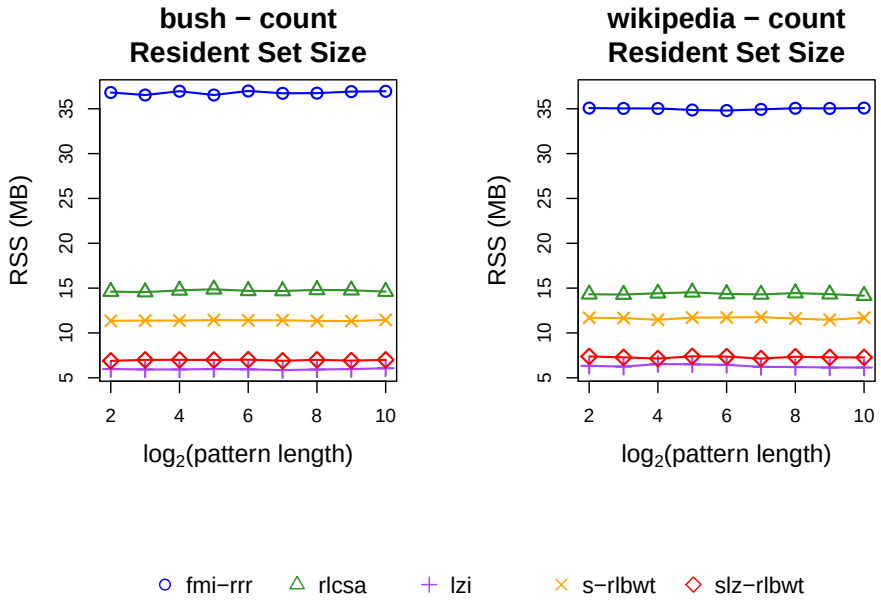


Figure 7.37: Count - Resident Set Size of the indexes on `bush` and `wikipedia`

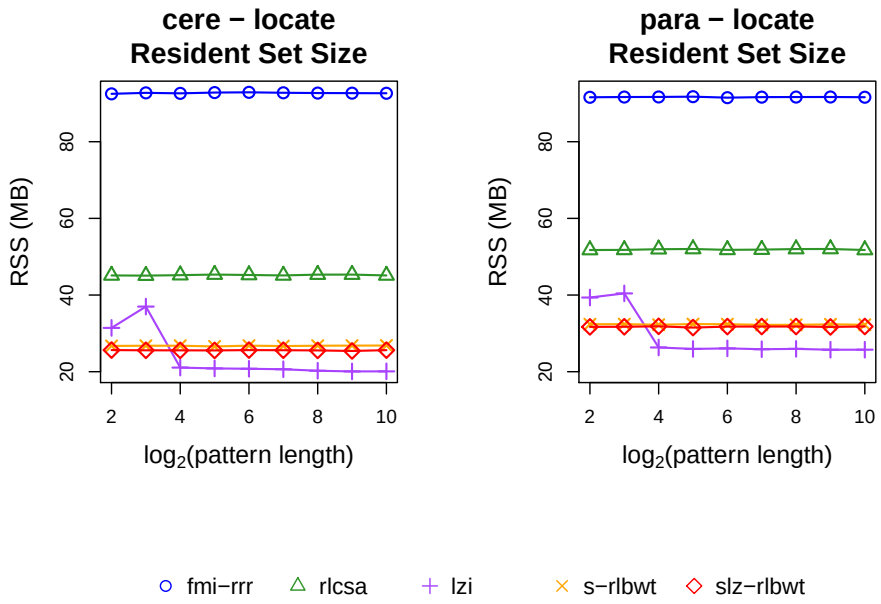


Figure 7.38: Locate - Resident Set Size of the indexes on `cere` and `para`

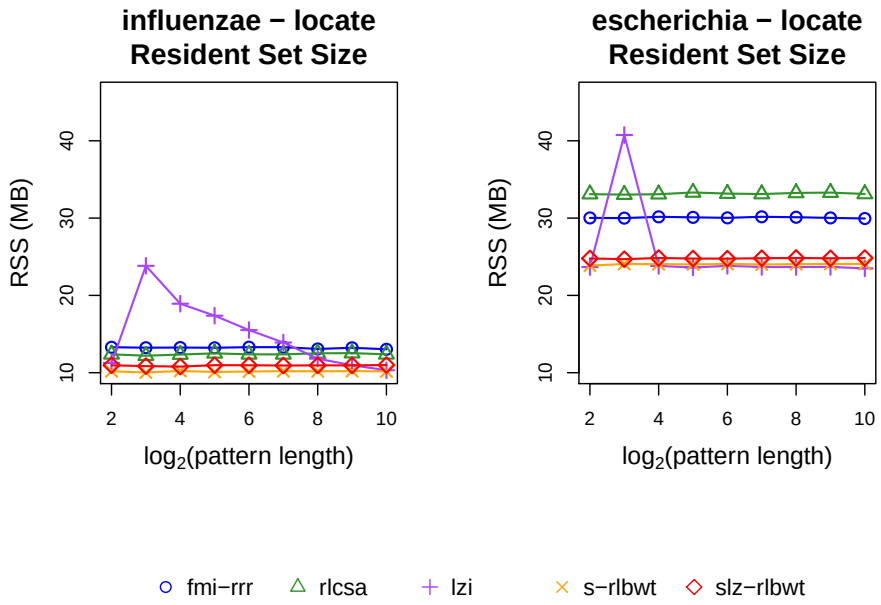


Figure 7.39: Locate - Resident Set Size of the indexes on `influenzae` and `escherichia`

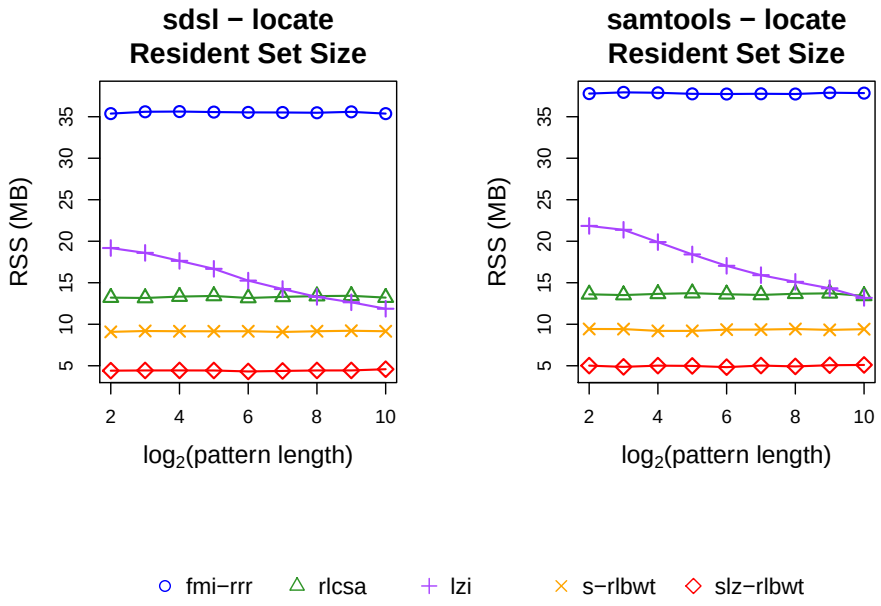


Figure 7.40: Locate - Resident Set Size of the indexes on `sdsl` and `samtools`

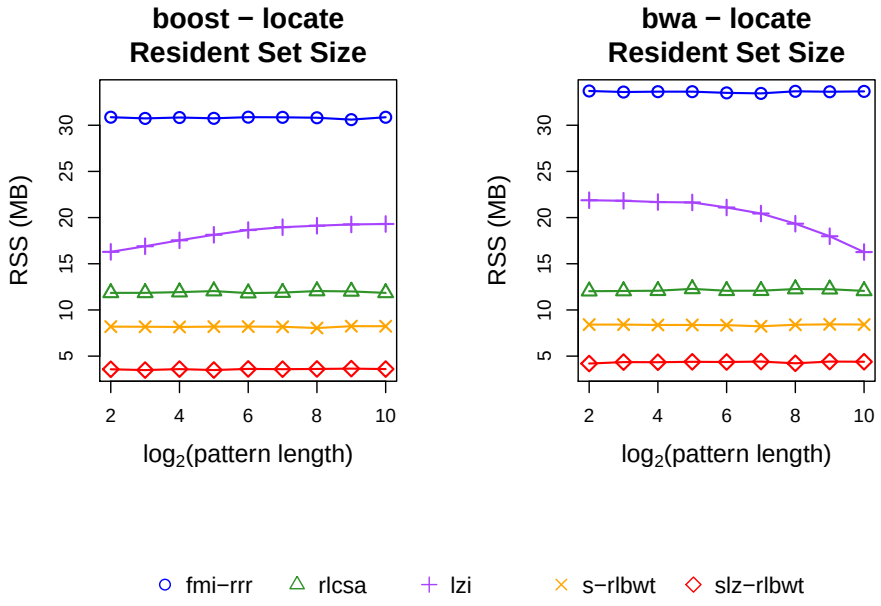


Figure 7.41: Locate - Resident Set Size of the indexes on `boost` and `bwa`

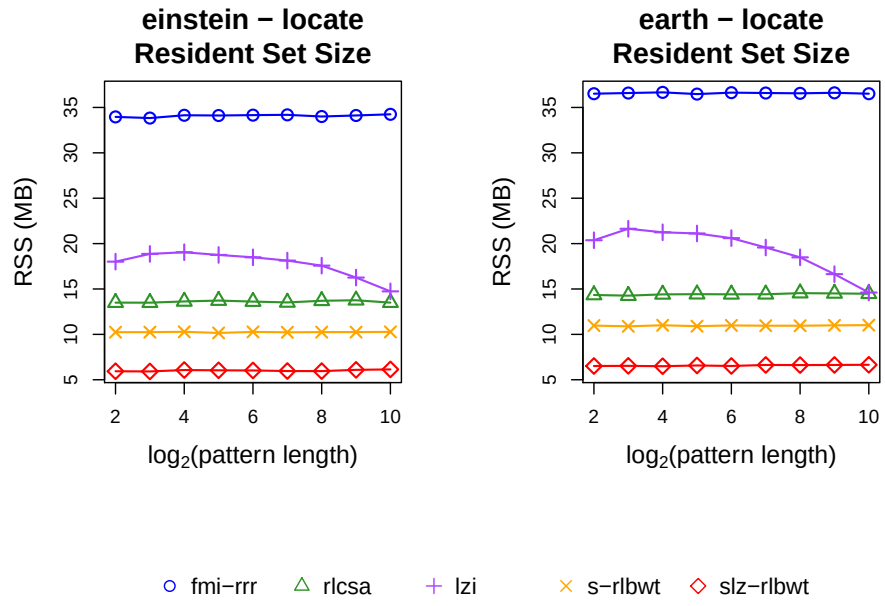


Figure 7.42: Locate - Resident Set Size of the indexes on `einstein` and `earth`

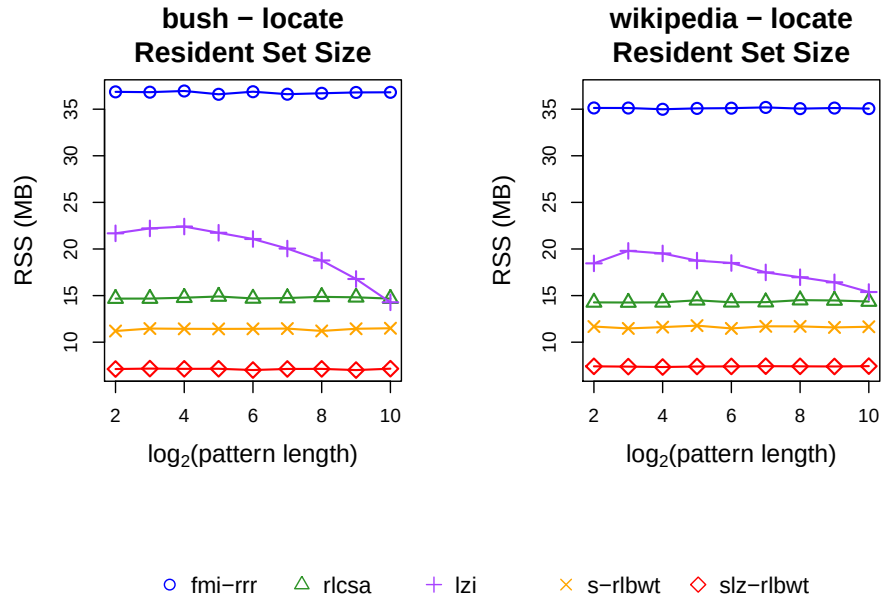


Figure 7.43: Locate - Resident Set Size of the indexes on `bush` and `wikipedia`

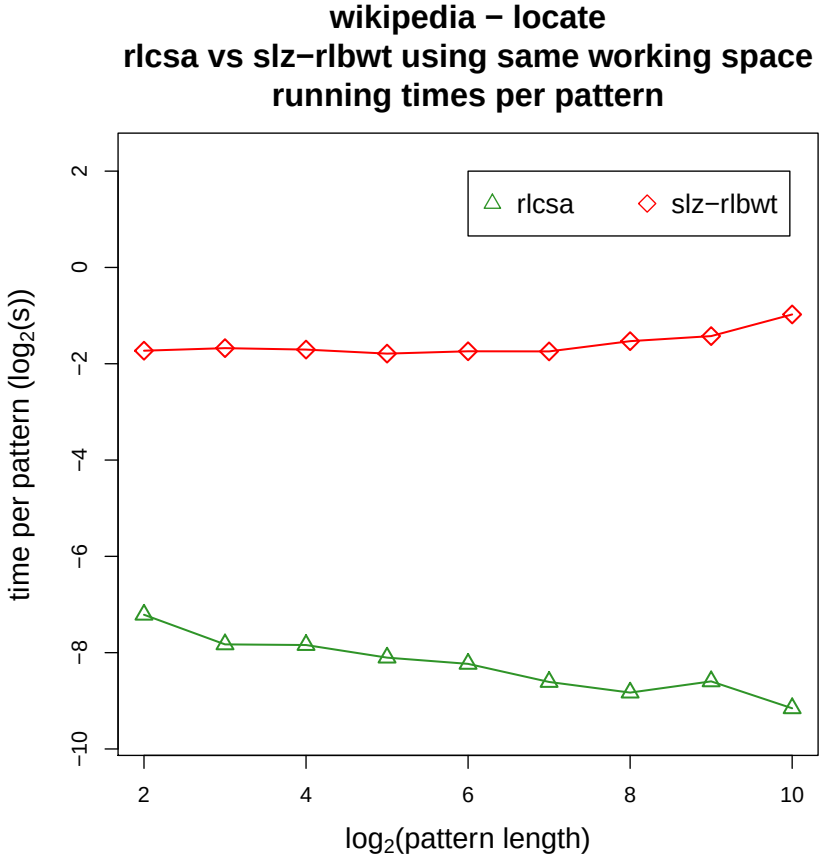


Figure 7.44: For a more fair comparison, we tested `rlcsa` and `slz-rlbwt` on `locate` queries on the `wikipedia` dataset choosing the sample rate for `rlcsa` in such a way that its final size matched that of `slz-rlbwt`. With a sample rate of 2200 for `rlcsa` and of 512 for `slz-rlbwt`, both indexes' sizes were approximately 4 MB. `rlcsa` is still the fastest index, supporting `locate` queries from 2^5 to 2^{10} times faster than `slz-rlbwt`.

8

Conclusions

In this thesis, we presented a framework of algorithms and data structures to compress and index text within compressed working space. We provided two algorithms computing a (entropy/run-length) compressed Burrows-Wheeler transform of the text, three algorithms computing the LZ77 factorization in (entropy/run-length) compressed working space, two algorithms to convert between LZ77 and run-length BWT within asymptotically optimal working space, and two full-text indexes for repetitive text collections. We moreover presented a C++ framework—`DYNAMIC`—implementing compressed dynamic data structures and proved (in theory and practice) tight bounds for the resources used by its components. All compression algorithms discussed in the thesis have been implemented using `DYNAMIC`. Our indexes for repetitive text collections were instead implemented using the succinct data structures library `sdsl` [43].

Theory On the theoretical side, our compression tools are among the first algorithms solving these problems within compressed working space. In particular, `rle-lz77-1` and `rle-lz77-2` are the first LZ77-factorization algorithms geared towards the compression of highly repetitive datasets and able to achieve exponentially-compressed working space on very repetitive sequences. `h0-lz77` and `cw-bwt` are, on the other hand, the first LZ77 and BWT algorithms working in entropy-compressed space within $\mathcal{O}(n \log n)$ and $\mathcal{O}(n \log \sigma)$ time, respectively (in the latter case, this running time is achieved only under the hypothesis of near-uniform text distribution).

Our algorithms to convert between run-length BWT and LZ77 in $\mathcal{O}(r + z)$ working space are the firsts of their kind. With these results, we extend the applicability of algorithms that can work only starting from one of the two compression formats (e.g. some indexing algorithms). In particular, using these results we showed—for the first time—that indexes based on LZ77 and RLBWT can be built in repetition-aware working space, i.e. up to exponentially less space than the text on very repetitive inputs.

Finally, our `s-rlbwt` and `slz-rlbwt` indexex are the firsts able to index the run-length Burrows-Wheeler transform in optimal space, and—in the second case—compress the suffix array sampling in a repetition-aware manner with the LZ77 scheme. Both indexes achieve optimal space usage with only a small penalty in query times with respect to the `rlcsa` index. Our most space-efficient index—`slz-rlbwt`—uses roughly 4|LZ77-d|+|RLBWT| space, i.e. 4 times the output of the LZ77-d factorization (whose size—as we showed—is several times smaller than that of the standard LZ77 factorization) plus the size of a run-length BWT. In practice—see below—this space is always very close to the most space-efficient index for repetitive text collections to date (i.e. the LZ77-index).

Practice Strong emphasis in this thesis was given to practice. With our `DYNAMIC` library we presented a new practical way of approaching succinctness in dynamic structures. Our SPSI (Searchable Partial Sums with Inserts) structure uses small space on top of the information-theoretic lower bound and supports logarithmic-time queries. We moreover show a practical reduction of the (gap-encoded/succinct) bitvector problem to the SPSI problem. `DYNAMIC` is the first freely available library offering many useful compressed dynamic structures: SPSIs, (gap-encoded, succinct) bitvectors, (succinct, entropy/run-length compressed) strings, (entropy/run-length compressed) FM-indexes. In practice, memory usage and running times of our structures follow very closely theoretical predictions.

Many things can be improved in our library. First of all, support for `delete` operations (not considered in this work as not needed in our algorithms). Then, a memory allocator as the one considered by Cordova and Navarro [21] to deal with memory fragmentation (accounting for most of the overhead of our structures on top of the allocated memory). Dynamic geometric structures (e.g. range search) and batch insert operations would also be interesting extensions of the library.

In our experiments, our repetition-aware compression algorithms `rle-bwt`, `rle-lz77-1`, and `rle-lz77-2` used up to three orders of magnitude less space than classic linear-space algorithms on highly repetitive datasets. To the best of our knowledge, this is the first time such a result is achieved, both from the theoretical and practical standpoint. The penalty our algorithms pay for working in such a small space is increased running times, which are up to three orders of magnitude higher than those of classical algorithms performing the same tasks. An interesting extension is therefore that of making these algorithms practical. We note that the bottleneck in our repetition-aware algorithms is the space-efficient construction of the run-length Burrows-Wheeler transform: this task requires to perform n `insert` queries, which we proved (Chapter 7) to be one to two orders of magnitude slower than `access` and `rank` queries on our dynamic strings. We anticipate that we are working on a more practical solution to this problem based on merging two *static* RLBWTs.

To conclude, we proved our indexes `s-rlbwt` and `slz-rlbwt` to be extremely competitive also in practice. In particular, the `slz-rlbwt` used—on average—a space only 1.1 times bigger than the lightest variant of the LZ77 index [67], i.e. the most space-efficient index for repetitive text collections to date. In addition to this space efficiency, we showed that `slz-rlbwt` supports `count` queries three orders of magnitude faster than the LZ77 index on all datasets and pattern lengths. The LZ77 index is, however, up to two orders of magnitude faster than `slz-rlbwt` on `locate` queries on patterns smaller than 2^{10}.

8.1 Future Directions of Research

From the development of the first full-text indexes in the early 70's, the areas of text compression and indexing have been the subjects of many exciting breakthroughs. Nowadays, we acknowledge that problems such as compression and indexing are two sides of the same coin: this remarkable connection allowed us to design advanced compressed full-text indexes supporting extremely fast pattern matching queries while occupying a space that could be thousands of times smaller than the plain text. Yet, much research is still going on in the field. Fully-dynamic compressed indexes (i.e. indexes supporting indels at any text position), as an instance, are attracting a lot of attention as they would remove the need of working on uncompressed files and then compressing/indexing them. Using a

dynamic compressed indexes, data would be compressed and indexed for the whole time of its existence. Pushing further this concept, one could even imagine entire databases and filesystems based on this idea: systems based on this technology would offer better performance (e.g. faster file search and edits) while—at the same time—efficiently storing the data in a compressed form. To date, few fully-dynamic compressed indexes have been designed, but they are either too slow in practice or too complicated to implement (see [117] and Section 6.3.3). This consideration brings up a related problem: can compressed dynamic data structures be practical, in addition to being theoretical appealing? As this thesis proved, straightforward implementations making use of balanced trees do not meet this criteria, as they are orders of magnitude slower than their static counterparts (despite offering very similar theoretical performance).

Another fascinating problem—which is lately attracting a lot of research—is that of compressing (and indexing) structured data such as graphs or sets of multi-dimensional points. Graphs are challenging for standard text-based compression algorithms due to their inherent non-linear structure. Considering the hardness of problems such as graph and subgraph isomorphism, however, optimally identifying repetitions in graphs is just a too complicated task to be solved in an exact manner. Sub-optimal grammar-based techniques (such as Re-Pair) have already been developed to tackle the problem (see, e.g. [17]). Alternatively, one could focus on easier-to-treat graph topologies. Extensions of LZ77 on labeled trees are currently being investigated, and techniques such as tree grammars [71,72] and top trees [9] have been shown to be particularly efficient in compressing this class of objects. On the graph-indexing side, recently there have been interesting advances. Given a labeled graph, a graph index is a data structure that supports fast pattern matching queries on *paths* of the graph. An optimal-space and fast extension of the Burrows-Wheeler transform for labeled trees has been recently described [32]. There even exists a version of the BWT for directed acyclic graphs [112], but it suffers from exponential blowup in size (with respect to the input graph size) in the worst case.

The optimization of existing indexes also offers exciting lines of research. In particular: can we support `locate` in optimal $\mathcal{O}(m + occ)$ time within $\mathcal{O}(nH_k)$ bits of space? can we support efficient random access (e.g. in logarithmic time) within $\mathcal{O}(z)$ words of space? can Lempel-Ziv indexes be augmented to support efficient `count` within asymptotically optimal working space? Does there exist a general sampling mechanism for the BWT taking $\mathcal{O}(r)$ words of space? (note that the sampling of Section 4.3.1 is not general as it supports only locating *at least one* pattern occurrence).

Index construction and text compression in optimal time and space is also still an open problem for several compression schemes. As shown in this thesis, the LZ77 factorization can be computed in $\mathcal{O}(r)$ words of space. On certain classes of texts, this space is even asymptotically smaller than z (see Section 5.1). However, for many texts of practical interest, z is often smaller than r. An open question remains, therefore, whether LZ77 can be computed efficiently within $\mathcal{O}(z)$ words of working space (see [39] for approximations of LZ77 within this working space). Analogously, very recently it has been shown [6, 82] that FM indexes can be computed in linear time and compact working space. Can the same task be performed in the same time within $\mathcal{O}(nH_k)$ bits of space? Similar questions arise in the field of grammar compression. To date, the most space-efficient algorithm computing the Re-Pair compression scheme uses $n + \sqrt{n}$ words of space on top of the input [10]. This is even asymptotically bigger than the sheer size of the plain text, and could be thousands of times larger than the final compressed representation if the text is

very repetitive. In general, fast compression within asymptotically-optimal working space is still an open and intriguing problem for many compression schemes.

Bibliography

[1] R. A. Baeza-Yates and G. H. Gonnet. A New Approach to Text Searching. In *Proceedings of the 12th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '89, pages 168–175, New York, NY, USA, 1989. ACM.

[2] Hideo Bannai, Paweł Gawrychowski, Shunsuke Inenaga, and Masayuki Takeda. Converting SLP to LZ78 in almost Linear Time. In *Combinatorial Pattern Matching*, pages 38–49. Springer, 2013.

[3] Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Efficient LZ78 factorization of grammar compressed text. In *String Processing and Information Retrieval*, pages 86–98. Springer, 2012.

[4] Djamal Belazzougui. Linear time construction of compressed text indices in compact space. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, pages 148–193. ACM, 2014.

[5] Djamal Belazzougui, Fabio Cunial, Travis Gagie, Nicola Prezza, and Mathieu Raffinot. Composite repetition-aware data structures. In *Proc. CPM*, pages 26–39, 2015.

[6] Djamal Belazzougui, Fabio Cunial, Juha Kärkkäinen, and Veli Mäkinen. Linear-time string indexing and analysis in small space. *arXiv preprint arXiv:1609.06378*, 2016.

[7] Djamal Belazzougui and Simon J Puglisi. Range predecessor and Lempel-Ziv parsing. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2053–2071. SIAM, 2016.

[8] Timo Beller, Maike Zwerger, Simon Gog, and Enno Ohlebusch. Space-efficient construction of the Burrows-Wheeler transform. In *String Processing and Information Retrieval*, pages 5–16. Springer, 2013.

[9] Philip Bille, Inge Li Gørtz, Gad M Landau, and Oren Weimann. Tree compression with top trees. *Information and Computation*, 243:166–177, 2015.

[10] Philip Bille, Inge Li Gørtz, and Nicola Prezza. Space-Efficient Re-Pair Compression. In *Data Compression Conference (DCC), 2017*. IEEE, 2017.

[11] Michael Burrows and David J Wheeler. A block-sorting lossless data compression algorithm, 1994.

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

[13] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, Amit Sahai, and Abhi Shelat. The smallest grammar problem. *Information Theory, IEEE Transactions on*, 51(7):2554–2576, 2005.

[14] David Clark. *Compact Pat trees*. PhD thesis, PhD thesis, University of Waterloo, 1998.

[15] Francisco Claude. libcds: compact data structures library. `https://github.com/fclaude/libcds`. Accessed: 2016-11-17.

[16] Francisco Claude, Antonio Farina, Miguel A Martínez-Prieto, and Gonzalo Navarro. Universal indexes for highly repetitive document collections. *Information Systems*, 61:1–23, 2016.

[17] Francisco Claude and Gonzalo Navarro. A fast and compact Web graph representation. In *International Symposium on String Processing and Information Retrieval*, pages 118–129. Springer, 2007.

[18] Francisco Claude and Gonzalo Navarro. Self-indexed text compression using straight-line programs. In *International Symposium on Mathematical Foundations of Computer Science*, pages 235–246. Springer, 2009.

[19] Francisco Claude and Gonzalo Navarro. Self-indexed grammar-based compression. *Fundamenta Informaticae*, 111(3):313–337, 2011.

[20] Francisco Claude and Gonzalo Navarro. Improved grammar-based compressed indexes. In *International Symposium on String Processing and Information Retrieval*, pages 180–192. Springer, 2012.

[21] Joshimar Cordova and Gonzalo Navarro. Practical dynamic entropy-compressed bitvectors with applications. In *International Symposium on Experimental Algorithms*, pages 105–117. Springer, 2016.

[22] Maxime Crochemore and Lucian Ilie. Computing longest previous factor in linear time and applications. *Information Processing Letters*, 106(2):75–80, 2008.

[23] Maxime Crochemore, Lucian Ilie, and William F Smyth. A simple algorithm for computing the Lempel-Ziv factorization. In *18th Data Compression Conference (DCC'08)*, pages 482–488. IEEE Computer Society Press, Los Alamitos, CA, 2008.

[24] Nicolaas Govert de Bruijn and Paul Erdos. A combinatorial problem. *Koninklijke Nederlandse Akademie v. Wetenschappen*, 49(49):758–764, 1946.

[25] ds-vector: C++ library for dynamic succinct vector. `https://code.google.com/archive/p/ds-vector/`. Accessed: 2016-11-17.

[26] Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM (JACM)*, 21(2):246–260, 1974.

[27] Peter Elias. Universal codeword sets and representations of the integers. *IEEE transactions on information theory*, 21(2):194–203, 1975.

[28] Robert Mario Fano. *On the number of bits required to implement an associative memory*. Massachusetts Institute of Technology, Project MAC, 1971.

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

[30] Paolo Ferragina, Raffaele Giancarlo, and Giovanni Manzini. The myriad virtues of wavelet trees. *Information and Computation*, 207(8):849–866, 2009.

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

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

[33] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

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

[35] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly FM-index. In *String Processing and Information Retrieval*, pages 150–160. Springer, 2004.

[36] Paolo Ferragina, Giovanni Manzini, Veli Mäkinen, and Gonzalo Navarro. An alphabet-friendly FM-index. In *String Processing and Information Retrieval*, pages 150–160. Springer, 2004.

[37] Paolo Ferragina and Gonzalo Navarro. Pizza&Chili corpus. `http://pizzachili.dcc.uchile.cl`. Accessed: 2016-07-25.

[38] Gabriele Fici. Factorizations of the Fibonacci Infinite Word. *Journal of Integer Sequences*, 18(2):3, 2015.

[39] Johannes Fischer, Travis Gagie, Paweł Gawrychowski, and Tomasz Kociumaka. Approximating LZ77 via small-space multiple-pattern matching. In *Algorithms-ESA 2015*, pages 533–544. Springer, 2015.

[40] Michael Fredman and Michael Saks. The cell probe complexity of dynamic data structures. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 345–354. ACM, 1989.

[41] Travis Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99(6):246–251, 2006.

[42] Nicola Gigante. bitvector: succinct dynamic bitvector implementation. `https://github.com/nicola-gigante/bitvector`. Accessed: 2016-11-17.

[43] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[44] Gaston H Gonnet, Ricardo A Baeza-Yates, and Tim Snider. New Indices for Text: Pat Trees and Pat Arrays. *Information Retrieval: Data Structures & Algorithms*, 66:82, 1992.

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

[46] Keisuke Goto, Shirou Maruyama, Shunsuke Inenaga, Hideo Bannai, Hiroshi Sakamoto, and Masayuki Takeda. Restructuring compressed texts without explicit decompression. *arXiv preprint arXiv:1107.2729*, 2011.

[47] 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*, pages 841–850. Society for Industrial and Applied Mathematics, 2003.

[48] Ankur Gupta, Wing-Kai Hon, Rahul Shah, and Jeffrey Scott Vitter. Compressed data structures: Dictionaries and data-aware measures. In *Data Compression Conference (DCC'06)*, pages 213–222. IEEE, 2006.

[49] Meng He and J Ian Munro. Succinct representations of dynamic strings. In *String Processing and Information Retrieval*, pages 334–346. Springer, 2010.

[50] Danny Hucke, Markus Lohrey, and Carl Philipp Reh. The smallest grammar problem revisited. In *International Symposium on String Processing and Information Retrieval*, pages 35–49. Springer, 2016.

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

[52] Guy Joseph Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, 1988.

[53] Artur Jeż. Recompression: Word equations and beyond. In *International Conference on Developments in Language Theory*, pages 12–26. Springer, 2013.

[54] Artur Jeż. Approximation of grammar-based compression via recompression. *Theoretical Computer Science*, 592:115–134, 2015.

[55] Artur Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms (TALG)*, 11(3):20, 2015.

[56] Artur Jeż. Recompression: a simple and powerful technique for word equations. *Journal of the ACM (JACM)*, 63(1):4, 2016.

[57] Juha Kärkkäinen, Dominik Kempa, and Simon J Puglisi. Lightweight Lempel-Ziv parsing. In *Experimental Algorithms*, pages 139–150. Springer, 2013.

[58] Juha Kärkkäinen, Dominik Kempa, and Simon J Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Combinatorial Pattern Matching*, pages 189–200. Springer, 2013.

[59] Juha Kärkkäinen and Simon J Puglisi. Fixed block compression boosting in FM-indexes. In *International Symposium on String Processing and Information Retrieval*, pages 174–184. Springer, 2011.

[60] Juha Kärkkäinen and Esko Ukkonen. Lempel-Ziv parsing and sublinear-size index structures for string matching. In *Proc. 3rd South American Workshop on String Processing (WSP'96*. Citeseer, 1996.

[61] Dominik Kempa and Simon J Puglisi. Lempel-Ziv factorization: Simple, fast, practical. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 103–112. Society for Industrial and Applied Mathematics, 2013.

[62] Patrick Klitzke and Patrick K Nicholson. A general framework for dynamic succinct and compressed data structures. *Proceedings of the 18th ALENEX*, pages 160–173, 2016.

[63] S Rao Kosaraju and Giovanni Manzini. Compression of Low Entropy Strings with Lempel–Ziv Algorithms. *SIAM Journal on Computing*, 29(3):893–911, 2000.

[64] Sebastian Kreft. LZ77 index. `http://pizzachili.dcc.uchile.cl/indexes/LZ77-index/`. Accessed: 2016-11-25.

[65] Sebastian Kreft and Gonzalo Navarro. Self-index based on LZ77 (MSc thesis). *arXiv preprint arXiv:1112.4578*, 2011.

[66] Sebastian Kreft and Gonzalo Navarro. Self-indexing based on LZ77. In *Combinatorial Pattern Matching*, pages 41–54. Springer, 2011.

[67] Sebastian Kreft and Gonzalo Navarro. On compressing and indexing repetitive sequences. *Theoretical Computer Science*, 483:115–133, 2013.

[68] Tak Wah Lam, Ruiqiang Li, Alan Tam, Simon Wong, Edward Wu, and Siu-Ming Yiu. High throughput short read alignment via bi-directional BWT. In *Bioinformatics and Biomedicine, 2009. BIBM'09. IEEE International Conference on*, pages 31–36. IEEE, 2009.

[69] Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *Information Theory, IEEE Transactions on*, 22(1):75–81, 1976.

[70] Ross A Lippert, Clark M Mobarry, and Brian P Walenz. A space-efficient construction of the Burrows-Wheeler transform for genomic data. *Journal of Computational Biology*, 12(7):943–951, 2005.

[71] Markus Lohrey. Grammar-based tree compression. In *International Conference on Developments in Language Theory*, pages 46–57. Springer, 2015.

[72] Markus Lohrey, Sebastian Maneth, and Roy Mennicke. Tree structure compression with repair. In *Data Compression Conference (DCC), 2011*, pages 353–362. IEEE, 2011.

[73] LZ77 factorization algorithms. `https://www.cs.helsinki.fi/group/pads/lz77.html`. Accessed: 2016-05-20.

[74] Veli Mäkinen and Gonzalo Navarro. Implicit compression boosting with applications to self-indexing. In *International Symposium on String Processing and Information Retrieval*, pages 229–241. Springer, 2007.

[75] Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.

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

[77] 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.

[78] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327. ACM, 1990.

[79] Sabrina Mantaci, Antonio Restivo, and Marinella Sciortino. Burrows–Wheeler transform and Sturmian words. *Information Processing Letters*, 86(5):241–246, 2003.

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

[81] Y Mori. Short description of improved two-stage suffix sorting algorithm, 2005.

[82] J Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Space-efficient construction of compressed indexes in deterministic linear time. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 408–424. SIAM, 2017.

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

[84] Gonzalo Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

[85] Gonzalo Navarro. *Compact Data Structures: A Practical Approach*. Cambridge University Press, 2016.

[86] Gonzalo Navarro. A self-index on block trees. *arXiv preprint arXiv:1606.06617*, 2016.

[87] Gonzalo Navarro and Yakov Nekrich. Optimal dynamic sequence representations. *SIAM Journal on Computing*, 43(5):1781–1806, 2014.

[88] Gonzalo Navarro and Kunihiko Sadakane. Fully functional static and dynamic succinct trees. *ACM Transactions on Algorithms (TALG)*, 10(3):16, 2014.

[89] Takaaki Nishimoto, Shunsuke Inenaga, Hideo Bannai, Masayuki Takeda, et al. Dynamic index, LZ factorization, and LCE queries in compressed space. *arXiv preprint arXiv:1504.06954*, 2015.

[90] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Data Compression Conference, 2009. DCC'09.*, pages 193–202. IEEE, 2009.

[91] Enno Ohlebusch and Simon Gog. Lempel-Ziv factorization revisited. In *Combinatorial Pattern Matching*, pages 15–26. Springer, 2011.

[92] Daisuke Okanohara and Kunihiko Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the Meeting on Algorithm Engineering & Expermiments*, pages 60–70. Society for Industrial and Applied Mathematics, 2007.

[93] Daisuke Okanohara and Kunihiko Sadakane. An online algorithm for finding the longest previous factors. In *Algorithms-ESA 2008*, pages 696–707. Springer, 2008.

[94] Alessio Orlandi. *Advanced rank/select data structures: succinctness, bounds, and applications*. PhD thesis, Ph. D. thesis, 2012.

[95] Giuseppe Ottaviano. succinct library. `https://github.com/ot/succinct`. Accessed: 2016-11-17.

[96] Nicola Prezza. BWTIL: static succinct data structures. `https://github.com/nicolaprezza/BWTIL`. Accessed: 2016-11-17.

[97] Nicola Prezza. DYNAMIC: dynamic succinct/compressed data structures library. `https://github.com/nicolaprezza/DYNAMIC`. Accessed: 2016-11-17.

[98] Nicola Prezza. FMI: a wrapper on sdsl's FM-index. `https://github.com/nicolaprezza/FMI`. Accessed: 2016-11-25.

[99] Nicola Prezza. get-git-revisions: Get all revisions of a git repository. `https://github.com/nicolaprezza/get-git-revisions`. Accessed: 2016-11-17.

[100] Nicola Prezza. lz-rlbwt: Run-length compressed BWT with LZ77 sampled suffix array. `https://github.com/nicolaprezza/lz-rlbwt`. Accessed: 2016-11-17.

[101] Nicola Prezza. s-rlbwt: Sparse run-length compressed BWT. `https://github.com/nicolaprezza/s-rlbwt`. Accessed: 2016-11-25.

[102] Nicola Prezza. slz-rlbwt: Run-length compressed BWT with sparse LZ77 sampled suffix array. `https://github.com/nicolaprezza/slz-rlbwt`. Accessed: 2016-12-7.

[103] Nicola Prezza. wiki-get: Download all versions of a Wikipedia page. `https://github.com/nicolaprezza/wiki_get`. Accessed: 2016-11-17.

[104] Rajeev Raman, Venkatesh Raman, and S Srinivasa Rao. Succinct dynamic data structures. In *Algorithms and Data Structures*, pages 426–437. Springer, 2001.

[105] Rajeev Raman, Venkatesh Raman, and Srinivasa Rao Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms (TALG)*, 3(4):43, 2007.

[106] Wojciech Rytter. Application of Lempel–Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1):211–222, 2003.

[107] Kunihiko Sadakane. dbwt: direct construction of the BWT. `http://researchmap.jp/muuw41s7s-1587/#_1587`. Accessed: 2016-11-17.

[108] Thomas Schnattinger, Enno Ohlebusch, and Simon Gog. Bidirectional search in a string with wavelet trees. In *Annual Symposium on Combinatorial Pattern Matching*, pages 40–50. Springer, 2010.

[109] Claude Elwood Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–656, 1948.

[110] Jouni Sirén. Run-Length compressed suffix array (RLCSA). `http://jltsiren.kapsi.fi/rlcsa`. Accessed: 2016-11-17.

[111] Jouni Sirén. *Compressed full-text indexes for highly repetitive collections*. PhD thesis, Helsingin yliopisto, 2012.

[112] Jouni Sirén, Niko Välimäki, and Veli Mäkinen. Indexing graphs for path queries with applications in genome research. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 11(2):375–388, 2014.

[113] Victor Smirnov. Memoria: C++14 framework providing general purpose dynamic data structures. `https://bitbucket.org/vsmirnov/memoria/wiki/Home`. Accessed: 2016-11-17.

[114] Tatiana Starikovskaya. Computing Lempel-Ziv factorization online. In *Mathematical Foundations of Computer Science 2012*, pages 789–799. Springer, 2012.

[115] Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto. Improved ESP-index: a practical self-index for highly repetitive texts. In *International Symposium on Experimental Algorithms*, pages 338–350. Springer, 2014.

[116] Yoshimasa Takabatake, Yasuo Tabei, and Hiroshi Sakamoto. Online self-indexed grammar compression. In *proc. SPIRE*, volume 9309 of *Lecture Notes in Computer Science*, pages 258–269. Springer International Publishing, 2015.

[117] Masayuki Takeda. Dynamic Index and LZ Factorization in Compressed Space. In *Prague Stringology Conference 2016*, page 158, 2016.

[118] Yuya Tamakoshi, I Tomohiro, Shunsuke Inenaga, Hideo Bannai, and Masanori Takeda. From run length encoding to LZ78 and back again. In *Data Compression Conference (DCC), 2013*, pages 143–152. IEEE, 2013.

[119] German Tischler. Faster average case low memory semi-external construction of the Burrows-Wheeler transform. In *CEUR Workshop Proceedings,*, volume 1146, pages 61–68, 2014.

[120] Sebastiano Vigna. sux library. `http://sux.di.unimi.it/`. Accessed: 2016-11-17.

[121] Peter Weiner. Linear pattern matching algorithms. In *Switching and Automata Theory, 1973. SWAT'08. IEEE Conference Record of 14th Annual Symposium on*, pages 1–11. IEEE, 1973.

[122] Jun'ichi Yamamoto, Tomohiro I, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Faster Compact On-Line Lempel-Ziv Factorization. In *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014)*, volume 25 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 675–686, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[123] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[124] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536, 1978.