 Parsing Algorithms for Data Compression

Igor Nitto

Supervisor
prof. Paolo Ferragina

April 26, 2010
Abstract

The task of parsing consists of splitting an input text into a sequence of contiguous phrases. Several classes of data compression algorithms rely on parsing techniques either to identify repetitions inside the input string (*dictionary-based compression*) or to locate homogeneous pieces of data which are separately compressed (*Permute-Partition-Compress paradigm*).

In these applications, the choice of the parsing strategy is determinant for the final performance of the compressor. An ideal parsing algorithm should be able to parse the input in a way that minimizes the output-size of the underlying compressor, and the question is how efficiently this can be done. Many investigations have focused on parsing algorithms that achieve optimality in the compressor’s output-size but the solutions proposed in literature are far from being satisfactory. In fact, most of them are either simple approaches based on dynamic programming with prohibitive time complexities [10, 79, 42], or heuristic algorithms which do not offer any bounds on the efficacy of the solution [11, 56, 81, 7, 18].

We propose a new approach to the design of optimal parsing algorithms, achieving significant improvements in running time over previous methods. As originally pointed out by [79], this problem can be modeled as a shortest-path computation over a particular directed-acyclic graph. We build upon this idea by showing that the class of graphs arising from this reduction satisfies particular structural properties that can be exploited by our algorithms to speed-up a lot shortest-path computation.

We obtain new results by applying this approach to the contexts of dictionary-based compression and Permute-Partition-Compress paradigm. We consider the class of *LZ77*-based compressors, the most powerful example of dictionary-based compression, and design the first parser which achieve optimality in the compressed output size (measured in bits) by taking efficient/optimal time and optimal space [35]; this solves an open problem raised in [76, pag. 159]. Then, using similar techniques, we provide an approximate parsing algorithm that, when used inside the Permute-Partition-Compress paradigm, produces a compressed output whose size is guaranteed to be no more than $(1 + \epsilon)$-worse the optimal one, where $\epsilon$ is a user defined constant [34].
To my parents
Contents

I Introductory Material

  Introduction ............................................. 9
    I.1 Parsing Algorithms for Data Compression ................. 9
      I.1.1 The bit-optimal LZ77-parsing problem .............. 10
      I.1.2 The optimal partitioning problem .................. 11
    I.2 Remapping to shortest-path computation ................. 12
    I.3 Contributions: A Novel Pruning Based Approach .......... 13
    I.4 Map of the Thesis .................................... 15
    I.5 Notations and Model of Computation ..................... 16

1 Preliminaries ............................................ 17

  1.1 Tries, Suffix Trees and Suffix Arrays ..................... 17
  1.2 Basic Tools for Lossless Compression ..................... 19
    1.2.1 Universal Codes .................................. 20
    1.2.2 Empirical Entropy ................................ 21
    1.2.3 Statistical Encoders ................................ 22
    1.2.4 The Burrows-Wheeler Transform ...................... 23

II Dictionary-Based Compressors ............................................ 27

  2 Background ............................................. 29
    2.1 Anatomy of Dictionary-Based Compressors ................. 29
      2.1.1 Dictionary Construction Schemes ...................... 30
      2.1.2 Parsing strategies ................................ 31
    2.2 The Lempel-Ziv Family ................................ 32
      2.2.1 The LZ77 Algorithm ................................ 32
      2.2.2 The LZ78 Algorithm ................................ 34
      2.2.3 Compression Tools Based on LZ77/78 Algorithms .... 34
    2.3 The ACB compressor .................................... 36
    2.4 Compression Issues .................................... 37
      2.4.1 Entropy Bounds .................................... 37
      2.4.2 Greedy vs Non-Greedy ......................... 38
CHAPTER 0. CONTENTS

3 Efficient Bit-Optimal Lempel-Ziv Compression 39
  3.1 Introduction ................................................. 39
  3.2 Problem Definition ......................................... 40
  3.3 Sub-Optimality of Greedy Parsing ......................... 41
  3.4 Pruned Parsing Graph ....................................... 42
  3.5 An Efficient Bit-Optimal Parser ......................... 45
    3.5.1 Some Notation and Basic Facts ...................... 46
    3.5.2 Dynamic Generation of Maximal Edges ............. 47
    3.5.3 Achieving $O(1)$ amortized time per edge ........ 49
    3.5.4 Building $T_B$ optimally ......................... 52
  3.6 Experimental Results ................................... 53

4 Extensions 57
  4.1 Overview .................................................. 57
  4.2 Bit-Optimal ACB Compression ............................. 58
    4.2.1 Pruned Parsing Graph .............................. 58
    4.2.2 Generating Maximal Edges Efficiently ............. 60
    4.2.3 Fast Encoding of Phrases .......................... 61
  4.3 Bit-Optimal Greedy Parsing ................................ 63

III Reorganizing Data for Compression 67

5 The PPC-paradigm 69
  5.1 Theory and Practice of PPC ............................. 69
  5.2 Table Compression ....................................... 70
  5.3 Text Compression and the Burrows-Wheeler Transform ... 71
    5.3.1 Overview ........................................... 71
    5.3.2 Compression Boosting and the PPC-paradigm ....... 72
  5.4 Space-Constrained Compression .......................... 74

6 Fast Approximate Optimal Partitioning 77
  6.1 The Optimal Partitioning Problem ....................... 77
  6.2 An Approximation Scheme ................................ 78
  6.3 Compressive Estimates .................................. 79
  6.4 The Parsing Graph and a Pruning Strategy ............. 80
    6.4.1 A pruning strategy ................................ 80
  6.5 Space and time efficient algorithms for generating $G_r(S)$ 82
    6.5.1 A General Scheme .................................. 82
    6.5.2 0th Order Compressors ............................ 83
    6.5.3 $k$th Order Compressors ........................... 86
  6.6 Extension to BWT-based Compressor ...................... 87
Part I

Introductory Material
Introduction

I.1 Parsing Algorithms for Data Compression

A string parsing (or factorization) is a partition of the input string into a sequence of contiguous substrings, called phrases. Our main interest in parsing algorithms is driven by their applications to the field of Lossless Data Compression. There exist at least two famous approaches to data compression that crucially rely on parsing methods:

(i) Dictionary-Based Compression: one of the most popular lossless data-compression schemes, pioneered by the works of Lempel and Ziv in the late 70s, and implemented by many (non-)commercial compression programs — like gzip, zip, pkzip, arj, rar, just to cite a few. A dictionary-based compressor parses the input string into phrases, and then replaces each phrase with its corresponding codeword, which is actually a pointer into a dictionary of phrases. Compression occurs whenever the size of the codeword is smaller than that of the corresponding phrase. In dictionary-based compression, the parsing strategy is determinant for the final compression ratio, as the smallest is the number of parsed phrases the shortest is typically the compressed text.

(ii) Permute-Partition-Compress: a general paradigm for the design of lossless compression algorithms based on the principle that compressibility of an input string can be improved by re-organizing its symbols before compression. The reorganization process consists in permuting the input symbols in a way that the new permuted string can be partitioned into a sequence of contiguous substrings that can be compressed individually (and better) by a given base compressor. The intuition is that compressing separately the different parts in which the input string has been partitioned may give an improvement over compressing the entire input at once. In fact, if the parts of the partition are homogeneous, then one can hope to compress them very well even using a weak base-compressor. Starting from these considerations, many researches have developed parsing algorithms that try to maximize the redundancy inside each part of the permuted string, in order to achieve its best compression once its permutation is given. As discussed in Chapter 5, this approach has been used in a wide range of data compression applications: like Table Compression,
compression of massive collections of texts, and in compression based on the Burrows-Wheeler Transform (see Sections 5.2, 5.3, 5.4).

In the above compression schemes, the parsing step has a crucial impact on the size of the compressed output, and a fundamental question is how to efficiently compute a parsing which minimizes the output-size of the compressor. The main research problems investigated in this thesis will concern with the analysis of this question in the above two contexts.

I.1.1 The bit-optimal LZ77-parsing problem

Let us first consider dictionary-based compressors, and illustrate by an example the problem of parsing a text in such a way that the output-size of the compressor is minimized. Assume the simplest case of a static dictionary, which is fixed in advance and consists of a list of phrases and their corresponding codewords. For instance, the dictionary may consist of the phrases \{a, b, ab, bab\}, respectively identified by the integer codewords \{0, 1, 2, 3\}. Given an arbitrary input string \(S = ababab\), the compressor first parses \(S\) into a sequence of dictionary phrases, for instance \(S = ab|a|bab\), and then replaces these phrases by their corresponding codewords 213. The compressed string is eventually obtained by representing each codeword in binary with a prefix-free code. Clearly the same input string may admit many different parsings, and we aim at computing a parsing into phrases of \(D\) that minimizes the total bit-length of the output codewords, and refer it as a bit-optimal parsing for \(D\). For instance, if in the above example we use a single bit to encode the codeword 2, and encode all the other codewords with three bits, then the bit-optimal parsing of \(S\) is \(S = ab|ab|ab\).

A particularly interesting case is that of the well-known LZ77 algorithm (Section 2.2), proposed by Lempel and Ziv in [94], and popularized by the compression program gzip. This is a dynamic dictionary method, which compresses the text via a single scan, by replacing repeated substrings with codewords that represent backward references to previous occurrence of a phrase in the text.

Although many fundamental results are nowadays known about the speed and effectiveness of this compression process, no efficient parsing scheme for the LZ77 dictionary is known that achieves bit-optimality in the general case of variable-length codewords [76, pag. 159]. In fact, the only known solutions are either inefficient [79], in that they require quadratic time, or they rely on heuristics which do not guarantee optimality of the parsing, such as the greedy match used by gzip which we prove to be far from optimal of a factor \(O(\log n / \log \log n)\), where \(n\) is the length of the string to be compressed (see Lemma 5). The construction of an efficient bit-optimal parser for the LZ77 scheme is thus one of the main problems left open in this context, and we will dedicate the first part of this thesis to its solution.
I.1.2 The optimal partitioning problem

The Permute-Partition-Compress (or PPC for brevity) paradigm is a general methodology for designing compression algorithms. Compressors based on the PPC-paradigm follow a common scheme which consists of *permuting* the input string $S$ to form a new string $S'$ which is then *partitioned* into substrings $S' = S_1' \cdot S_2' \cdots S_k'$ that are finally compressed *individually* by means of a base compressor $C$. The concept is that, by cleverly exploiting the combined steps permuting+partitioning, one can split the input string $S$ into a sequence of substrings that, compressed individually, achieve an output shorter than compressing the whole $S$ at once. In many scenarios, this simple idea has been turned into very effective compression strategies.

Some of the first results on the PPC-paradigm were obtained in the context of Table Compression (Section 5.2), where the input is a sequence of fixed-size columns. In [10], the authors observed experimentally that properly rearranging and then partitioning the table into intervals of columns that are separately compressed can significantly improve the overall compression. To permute the columns they devised an heuristic algorithm based on a reduction to the TSP problem, while for the partitioning step they proposed a simple cubic-time solution based on dynamic programming.

In the context of text compression, the most famous instantiation of the PPC-paradigm has been obtained by applying a context-based partitioning of the input characters over the Burrows and Wheeler Transform (Section 1.2.4) of the input text (Section 5.3). The substrings into which the BW-transformed text is partitioned are then compressed with a simple statistical compressor $C$ (such as Huffman, or Arithmetic). This scheme is named *compression booster*, as the net result it produces is to turn any memoryless base compressor $C$ into a compressor that automatically selects the optimal contexts and thus scales to $k$th order memories, for any $k \geq 0$.

Finally, partitioning and permuting are mandatory steps when the input-size is much larger than the maximum amount of data that the base compressor can process at once (Section 5.4). In fact, for efficiency reasons, many compression tools such as *gzip*, *bzip2* and others keep the size of the working-memory below a certain threshold, called *window*, which is typically at most a few hundreds KiloBytes. When the input size exceeds the window, the input has to be split into sufficiently small blocks which are then compressed separately. The optimal partitioning is not necessarily equal to the trivial partitioning into equal-width blocks (see Section 5.4 for an example involving binary strings).

In all the above applications, we are naturally interested in finding the best possible instantiation of the permuting and partitioning steps. While finding the best permutation is generally an hard optimization problem (see e.g. [11]) which can be addressed only by heuristics, the computation of the most compressible partition (once the permutation is given), referred as the *optimal partitioning* problem, can be treated in polynomial time using Dynamic Programming [11, 43]. However this (exact) solution is very expensive, as it requires cubic-time complexity, thus result-
ing unusable in practice even on files of few MBs. Although several alternatives have been proposed, none of them can guarantee any bound on the efficacy of the computed partition [10, 11], except on some restricted scenarios (such as compression boosting, see [29, 64]).

In the second part of this thesis we propose a substantial improvement over these methods by giving the first efficient approximation scheme for the optimal partitioning problem whose error is controlled by a user-defined parameter. Our method is applicable to a wide class of base compressors, including finite-order symbolwise encoders, such as Huffman, Arithmetic, or ppm-like encoder and the compressors based on the Burrows-Wheeler Transform (such as bzip2) (see Sections 6.5, 6.6).

I.2 Remapping to shortest-path computation

In this thesis we adopt a unified approach to recast both the problems above as shortest-path problems on a special class of weighted DAG, hereafter called the parsing graphs. This reduction is actually not new and dates back to Schuegraf [79], which originally studied it in the context of dictionary-based compression.

More specifically, given an input string $S$, Schuegraf defined a DAG $G(S)$, which has one node per character of $S$ plus a special node marking the end of the text and one directed edge per pair of nodes (i.e. characters) that delimits a potential phrase of the parsing of $S$. In the case of dictionary-based compression, the parsing graph contains for each dictionary phrase an edge going from the position where the phrase begins to the position immediately after the end of the phrase, while in the context of the optimal partitioning problem the parsing graph contains an edge connecting any pair of nodes, since there are no restraints on the blocks of the partition. As an example take the input string $S = ababab$ and the (static) dictionary $D = \{a, b, ab, bab\}$ of subsection I.1.1. The parsing graph $G(S)$ is reported in figure I.1:

![Figure I.1: The parsing graph for the example of subsection I.1.1](image)

The edges of $G(S)$ represent all possible steps the parsing can perform over $S$. More precisely there is a one-to-one correspondence between parsings of $S$ and paths in $G(S)$ that go from its first to its last node. Edges of these paths represent the phrases of the corresponding parsings; so by assigning a cost to each edge equal to the number of bits necessary to encode the corresponding phrase, the computation...
of a parsing minimizing the output size of the underlying compressor reduces to a Single-Source Shortest-Path (shortly SSSP) computation over $G(S)$.

The following is a well-known fact:

**Fact 1 ([20])** Single-Source Shortest-Path over a DAG with $n$ vertices and $m$ edges can be computed in $O(n + m)$ time.

The algorithm consists of scanning the sequence of vertices in topological order and relaxing the edges outgoing from the current vertex. However, this algorithm cannot be applied naively to $G(S)$. In fact, in all the applications of our interest the size of $G(S)$ can be quadratic in the worst case, thus making direct shortest-path computation theoretically inefficient and practically unfeasible, even on short inputs.

### I.3 Contributions: A Novel Pruning Based Approach

This thesis introduces novel techniques and algorithmic tools to significantly speed-up the shortest-path computation over the parsing graph. At the most general level, our approach consists of two big steps.

First of all we recognize that, although the original parsing graph can have quadratic size, a large fraction of its edges are not relevant for shortest-path computation. This allows us to restrict our attention on a pruned subgraph of the parsing graph, which has much smaller size ($O(n \log n)$ in many situations) yet preserves shortest-path distances.

Here comes into play the second step of our approach, which consists in devising algorithms for efficiently generating the pruned parsing graph starting from the input string, whose size is $n$. In this way, instead of computing the shortest-path over the original parsing graph, which would require quadratic time in the worst case, we compute the shortest-path directly over the pruned subgraph thus improving significantly the running time of previous methods from $O(n^2)$ to $O(n \log n)$. The overall approach is schematized in figure I.2.

We apply the pruning scheme summarized in figure I.2 to the two main questions posed in subsections I.1.1 and I.1.2.

Specifically, for the bit-optimal LZ77-parsing problem we make some reasonable assumption on the functions used to encode codewords (see Prop. 1), and identify a pruned subgraph of $G(S)$ that preserves the shortest-path distance from the first to the last node (Theorem 10). The size of this subgraph depends in general on the choice of the encoding function, and is $O(n \log n)$ for many encoders used in practice (see Section 3.4). For the optimal partitioning problem, we adopt a different pruning scheme providing a trade-off between accuracy and efficiency. More precisely, for any choice of the parameter $\epsilon$, we show that the number of edges in $G(S)$ can be
Figure I.2: A diagram showing the sequence of reductions in our approach

shrink to $O(n \log (1+\epsilon)n)$ without increasing the shortest-path distance from the first to the last node of $G(S)$ by more than a factor $1 + \epsilon$ (Theorem 25).

In both cases, the final step of the solution is to devise efficient algorithms for generating the pruned subgraph, sometimes denoted $\tilde{G}(S)$, and computing a shortest-path over it, which is the most technical part of our contribution. The main difficulty is that $\tilde{G}(S)$ cannot be generated by examining the edge-set of $G(S)$, since the latter has quadratic size. Moreover, we cannot even assume the entire graph $\tilde{G}(S)$ to be stored in memory during execution, as this would occupy space linear in the number of edges (using e.g., adjacency lists), that is an amount of space superlinear in the size of $S$. To overcome these difficulties and achieve an efficient solution working in optimal $O(n)$ space, we devise algorithmic tools to dynamically generate the edges of $\tilde{G}(S)$ directly from $S$ as they are inspected during the shortest-path computation.

As a result, we will be able to show that:

- A bit-optimal parser for the LZ77 dictionary can be computed in linear space and in time which is $O(n \log n)$ for most practical choices of the encoding functions used in to represent codewords (see Theorem 11 [35]), to which we dedicate the whole Chapter 3. We also provide experimental results, reported in Section 3.6, that show the effectiveness of our approach and its promising performance. In Chapter 4, we extend our study to other dictionary-construction schemes, such as the one used by the ACB compressor (Section 4.2), and by considering the related question of optimally encoding a fixed parsing (see Section 4.3 for details).

- There exists an approximation algorithm for the optimal partitioning problem which computes in $O(n \log_{1+\epsilon} n)$ time a partition of the permuted string whose compressed output-size is guaranteed to be no more than $(1 + \epsilon)$-worse the optimal one, where $\epsilon$ may be any positive constant (see Theorem 30 [34]). Our approach relies on the hypothesis that the output size of the base compressor
can be accurately estimated via entropy-based bounds. As discussed in Section 1.2, the empirical entropy can provide reasonable bounds on the performance of many compression algorithms. Then, in Section 6.6, we extend our technique to handle base compressors based on the Burrows-Wheeler Transform, such as bzip2, for which entropy-based estimates may result far from the real output-size of the compressor [29].

Our solution to the bit-optimal LZ77-parsing problem, as well as all the material of Chapter 3, have been published in [35]. The results of Section 4.3 also appeared in [35], while the content of Section 4.2 on the ACB compressor is new and still unpublished. Our results on the optimal partitioning problem, that include Theorem 30, are described in Chapter 4 which is essentially derived from [34].

I.4 Map of the Thesis

This thesis is organized in three parts:

- Part 1, Introductory Material (Introduction and Chapter 1). Chapter 1 briefly covers the basic tools and notions used throughout the rest of this thesis. In particular, Section 1.1 recalls the most fundamental data structures for strings, such as Tries, Suffix-Trees, Suffix-Arrays, and lcp-queries, while Section 1.2 discusses the important notion of empirical entropy as well as the most common classes of universal and statistical encoders.

- Part 2, Dictionary-Based Compression (Chapters 2 to 4). This part is dedicated to the question of bit-optimal parsing for dictionary-based compressors. Chapter 2 provides a general introduction to the dictionary-based compression, focusing in particular on the Lempel-Ziv family of compression algorithms, and the ACB compressor. Chapter 3 presents our solution to the problem of efficiently computing a bit-optimal parsing for LZ77, and reports some experimental results. Then, Chapter 4 extends this approach to the ACB compressor and the related question of computing the greedy LZ77-parsing having the smallest encoding.

- Part 3, Reorganizing Data for Compression (Chapters 5 and 6). Contains our results related to the optimal partitioning problem. In Chapter 5, we provide a short introduction to the PPC-paradigm and its main applications to data compression; after that, Chapter 6 describes our novel approximation algorithm for the optimal partitioning problem. This algorithm systematically yields new improved solutions for all the applications of the PPC-paradigm mentioned in I.1.2

In the final section, we will then report some of the main open problems and directions of research connected to our work.
I.5 Notations and Model of Computation

In what follows, we will always denote by $S$ the input string and with $\Sigma$ the input alphabet of size $\sigma$. When talking about algorithms on strings we distinguish the main cases of integer alphabet, where $\Sigma = [1, m]$ for some integer $m$, and that of ordered alphabet, where $\Sigma$ is given with a total ordering where any pair of symbols can be compared in constant time. We assume the standard RAM model of computations, and measure space complexity in memory words (unless otherwise specified).

The length of a string $x$ will be denoted by $|x|$, and in the rest of this thesis $n$ will always indicate the length of the input string $S$. We use $S[i]$ to denote the $i$th symbol of $S$ and $S[i : j]$ to denote the substring extending from the $i$th to the $j$th symbol of $S$ (extremes included). A string $S[1 : i]$, for any $i \leq n$, is called prefix of $S$. The suffix of $S$ starting at the $i$-th position is the string $S[i : n]$, and it is denoted by $S_i$ for brevity. The reverse of a string $x[1 : n]$ is the string $x^R = x[n]x[n-1]\ldots x[1]$.

We will write $x \prec y (x \succ y)$ to indicate that $x$ is lexicographically smaller (larger) than $y$.

The terms parsing and text partitioning are essentially synonyms and both denote a partitioning of $S$ into contiguous substrings called phrases. To respect the original terminology used in literature, we use the term parsing in the context of dictionary-based compression and partitioning when talking about the PPC-paradigm. The real difference among these two is that in the latter the choice of the phrases is arbitrary, while in the former it is conditioned on the dictionary-construction scheme.
Chapter 1

Preliminaries

This chapter recalls the most fundamental techniques and concepts used throughout the rest of this thesis, grouped in two sections. Section 1.1 surveys the most classic data structures for representing and indexing strings, such as tries, suffix arrays and suffix trees, motivated by their wide use in the algorithms developed in this thesis. Section 1.2 briefly recalls some basic notions of data compression that will be useful in the next two parts of this thesis. We will talk about Universal Codes (1.2.1), which are a key tool for Lempel-Ziv compression (Section 2.2) and play a fundamental role in the discussion of Part II. Then, we will introduce the main ideas of statistical encoders (1.2.3), and the Burrows-Wheeler Transform in (1.2.4), which are main components of several applications of the PPC-paradigm. We will also define the notion of empirical entropy (1.2.2) which provides compressive estimates that are crucial to achieve efficient solutions of the optimal partitioning problem.

1.1 Tries, Suffix Trees and Suffix Arrays

A trie over the alphabet $\Sigma$ is a rooted tree whose edges are labeled with strings over $\Sigma$ such that no two labels of edges leaving the same vertex start with the same symbol. A trie is called compact when all its internal vertices, except possibly the root, have at least two child.

Given a string $S$, the suffix tree [89] of $S$, denoted by $ST(S)$, is the compact trie over $\Sigma$ such that the concatenations of the edge labels along the paths from the root to the leaves are the suffixes of $S$. By appending to $S$ an auxiliary symbol $\$ we can always assume that no suffix of $S$ is a proper prefix of another suffix. An example of suffix tree is given in Figure 1.1. Each leaf of $ST(S)$ uniquely corresponds to the suffix of $S$ spelled out by the downward path connecting the root to that leaf.

Suffix trees admits a linear-space representation. It is easy to see that in $ST(S)$ there are $O(n)$ nodes and edges, where $n = |S|$. In fact $ST(S)$ contains $n$ leaves and, since all of its internal nodes are branching, the number of such nodes is at most $n − 1$. Each edge of $ST(S)$ is labeled with a substring $S[l, r]$, which can be
Figure 1.1: The suffix tree of the string $S = \text{bbabab}\$.

A suffix tree is compactly represented in constant space by the integer pair $(l, r)$. The structure and edge labels of $\mathcal{ST}(S)$ can therefore be represented in $O(n)$ space.

The time complexity of constructing the suffix tree has been an important research problem for long time. In particular, we mention the following results:

- Suffix trees can be constructed in $O(n \log \sigma)$ time and $O(n)$ space in the case of generic ordered alphabets [69, 85].
- For integer alphabets of polynomial size, there exists an optimal $O(n)$ time algorithm for constructing the suffix tree [25].

In a given tree, the **lowest common ancestor** of two nodes $u$ and $v$, denoted by $\text{lca}(u, v)$, is defined as the deepest shared ancestor of $u$ and $v$. An lca-query consists in computing the lowest common ancestor of two given nodes. It is often useful to extend the functionalities of the suffix tree to efficiently support lca-queries. This is a well-known data structuring problem, and it has been proven by many authors that:

**Theorem 2 ([2])**

A tree can be preprocessed in linear time to support lca-queries in constant time.

The main shortcoming of suffix trees is their high space consumption, which in practice can range from 9 to 11 times the size of the string to be indexed [44]. **Suffix arrays** provide a more space-efficient alternative to suffix trees which support all their main functionalities. The suffix array [65] is the lexicographically sorted array of all the suffixes of a string. More precisely, the suffix array $SA[1, n]$ of $S$, is a permutation of $[1, n]$ satisfying $S_{SA[1]} < S_{SA[2]} < \ldots < S_{SA[n]}$. Although the functionalities of suffix arrays are already included into suffix trees, in this thesis we will adopt suffix arrays in every context where the additional information supplied by suffix trees are inessential.

Clearly, the suffix array of a string can be constructed from the suffix tree in a straightforward way, but this approach would make suffix arrays redundant. A
naive solution which avoid suffix trees consists of using a general sorting algorithm or an algorithm for sorting strings. However, such algorithms have running time at least equal to the total length of the suffixes, which is $O(n^2)$.

Faster algorithms for constructing the suffix array appeared in [60, 65], which achieved $O(n \log n)$ time. More recently, several authors [49, 55, 57] independently discovered linear-time algorithms for suffix array construction in the case of integer alphabets of polynomial size. The case of generic ordered alphabet can always be reduced to the alphabet $[1, n]$, by first sorting the characters of $S$ and then replacing each symbol with its ranks in the ordering. The suffix array of the remapped string can be built in linear time and is exactly the same as the suffix array of $S$. These results are summarized by the following:

**Theorem 3** The suffix array of a string $S$ of length $n$ drawn from alphabet $\Sigma$ can be computed in $T_{\text{sort}}(n, \Sigma) + O(n)$ time, which denotes the complexity of sorting a generic $n$-elements subset of $\Sigma$.

In many questions considered in this thesis, it is required to efficiently compute the longest common prefix (shortly $\text{lcp}$) of two given suffixes of a string. There exist several strategies to efficiently perform $\text{lcp}$-computations (also referred as $\text{lcp}$-queries) which are based either on suffix trees or on suffix arrays. For suffix arrays, we can resort the following well-known result:

**Theorem 4 ([75])** The suffix array can be augmented with a linear-space data structure supporting $\text{lcp}$-computations for any pair of suffixes in constant time.

We will denote the $\text{lcp}$ of two suffixes $S_i$ and $S_j$ as $\text{lcp}(S_i, S_j)$. The basic component of the data structure is an array $\text{LCP}$, which stores the $\text{lcp}$ of any pair of lexicographically consecutive suffixes, i.e. $\text{LCP}[i] = \text{lcp}(S_{\text{SA}[i]}, S_{\text{SA}[i+1]})$. The array $\text{LCP}$ can be computed from the suffix-array with a simple linear-time algorithm [50]. It is easy to see that computing $\text{lcp}(S_i, S_j)$ reduces to determining the position of a minimum value in the subarray $\text{LCP}[i' : j']$, where $\text{SA}[i'] = i$ and $\text{SA}[j'] = j$. This kind of operation is called Range Minimum Query (RMQ) and several methods are known to preprocess an array in optimal linear time such that future RMQs can be computed in constant time using a small amount of additional information, for example [38] presented a solution requiring only $O(n)$ bits in addition to the array.

### 1.2 Basic Tools for Lossless Compression

This section contains the essential background on lossless data compression related to the next two parts of the thesis, including (semi-)static encoding, the notion of empirical entropy and the Burrows-Wheeler Transform.
1.2.1 Universal Codes

Encoding is a fundamental stage of many compression algorithms which consists of uniquely representing a sequence of integers as a binary sequence. In the simplest case the encoder makes use of a code, that is a mapping of the positive integers onto binary strings (codewords), in order to replace each value in input with its corresponding codeword. Codewords can be of variable-lengths as long as the resulting code is uniquely decodable. This is often enforced by requiring the prefix-free property, which states that no codeword can be a prefix of another codeword. Several codes have been proposed that achieve small average codeword-length whenever the frequencies of the input integers are monotonically distributed, so that smaller values occur more frequently than larger values. This kind of input often emerges in the intermediate steps of several data compression systems, such as the encoding step of the LZ77 algorithm (Section 2.2) or in the storage of inverted text indexes used in Web Search [92]. This codes are known in literature as Universal Codes or Variable-Length Codes. Compared to statistical encoders (such as Huffman or Arithmetic), universal codes are not affected by the overhead of transmitting a statistical model of the input, which can provide a significative advantage when the input sequence contains many distinct values. The following list reports some of the universal codes most commonly used in practice (see [54] for a more complete list):

- The unary encoding of an integer $n$ is simply a sequence of $n$ 1s followed by a 0. Unary encoding is rarely used as stand-alone tool and is often component of more complex codes. It achieves optimality when integer frequencies decrease exponentially as $p(i+1) \leq p(i)/2$.

- The Elias codes [54] are a recursively defined family of encoders. Each member is defined using the previous one starting from unary encoding as base element. The representation of an integer $x$ consists of a prefix-part, that specifies the bit-length of the standard binary representation of $x$ encoded with the previous encoder of the family, followed by the binary representation of $x$ with the most significative bit suppressed. The first useful Elias encoder is the well-known $\gamma$-code, which stores the prefix-part in unary. After $\gamma$, the next encoder of the family is $\delta$, which differs from $\gamma$ in that it encodes the prefix-part using $\gamma$ rather than unary. Further elements of the family can be generated by iterating the same scheme, but only $\gamma$ and $\delta$ are significative in practice.

- The Golomb codes [54] are a combination of unary and binary codes. These codes depend on the choice of a parameter $M$. An integer $x$ is encoded by computing a quotient $1 + ((x - 1) \div M)$ and a remainder $1 + ((x - 1) \mod M)$ and representing the former in unary and the latter in plain binary using at most $1 + \lfloor \log(M - 1) \rfloor$ bits. The concatenation of these two parts form the representation of $x$. It is possible to prove that Golomb codes achieve optimality over input integers that follow a geometric distribution of the form $P(i) = (1 - p)^{i-1}p$, for some $0 \leq p \leq 1$, when we set $M$ equal to $\log_{0.5} p$. 
• The *Fibonacci codes* [54] are a universal code based on the Zeckendorf’s theorem, which states that every positive integer can be decomposed in a unique way as the sum of one or more distinct Fibonacci numbers in such a way that the sum does not include any two adjacent Fibonacci numbers. The Fibonacci encoding of a value is a binary string such that the \( i \)-th digit (starting from left) is equal to 1 if \( F_i \) appear in the sum obtained from the Zeckendorf’s theorem, or 0 otherwise. Since it is not possible for both \( F_i \) and \( F_{i+1} \) to be part of the sum, the last two bits of this string must be 01. Thus appending a 1 bit to the end is sufficient to mark the end of each codeword.

• The *Byte-Aligned codes* are a simple coding mechanism that provides high decoding speed. Every integer \( x \) is represented by a codeword whose bit-length is a multiple of 8. Each codeword can be seen as a sequence of bytes where each byte is logically divided in two parts: a single-bit flag (usually the topmost bit) indicating whether this is the last byte of the codeword, and a remaining data part storing a chunk of seven bits taken from the binary representation of \( x \). The first seven bits of the binary representation of \( x \) are stored in the data part of the first byte, the next seven bits in the second byte and so on (if the length of the binary representation of \( x \) is not divisible by 7 the data part of the last byte is padded with 0s).

### 1.2.2 Empirical Entropy

The *empirical entropy* has been established by several papers [66, 31, 58] as a popular compressibility measure for strings. While the classic notion of Shannon’s entropy is a function of the source generating the input, the empirical entropy depends *only on the specific input string*. For this reason, the empirical entropy is naturally used to provide worst-case estimates on the output-size of compression algorithms.

For each \( c \in \Sigma \), we let \( n_c \) be the number of occurrences of \( c \) in the input string \( S \). The 0th order *empirical* entropy of \( S \) is defined as:

\[
H_0(S) = \frac{1}{|S|} \sum_{c \in \Sigma} n_c \log \frac{n}{n_c} \tag{1.1}
\]

The quantity \( \frac{n_c}{n} \) is called *empirical probability* of the symbol \( c \). The value \( |S|H_0(S) \) represents the output size of an ideal compressor which uses \( \log \frac{n}{n_c} \) bits for coding the symbol \( c \), and it is well-known that this is the maximum compression attainable by any compressor that encodes each symbol with a fixed codeword [92].

We can achieve a greater compression if the codeword we use for each symbol depends on the \( k \) symbols preceding it. The 0th order empirical entropy can be generalized to take into account the statistical correlations among consecutive symbols of the string.

For any string \( u \) of length \( k \), we denote by \( u_S \) the string of single symbols following the occurrences of \( u \) in \( S \), taken from left to right. For example, if \( S = \text{mississippi} \)
and \( u = si \), we have \( u_S = sp \) since the two occurrences of \( si \) in \( S \) are followed by the symbols \( s \) and \( p \), respectively. The \( k \)th order *empirical* entropy of \( S \) is defined as:

\[
H_k(S) = \frac{1}{|S|} \sum_{u \in \Sigma^k} |u_S| H_0(u_S). \quad (1.2)
\]

The value \( |S|H_k(S) \) represents a lower bound to the compression we can achieve using codes which depend on the \( k \) most recently seen symbols and, not surprisingly, we have that \( H_k(S) \geq H_{k+1}(S) \) for any \( k \geq 0 \).

### 1.2.3 Statistical Encoders

The idea of statistical encoding is to compactly represent a sequence of symbols by using a statistic model of the input. The goal of the model is to provide probability information about the input symbols. The modeling process may be *static*, if the model is unique for every input, *semi-static*, if the model is built via a preliminary scan of the input, or *adaptive* if the model is dynamically constructed as the input is processed.

The simplest form of statistical encoding is 0th order encoding. In this case the model simply consists of the empirical probability of each symbol in the input sequence. Two representative examples of these techniques are *Huffman coding* and *Arithmetic coding*. For simplicity, we will consider here only the semi-static variants of these methods. In this case, the input is preprocessed once to record the frequency of each symbol, then a code tailored on the input distribution is computed trying to assign shorter codewords to the most occurring symbols. These encoders are more general than universal codes because they assume nothing particular about the frequency distribution of the input, however they incur into the additional overhead of storing the model as part of the compressed output.

Knowing the frequency distribution, the Huffman algorithm [92] can be used to find the prefix-free code that minimizes the average bit-length of its codewords. This algorithm maintains a forest of binary trees representing disjoint subsets of the alphabet with weights equal to the sum of probabilities of the elements in the subset. Initially each symbol corresponds to a tree containing one single node. Then, at each iteration, the algorithm select two trees with lowest weights and merge them into a single tree whose weight is the sum of the two individual weights. The above process is repeated until a single tree remains, which is called the Huffman tree. The codeword of a particular symbol is obtained by traversing the tree from the root to that symbol and emitting a 0 for any left branch and a 1 for any right branch.

The Huffman codes are particularly inefficient in coding highly skewed distributions. For instance, an input sequence consisting of a single repeated value has null empirical entropy while its Huffman code still requires one bit per value.

0th order Arithmetic coding [77] remove this inefficiency by adopting codewords of “fractionary” bit-lengths. The basic idea of arithmetic coding is to represent
the entire input with an interval of real numbers between 0 and 1. The interval is initialized to \([0, 1]\) and incrementally refined as the input sequence is inspected. At each step, the encoder subdivides the current interval into as many subintervals as possible symbols in input, each one having size proportional to the frequency of the corresponding symbol. The current interval is then narrowed to the subinterval corresponding to the next symbol in input. After the last symbol has been processed, the encoder output enough bits to distinguish the final current interval from all other possible final intervals. In theory, the length of the arithmetic encoding achieve essentially the 0th order entropy of the input, but in practice the output is slightly increased by the approximation errors made when operating on finite-precision registers of real computers. The main disadvantage of arithmetic coding is that its implementations are generally slow for the frequent use of multiplications (and in some case divisions).

Huffman and Arithmetic coding are also called entropy-encoders because the output-length in bits can be accurately estimated using the 0th order empirical entropy of the input. If \(C\) is a 0th order entropy-encoder, and \(C(s)\) denotes the binary encoding of an input \(s\) through \(C\), these estimates take the general form \(|C(s)| \leq \lambda n H_0(s) + f(n, \sigma)\), where \(f(n, \sigma)\) is a function including the extra costs of encoding the source model and/or other inefficiencies of \(C\).

For Huffman codes \(\lambda = 1\) and \(f(n, \sigma) = \sigma \log \sigma + n\) which account the space for storing the Huffman tree plus the extra bits wasted by the encoder. A similar bound has been proven for Arithmetic coding where \(f(n, \sigma) = \sigma \log n + O(1)\) in the ideal case of infinite precision [47].

The distinctive feature of 0th order encoder is that the input symbols are independently encoded and no advantage is taken of the potential statistical correlations between symbols in a sequence. It is possible to obtain considerably better compression by using higher order models. A \(k\)th order model uses the \(k\) previous symbols (called context) to estimate the probability of the occurrence of some given next character. During the encoding process, this probability value is passed to a backend entropy coder (often an arithmetic encoder) to obtain the encoding. A famous modeler of this kind is Prediction by Partial Matching (ppm) and all its variants (see [17] for more details). The output length of a \(k\)th order statistical encoder can be studied in terms of the \(k\)th order empirical entropy, leading to estimate whose form is identical to that used for Huffman and Arithmetic coding except that \(H_0\) is replaced by \(H_k\) and the function \(f\) depends also on \(k\).

1.2.4 The Burrows-Wheeler Transform

The Burrows-Wheeler transform (BWT) was introduced in a technical report [12] as a reversible transformation of the input text that makes redundancy in the input more accessible to simple coding schemes.

The introduction of the Burrows-Wheeler transform marked a turning point in the field of data compression. This technique is currently at core of state-of-the-art
compressors, such as the famous bzip2. Besides its usage in pure compression, the BWT has other remarkable applications related to data compression: it is a fundamental ingredient in modern Compressed Full-Text Indexes [72]; it can be extended to compress and index different data types, such as XML files [32] and tables [86], and it enables the design of boosting techniques for textual compression (Section 5.3).

The BWT acts on the input text by permuting its symbols. Given the input string $S$, the transformed string $\text{BWT}(S)$ is obtained by forming a conceptual matrix $M$ whose rows are the cyclic shifts of the string $S$ sorted in lexicographic order; where a special symbol $\$ smaller than any other symbol in the alphabet is appended to the end of $S$. The string $\text{BWT}(S)$ is then defined by taking the last column of $M$ (see Figure 1.2).

It is immediate to notice that $\text{BWT}(S)$ can be directly computed from the suffix array of $S$ in $O(n)$ time.

Although not intuitive, the original string $S$ can be reobtained from $\text{BWT}(S)$, and the inverse transform is computable in optimal $O(n)$ time. The details of the decompression-side are not required for our future works, therefore we only sketch here the two main observations behind the inversion procedure: (1) given a symbol of the $\text{BWT}(S)$, the symbol that follows it in $S$ (cyclically) is located at the same offset in the first column of $M$ and (2) the reordering of the symbols provided by the $\text{BWT}$ is stable with respect to the first column of $M$, that is the $j$th occurrence of a symbol in the first column of $M$ is mapped to the $j$th occurrence of the same symbol in $\text{BWT}(S)$. Notice that the first column of $M$ is trivially recovered by lexicographically sorting the (occurrence of) symbols of $\text{BWT}(S)$. Starting from the position of $\$$ in $\text{BWT}(S)$, the inversion algorithm reconstruct $S$ by iteratively applying (1) to determine the symbol cyclically following the current one in $S$ and then (2) to find the position

Figure 1.2: Burrows-Wheeler Transform of the word mississippi. The last column of the conceptual matrix is the transformed string ipsm$pissii
where this symbol is mapped in $\text{BWT}(S)$. The above steps can be made in constant time per symbol, after a suitable preprocessing of the transformed string (see [12] for a detailed explanation).

The property that makes the $\text{BWT}$ a powerful text compression tool is the following: for each substring $u$ of $S$, the symbols following $u$ in $S$ are grouped together inside $\text{BWT}(S)$. This is a consequence of the fact that all the rows of the conceptual matrix starting with $u$ appear consecutively in the sorted matrix. Thus the effect of the $\text{BWT}$ is to group together symbols following the same context. When applied to input like natural language texts, the output is likely to be a locally homogeneous string, that consists of the concatenation of several substrings containing few distinct symbols.

To take advantage of this property, compressors based on the $\text{BWT}$ process the string $\text{BWT}(S)$ using a technique called Move-To-Front encoding [5] (MTF). MTF encodes each symbol with the number of distinct symbols encountered since its previous occurrence. To this end, MTF maintains a list of the symbols ordered by recency of occurrence; when the next symbol arrives the encoder outputs its current rank and moves it to the front of the list. Notice that MTF produces a string which has the same length as $\text{BWT}(S)$ and, if $\text{BWT}(S)$ is locally homogeneous, the string $\text{MTF}(\text{BWT}(S))$ will mainly consists of small integers. This makes $\text{MTF}(\text{BWT}(S))$ highly compressible by mean of simple statistical encoders like Huffman or Arithmetic coding, possibly preceded by the run-length encoding (RLE) of runs of equal integers.

The combination of the steps $\text{BWT, MTF and RLE}$ followed by a 0th order statistical encoder is the well-known Block-Sorting compression algorithm [27], which is actually implemented in $\text{bzip2}$. 
Part II

Dictionary-Based Compressors
Chapter 2

Background

Dictionary-based data compression is one of the main approaches to the problem of lossless data compression. This compression scheme squeezes an input string by replacing some of its substrings with (shorter) codewords which are actually pointers into a dictionary of phrases. This technique, also referred to as compression with textual substitution, has been introduced by the works of Ziv and Lempel [94, 95]. Although today there are alternative solutions to the problem of lossless data compression (e.g., Burrows-Wheeler compression and Prediction by Partial Matching [92]), dictionary-based compression is still widely used for its unique combination of compression power and compression/decompression speed. Indeed many (non-)commercial programs are currently based on it: consider for example the ZIP-utilities (gzip, zip, pkzip, arj, rar), the Unix compress tool and the compression of .gif images, to cite just a few.

The primary goal of a dictionary-based compressor is to achieve the maximum possible compression being as efficient as possible in terms of time and space consumption. In this context, textual parsing techniques are fundamental for achieving a competitive compression ratio. The aim of this chapter is to provide a brief introduction to dictionary-based compression. We begin by examining the two fundamental components of any dictionary-based method, namely the dictionary construction scheme and the parsing strategy. After that, we present two of the most important examples of dictionary-based compressors: the Lempel-Ziv family of compression algorithms, with some of their implementations, and the ACB compressor. Finally, Section 2.4 describes some of the main limitations of existing parsers and raises the bit-optimality question related to dictionary-based compressors, paving the way to our investigations of Chapters 3 and 4.

2.1 Anatomy of Dictionary-Based Compressors

Any dictionary-based compression method is made up of two relatively independent tasks. The first task is dictionary construction. A dictionary is a collection of strings,
called phrases, where each phrase of the collection is uniquely identified by a binary string, called codeword. The dictionary construction determines the set of phrases included in the dictionary as well as the codeword assigned to each phrase. The type of construction can be either static (in that the dictionary is constructed before the compression starts) or (semi-)dynamic (in that the dictionary is adaptively built upon the input string). The dictionary and the data structure employed for its representation largely affect the overall efficiency of the compressor, since most of the work performed during compression is due to look-up operations inside the dictionary.

The second task is parsing, which takes the input string $S$ and return a partition of $S$ into phrases, say $S = w_1 w_2 \ldots w_k$, where each $w_i$ is a dictionary phrase. The compressed output is then obtained by replacing each phrase with its corresponding codeword. In dictionary-based compression, the choice of the parsing strategy is determinant for the final compression performance, in fact the size of the compressed output is ultimately determined by the total bit-length of the codewords encoding each parsed phrase.

Altogether, dictionary construction and parsing must enforce the two fundamental requirements enforced by any compression method: generality, i.e. the compression algorithm must be applicable to every possible input string; and unique decompressibility, which implies the existence of a decompression algorithm which can uniquely restore the original text from the compressed representation.

### 2.1.1 Dictionary Construction Schemes

Dictionary construction methods can be classified into three basic families: static, semi-dynamic, and dynamic (some authors prefer to use the term adaptive instead of dynamic).

A static dictionary method uses the same dictionary for all input strings. The dictionary used must be available to both the compression algorithm and its corresponding decompression algorithm. It is easy to see that no static dictionary can achieve good performances for every possible input, thus such dictionaries are used only in specific applications where the input strings of interest contain many common phrases. Ideally the dictionary should contain the phrases that are more likely to occur in the input strings which are typically encountered in the application domain. Moreover, to guarantee the generality of the method, it must be possible to parse every input string into phrases of the dictionary.

In the semi-dynamic case, a dictionary $D$ specific for the string $S$ to be compressed is built by inspecting $S$ before compression and selecting some of its most frequently occurring substrings as phrases of $D$. To allow unique decompressibility, the dictionary $D$ must be passed to the decompressor (potentially encoded in compressed form) along with the compressed text. The dictionary construction scheme should aim at producing the “best” static dictionary for the text to be compressed, which is the one that minimizes the total size of the dictionary encoding and the
2.1. ANATOMY OF DICTIONARY-BASED COMPRESSORS

Dynamic dictionary methods are the most common in general applications. For instance, the well-known LZ77 and LZ78 compressors (described in Section 2.2), proposed by Lempel and Ziv in [94, 95], and all their variants [78], are interesting examples of dictionary-based compressors that rely on a dynamic dictionary construction scheme. A dynamic dictionary is adaptively built as the input text is compressed by inserting in the dictionary the substrings that are frequently observed in the input. In such schemes, a codeword that replaces a substring of the input can be seen as a pointer to another occurrence of the same substring. Most of the dynamic dictionary methods are unidirectional, in that all pointers have the same direction, i.e. they refer to a previous occurrence of the phrase in the text.

A very attractive feature of unidirectional methods is that compression and decompression can be both performed with a single-pass of the input. The importance of single-pass methods is mainly due to the fact that they usually admit on-line implementations which provide a short delay between reading a symbol and emitting the codeword representing the phrase containing that symbol; this is a fundamental requirement in many network applications.

2.1.2 Parsing strategies

Given a dictionary construction scheme, the task of parsing consists of splitting the input text into a sequence of dictionary phrases.

The most popular parsing method (for both static and dynamic dictionary) is greedy parsing. As the input string is compressed from left to right, the greedy parser determines at each step (and replaces with the corresponding codeword) the longest dictionary phrase that matches a prefix of the uncompressed suffix of the input string.

The greedy parsing strategy operates only on the basis of local information and does not require to look ahead or look back in the input. For this reason, a greedy parser can be implemented to work with a single pass and providing a limited delay between reading an input character and emitting the codeword that represents the phrase containing it, which makes it suitable for applications requiring on-line processing of the input. It has been proven that the greedy strategy achieves optimality in the number of phrases (i.e., it produces a parsing with the minimum number of phrases) when the dictionary is suffix-complete [19]. A dictionary $D$ is suffix-complete if, for every phrase in $D$, all its suffixes are also phrases in $D$; a concrete example of suffix-complete dictionary is the one used inside the LZ77 algorithm. In other remarkable cases, as such the LZ78 algorithm, the dictionary is prefix-complete; that is, for each phrase of $D$ all its prefixes are also phrases of $D$. For such dictionaries, compression with greedy parsing can be far from optimal: for any sufficiently large integer $m$, there exists a string that can be parsed to $O(m)$ phrases while the greedy strategy parses it in $O(m^{3/2})$ words [68]. However, by a slight modification to the greedy rule, it is possible to obtain optimality for prefix-complete dictionaries.
This variant of greedy parsing, called flexible greedy parsing \cite{68, 67}, is essentially a greedy parsing with one-step lookahead which achieves optimality in the number of phrases on any input when used with a prefix-complete dictionary.

2.2 The Lempel-Ziv Family

Most of the popularity of dictionary-based compressor is due to the Lempel-Ziv 77 and 78 algorithms (denoted LZ77 and LZ78). Since their introduction in the late ’70s, these methods have widespread as a general tool for lossless data compression. Moreover, because of their deep mathematical properties, Lempel-Ziv parsing strategies have also found applications in other algorithmic domains, being employed in the design of compressed text indexes \cite{73}, in universal clustering \cite{16} or classification tools \cite{93}, in designing optimal pre-fetching mechanisms \cite{87}, in streaming or on-the-fly compression applications \cite{21, 53, 39}, in accelerating edit-distance computation \cite{45}, and in sequence alignment \cite{23}. Lempel-Ziv compressors are dynamic-dictionary methods that compress a string with a single left-to-right scan: at each step a prefix of the uncompressed suffix is selected as next phrase of the parsing and, at the same time, the dictionary is updated with new phrases taken from the already compressed prefix of the text. Well-engineered implementations of the Lempel-Ziv scheme can offer an impressive efficiency/compression tradeoff, and this is the main reason behind its large success in applications.

2.2.1 The LZ77 Algorithm

Before presenting the original version of the LZ77 algorithm, we describe a simpler variant by Storer and Szymanski \cite{82}, denoted LZSS.

The parsing method employed by LZSS is the greedy one. Assume that the compressor has already parsed the prefix $S[1 \ldots m]$ of the input string $S$ into phrases $w_1, w_2, \ldots, w_{i-1}$, that is, $S[1 \ldots m] = w_1 w_2 \ldots w_{i-1}$. The dictionary, at each step, consists of two types of phrases: (1), all the substrings of $S$ which start in the already compressed prefix $S[1 \ldots m]$ and (2), all distinct symbols of the alphabet, also referred as literals. The next phrase selected by the greedy rule is the longest prefix of the uncompressed suffix of $S$ matching a dictionary phrase. Thus $w_i$ is the longest prefix of $S[m+1 \ldots n]$ satisfying at least one of the following condition:

1. $w_i$ consists of the single symbol $S[m + 1]$.
2. $w_i$ matches a substring of $S$ starting in $S[1 \ldots m]$.

Allowing an arbitrarily large dictionary may lead to storage issues, therefore in practice the dictionary is usually limited to the substrings of $S$ starting in the last $M$ compressed symbols, where the value of $M$, called window size, is smaller than
2.2. THE LEMPEL-ZIV FAMILY

the size of the main memory. The scheme described above coincides with the case of unbounded window, where $M = \infty$.

In the LZSS dictionary, each codeword is represented by a pair of integers $(d, l)$, whose meaning is defined as follows: if the phrase is of type (1), then $d$ expresses the relative offset of the phrase within the prefix (i.e., its distance from the current parsing position) and $l$ is the length of the phrase; if the phrase is of type (2), and consists of the single symbol $c$, the corresponding codeword is the pair $(0, c)$. In the original LZ77 algorithm, codewords also include a third component, which contains the symbol following the phrase in the uncompressed text, this symbol is then ignored when parsing the remaining part of the string. Both the components of each codeword are then encoded in binary either with a fixed number of bits (which generally achieves poor compression) or using a variable-length encoder, such as an Elias universal encoders or a statistical 0th order encoders (see Section 1.2). The final output of the compressor is obtained by concatenating the codewords representing each phrase of the parsing.

We define an LZ77-parsing as a parsing of the input string consisting of phrases taken from the dictionary built by LZ77. In particular, the sequence of phrase in which $S$ is parsed by LZSS is an LZ77-parsing, and we will call it the greedy LZ77-parsing of $S$ (also known as Lempel-Ziv Factorization in some literature). As an example, let us consider the input string $S = ababaaaababbaba$. The greedy LZ77-parsing of $S$ is the sequence of phrases: $a, b, aba, aaa, bab, baba$, and the respective codewords assigned by LZSS are: $(0, a), (0, b), (2, 3), (1, 3), (7, 3), (10, 4)$. Notice that a phrase and its copy may overlap: this happen, for instance, to phrase $aaa$ in the above parsing and its copy located at distance 1.

Due to its importance for LZ77-based compressors and beyond, several research efforts have aimed at developing efficient algorithms for computing the greedy LZ77-parsing of a string. In the case of unbounded window, the greedy LZ77-parsing of a string $S$ can be easily obtained in linear time by a bottom-up visit of the suffix tree of $S$ (see, for instance, [1]). Later, Puglisi et al. [15] showed how to reduce the constant in the space complexity of this method by using the suffix array instead of the suffix tree. In the more general case of bounded window the best current bound was obtained by Fiala and Green [37], which modified the suffix tree construction algorithm to produce the greedy LZ77-parsing in $O(n \log \sigma)$ time and $O(M)$ space (where $M$ as above denotes the size of the window). These algorithms are far from being practical, thus real compressors based on the LZ77 algorithm (such as gzip) generally resort simpler parsing methods that achieve reasonable efficiency but violate the greedy selection rule (see subsection 2.2.3).

Finally, we remark that one of the most valuable aspects of the LZ77 scheme is the simplicity of the decompression part. The compressed string can be decompressed from left to right by inspecting the sequence of codewords and copying the corresponding phrase from the decompressed prefix. This algorithm admits a small and efficient implementation, which works on-line and requires no extra space beside the output buffer. This is particularly useful for applications of the type...
compress-once/decompress-many, in which computational power and memory of the decompressing party (such as in mobile communications) are much lower than those of the sending party.

2.2.2 The LZ78 Algorithm

Although it is now known that a greedy LZ77-parsing can be computed in optimal linear time, the first algorithmic realizations of this method required $O(n^2)$ time, where $n$ is the number of symbols in the input. One year later, Ziv and Lempel [95] proposed an alternative method, the Lempel-Ziv 78 algorithm (denoted LZ78), having similar asymptotic properties (Section 2.4) but a very straightforward $O(n)$ time implementation. The popularity of the LZ78 algorithm is mainly due to a variant proposed by Welch [90], which is known as the LZW algorithm. Let us first describe LZW, and then talk about its differences with LZ78. The LZW compressor is based on greedy parsing, like the LZ77 algorithm, but adopts a different dictionary construction scheme. The dictionary initially contains each single-symbol substring. After each parsing step, the parsed phrase is concatenated with the first symbol of the uncompressed portion of the input and is inserted in the dictionary as a new phrase. This phrase is assigned $D + 1$ as a codeword (where $D$ is the number of phrases currently in the dictionary). At any given step of the algorithm each codeword is encoded with $\lceil \log_2 D \rceil$ bits.

The dictionary construction satisfies prefix-completeness. The simplest way to maintain the LZ78 dictionary is by storing its phrases in a trie (Section 1.1). At each step, the parsing algorithm can search the longest matching phrase (and retrieve the corresponding codeword) with a simple top-down traversal of the trie, spending time proportional to the length of the next parsed phrase. Summing up over each phrase, the total running time of the parser is $O(n)$.

For example, considering again the input string $S = ababaaaababbaba$, LZW parses $S$ as $a, b, ab, a, aa, aba, b, ba, ba$ and encodes the phrases with the sequence of codewords 1, 2, 3, 1, 6, 5, 2, 4, 4.

The original LZ78 algorithm again uses a variant of greedy parsing. At each step of the algorithm, the longest prefix of the uncompressed portion of the input is parsed and replaced with a codeword; the following symbol is skipped uncompressed to iteratively continue the compression process. The dictionary at a given step includes all parsed phrases concatenated with the single characters following them, which are left uncompressed.

2.2.3 Compression Tools Based on LZ77/78 Algorithms

The LZ77 and LZ78 algorithms have been implemented in a number of data compression programs, some of the most important are the gzip compressor, the UNIX compress utility and the .gif image format. In what follow we give a short insight in these tools, which are essentially slight variants of the Lempel-Ziv scheme:
• The gzip compression program is often regarded as one of the best compromises between compression ratio and time efficiency. The gzip algorithm is a modified LZSS with a window size $M = 32K$. The main data structure used by gzip is an hash-table storing the $M$ text locations falling in the window. The index associated to a location is computed from the string of length 3 starting in that location. Locations mapping to the same bucket of the table are kept in a linked-list sorted by last access time (the top of the list stores the most recently accessed location). When searching the next phrase, the parser determines the index of the current position and scan the corresponding list (if not empty) to determine the location giving the best match with the currently uncompressed suffix. To avoid a worst-case situation, very long hash chains are arbitrarily truncated at a certain length, determined by a runtime option. This implies that the parsed phrase is not necessarily the longest possible match. The gzip program is also capable of performing greedy parsing with one-step lookahead and return not the longest phrase but one which is shorter if the next phrase obtained is composed of a single character. The distance and length components of each codeword are then compressed with two separate Huffman trees.

• The compress program is an implementation of the LZW compression method for UNIX systems. The dictionary initially contains every symbol of the alphabet. At any step of the execution of the algorithm each codeword is encoded by $\lceil \log_2(|D| + 1) \rceil$ bits, where $|D|$ is the size of the dictionary. The size of the dictionary is not allowed to grow more than a user-defined constant. Once the dictionary reaches the maximum size, no more phrases are added and the compression rate is sampled at regular interval, whenever compression performance decreases more than a given threshold, the dictionary is reset to the initial state.

• The .gif image format was proposed in 1987 by CompuServe and is currently one of the most common graphic format in the World Wide Web. The pixel of the image are scanned horizontally, from top-left to low-right corner, and encoded with LZW algorithm. The dictionary has a maximum size set to 4096 entries and it is initialized with $2^r$ entries corresponding to all possible combinations of $r$ bits, where $r$ is the minimum number of bits required to represent any pixel in the image, plus two special entries for the clear code and the end of image. This compression method is particularly useful in lossless compression of synthetic (i.e., computer-generated) images which consist mainly of smooth regions of same color intensity values.
2.3 The ACB compressor

The name ACB stands for associative coder of Buyanovsky, a text compression method introduced by George Buyanovsky in 1994. Not many details are available about its actual implementation, the original documentation is outdated and in Russian [13] while an informal interpretation in English can be found in [9]. At the time of its introduction, ACB was reported as one of the most powerful text compressors known [54]. The only documented attempt to classify ACB in the taxonomy of compression algorithms was made by [28], which proposed to consider ACB a symbol-ranking (or better a phrase-ranking) compressor. We will instead regard ACB as a dynamic-dictionary compressor. The reason for adopting this perspective is that it will allow us to extend our results of Chapter 3 to the ACB compressor (see Chapter 4).

In what follows, sticking to the notation of Section I.5, we denote by $x^R$ the reverse of string $x$. The dictionary used by the ACB compressor is essentially an LZ77 dictionary with a different codeword assignment scheme. As in the LZ77 case, the dictionary, at each step, consists of all single-symbols phrase, called literals, and all the substrings of the text which start in the already compressed prefix. Recall that in LZ77 each codeword is represented by a pair of integers where the first component gives the distance of the phrase from the current position, and the second component is the length of the phrase. ACB adopts the same format to specify codewords, what makes the difference is the definition of “distance”.

Given a substring of the input string $S$ starting in position $i$, we define its associated context as the reversed prefix of $S$ ending in position $i - 1$; i.e., the string $S[1 : i - 1]^R$. We refer the portion of the text following a given context as the content of that context.

The compressor maintains the lexicographically sorted sequence of contexts associated to each phrase of the dictionary. Assume that the parsing is currently in position $m + 1$ and thus the prefix $S[1 : m]$ has been already compressed. We will call current context the string $S[1 : m]^R$. The compressor searches the location where the current context would be inserted in the lexicographically sorted sequence of dictionary contexts, the offset of this location is called anchor. Each phrase $x$ of the dictionary is then represented by the codeword $(\delta, \lambda)$, where:

- $\delta$ is the difference between the anchor and the position in lexicographic order of the context associated to $x$ (or $\delta = 0$ if $x$ is a literal).
- $\lambda$ is the length of $x$ (or $\lambda = c$ if $x$ is a literal consisting of the symbol $c$).

Notice that the value of $\delta$ is obtained by computing a difference and it can be negative. The components of each codeword are then encoded in binary by mean of an universal integer encoder or a 0-order statistical encoder (taking care of the sign of $\delta$).

Let us refer to the example of Figure 2.1. Assume that the input string is $S = ababaaaabababa$ and that the prefix $ababaaa$ of the string has been already
### Figure 2.1: State of the dictionary for the input string $S = \textbf{ababaaabababbaba}$ after that prefix $\textbf{ababaaa}$ has been parsed. The table reports the lexicographically sorted sequence of contexts (with symbols written in their original order) and their respective contents. Notice that the current context falls in the second position of the lexicographic order. Assume that from the uncompressed suffix we select $\textbf{aba}$ (in bold and underlined in Figure 2.1) as the next phrase of the parsing. This phrase is copied from the dictionary entry preceded by context $\textbf{ab}$, which is the sixth one in the lexicographic order. As a consequence, $\textbf{aba}$ is encoded by the codeword $(\delta = 6 - 1 = 5, \lambda = 3)$.

This way of defining codewords relies on the hypothesis that contents are good “predictors” of the associated contexts, which is often reasonable in natural language text. Therefore we expect the current context and the context of the phrase to appear close in lexicographic order, since both their contents are prefixed by the same phrase. This means that the values of $\delta$ produced by the above scheme should be small in average and therefore well-compressible.

### 2.4 Compression Issues

#### 2.4.1 Entropy Bounds

Given the practical relevance of Lempel-Ziv algorithms, many efforts have been done to analyze their performance. Several results have been obtained assuming that the input string is generated by a stationary, ergodic source. In this setting, it has been shown that both LZ77 and LZ78 are optimal, in the sense that their compression rate approaches the entropy of the source [80, 95].

Although these results have great theoretical value, they can only provide an average case analysis of the compression ratio.

In order to get worst-case estimates that hold for any input string, the authors of [58] related the performances of Lempel-Ziv algorithms to the $k$th order empirical contents.
entropy of the input (Section 1.2.2). They showed (Th.3.2 and Th.4.1 of [58]) that the compression ratios of both LZ77 and LZ78 on an input string $S$ differ from the $k$th order empirical entropy of $S$ by a quantity that vanishes as $|S|$ grows to infinity. More precisely, they proved that the following estimate hold both in the case of LZ77 and of LZ78:

$$\frac{|C(S)|}{|S|} \leq H_k(S) + \Theta\left(\frac{\log |S|}{\log \log |S|} + \frac{k}{\log |S|}\right) \quad (2.1)$$

Here $C(S)$ denotes the output of the compressor $C$ applied to $S$, where $C$ may be either LZ77 or LZ78.

2.4.2 Greedy vs Non-Greedy

The ultimate goal of parsing is to partition the input string into a sequence of phrases whose corresponding codewords have the smallest total bit-length. We say that a parsing algorithm is bit-optimal when it achieves this goal. The question of constructing bit-optimal parsers is of fundamental interest, since a bit-optimal parser achieves the best compression ratio attainable under a given dictionary-construction scheme.

It is natural to ask whether parsing algorithms deployed by existing dictionary-based compressor achieve bit-optimality or not. We have seen that both LZ77 and LZ78 use the greedy parsing strategy which consists in selecting, at each step, the longest prefix of the unparsed suffix which is in the dictionary. In the LZ78 case (and in the LZW variant), any parsing strategy that is optimal in the number of phrase (like the Flexible-Parsing mentioned in Section 2.1) is also bit-optimal, because codewords are encoded with a fixed number of bits.

In the LZ77 case, the dictionary construction method is suffix-complete (all suffixes of a phrase are by definition phrases), therefore greedy parsing emits the minimum possible number of phrases (Section 2.1). Thus, if codewords are encoded with a fixed number of bits, greedy parsing provides the best compression possible for the LZ77 dictionary construction. However, in order to achieve competitive compression, the existing implementations of LZ77 (such as gzip), generally resort to variable-length encoders for representing codewords; in this situation we will prove that greedy parsing is no longer bit-optimal (Section 3.3).

Starting from this premise, Chapter 3 and 4 address the main issues related to bit-optimal parsing, focusing in particular on the Lempel-Ziv compression scheme and ACB. We will propose new parsing algorithms which achieve bit-optimality in the compressed output size by taking efficient/optimal time and optimal space.
Chapter 3

Efficient Bit-Optimal Lempel-Ziv Compression

3.1 Introduction

In Chapter 2 we introduced dictionary-based compression, and discussed in details two well-known methods of this class, the LZ77 and LZ78 algorithms, and some of their practical variants (Section 2.2).

Classically, the LZ77-parser adopts the greedy parsing strategy, which achieves optimality with respect to the number of phrases in which the input can be parsed (subsection 2.1.2). Of course, the number of parsed phrases influences the compression ratio and, indeed, various authors proved that greedy parsing achieves asymptotically the (empirical) entropy of the source generating the input string (Section 2.4). But these fundamental results have not yet closed the problem of optimally compressing a string because the optimality in the number of parsed phrases is not necessarily equal to the optimality in the number of bits output by the final compressor on each individual input string $S$. In fact, if the phrases are compressed via an equal-length encoder, like in [58, 78, 94], then the produced output is bit optimal. But if one aims for higher compression, variable-length encoders should be taken into account (see e.g. [92, 26], and the discussion on gzip in Section 2.2), and in this situation the greedy-parsing scheme is no longer optimal in terms of the number of bits output by the final compressor (see Section 3.3).

Many authors have addressed the issue of bit-optimality in LZ77-parsing, but their solutions are either inefficient [79], in that they take $\Theta(n^2)$ worst-case time and space, or they are approximate [51], or they rely on heuristics [56, 81, 7, 18] which do not provide any guarantee on the time/space performance of the compression process. This is the reason why Rajpoot and Sahinalp stated in [76, pag. 159] that “We are not aware of any on-line or off-line parsing scheme that achieves optimality when the LZ77-dictionary is in use under any constraint on the codewords other than being of equal length”. In this chapter we investigate this question by considering
a general class of variable-length codeword encodings which are typically used in data compression (e.g. gzip) and in the design of search engines and compressed indexes [73, 78, 92]. Our final result is a time efficient (possibly, optimal) and space optimal solution for the problem of computing the bit-optimal (LZ77-)parsing of a string (Theorem 11).

The outline of the chapter is the following. In Section 3.2 we formalize the bit-optimal LZ77-parsing problem and state the assumptions on the encoding functions required by our solution. Section 3.3 proposes an infinite class of strings for which the compression gap between the parsing based on the greedy-rule and the fully bit-optimal parsing is unbounded (Lemma 5). Starting from this negative result, we design our novel bit-optimal parsing strategy which is efficient (possibly optimal) both in time and space (Theorem 11). Following the scheme outlined in Section I.3, we model the search for a bit-optimal parsing of an input string $S$ as a single-source shortest path problem (shortly, SSSP) on a weighted DAG $G(S)$ consisting of $n$ nodes, one per character of $S$, and $e$ edges, one per possible parsing step (Section 3.4). We prove new combinatorial properties for this SSSP problem and show that the computation of the SSSP in $G(S)$ can be restricted onto a subgraph $\tilde{G}(S)$ whose structure depends on the integer-encoding functions adopted to encode the LZ77 codewords, and whose size is provably smaller than the complete graph $G(S)$ (see Theorem 10). Finally, in Section 3.5 we design an algorithm that solves the SSSP on the subgraph $\tilde{G}(S)$ without materializing it all at once, but creating and exploring its edges on-the-fly in optimal $O(1)$ amortized time per edge and $O(n)$ optimal space overall. As a result, our novel LZ77-compressor achieves bit-optimality in $O(n)$ optimal working space and in time proportional to $|\tilde{G}(S)|$ (hence, it is optimal in its size). The latter is $O(n \log n)$ for a large class of integer encoders, like Elias and Fibonacci codes (Section 1.2), and it is optimal $O(n)$ for (most of) the encodings used by gzip [46]. This is the first result providing a positive answer to Rajpoot-Sahinalp’s question above! We will complement our study with some experimental comparisons among our novel LZ77-based compressor and some well known compression tools, such as gzip and bzip2, thus showing the effectiveness of our approach and its promising performance (Table 3.6).

### 3.2 Problem Definition

We have already introduced the notion of LZ77-parsing in Section 2.2. Recall that an LZ77-parsing of a string $S$ is a parsing of string $S$ of the form $S = w_1 w_2 \ldots w_k$ satisfying the property that each phrase $w_i$ is either a single symbol or there exists a copy of $w_i$ starting in the prefix $w_1 w_2 \ldots w_{i-1}$. In other words, the LZ77-parsings of a string $S$ are all the possible parsings of $S$ consisting of phrases selected from the dictionary built by LZ77 over $S$.

The bit-optimal LZ77-parsing problem asks to find the LZ77-parsing of the input string having the minimum possible encoding size in bits, which is the one that
produces the shortest compressed output. To turn this informal statement into a precise definition it is necessary to clarify how an LZ77-parsing is encoded into a binary string.

The LZ77-encoding scheme depends on the choice of two variable-length integer encoders, respectively denoted by \( f \) and \( g \). We specifically focus on the LZSS scheme (described in Section 2.2), where the codeword encoding phrase \( w_i \) is represented by the pair of integers \((d_i, \ell_i)\), such that \( d_i \) is the relative offset of the copied phrase \( w_1 \cdots w_{i-1} \) and \( \ell_i \) is its length \(|w_i|\) (or \( d_i = 0 \) and \( \ell_i = c \) if the phrase is a literal consisting of the single symbol \( c \)). Once the parsing has been computed and its phrases replaced by their corresponding pairs, we encode each pair \((d_i, \ell_i)\) with the binary string \( f(d_i)g(\ell_i) \). The concatenation of these binary strings finally gives the binary encoding of the parsing, whose size in bits is therefore equal to \( \sum_{i=1}^{k} |f(d_i)| + |g(\ell_i)| \).

The bit-optimal \((f, g)\)-LZ77-parsing (or bit-optimal parsing, for brevity) is the LZ77-parsing that minimizes this quantity.

In order to study and design bit-optimal parsing schemes we need to make some reasonable assumptions on the integer encoders \( f \) and \( g \), which are satisfied by most of the encoders used in practice. Let \( e \) be an integer-encoding function that maps any integer \( x \in [n] \) into a (bit-)codeword \( e(x) \) whose length is denoted by \(|e(x)|\) bits. We consider variable-length encodings which use longer codewords for greater integers:

**Property 1 (Increasing Cost Property)** For any \( x, y \in [n] \) it is \( x \leq y \) iff \(|e(x)| \leq |e(y)|\).

This property is satisfied by most of the integer encoders adopted in most of the applications of data compression, as for instance the equal-length encoder, and all the classes of universal codes discussed in Section 1.2.

### 3.3 Sub-Optimality of Greedy Parsing

We have already noticed in Section 3.1 that the greedy strategy used by LZ77 is not necessarily bit-optimal, so we will hereafter use \( \text{OPT}_{f,g}(S) \) and \( \text{LZ}_{f,g}(S) \) to denote respectively the binary encodings of the bit-optimal \((f, g)\)-LZ77-parsing and of the greedy LZ77-parsing of \( S \).

Of course \(|\text{LZ}_{f,g}(S)| \geq |\text{OPT}_{f,g}(S)|\), but this does not provide us with any estimate of how much worse the greedy parsing can be with respect to the bit-optimal one. In what follows we identify an infinite family of strings \( S \) for which \( \frac{|\text{LZ}_{f,g}(S)|}{|\text{OPT}_{f,g}(S)|} = \Omega\left(\frac{\log n}{\log \log n}\right) \), so the gap may be asymptotically unbounded thus stressing the need for an \((f, g)\)-optimal parser, as requested by [76].

Our argument holds for any choice of \( f \) and \( g \) from the family of encoding functions that represent an integer \( x \) with a bit string of size \( \Theta(\log x) \) bits (thus the well-known Elias’ and Fibonacci’s coders belong to this family). Taking inspiration
from the proof of Lemma 4.2 in [58], we consider the infinite family of strings \(S_l = ba^l c^2 ba ba^2 ba^3 \ldots ba^l\), parameterized in the positive value \(l\). The greedy LZ77-parser partitions \(S_l\) as: \((b) (a) (a^{l-1}) (c) (c^2a^{-1}) (ba) (ba^2) (ba^3) \ldots (ba^l)\), where the symbols forming a parsed phrase have been delimited within a pair of brackets. Thus it copies the latest \(l\) phrases from the beginning of \(S_l\) and takes at least \(l |f(2^l)| = \Theta(l^2)\) bits.

A more parsimonious parser selects the copy of \(ba^{i-1}\) (with \(i > 1\)) from its immediately previous occurrence thus parsing \(S_l\) as: \((b) (a) (a^{l-1}) (c) (c^2a^{-1}) (b) (a) (ba) (ba^2) (a) \ldots (ba^{i-1}) (a)\). Hence the encoding of this parsing, called \(r_{OPT}(S_l)\), takes \(|g(2^l)| + |g(l)| + \sum_{i=2}^{l}(|f(i)| + |g(i)| + f(0)) + O(l) = O(l \log l)\) bits.

**Lemma 5** There exists an infinite family of strings such that, for any of its elements \(S\), it is \(|LZ_{f,g}(S)| \geq \Theta(\log |S|/\log \log |S|) \cdot |OPT_{f,g}(S)|\).

**Proof:** Since \(|OPT(S_l)| \leq |r_{OPT}(S_l)|\), we can conclude that:

\[
\frac{|LZ_{f,g}(S)|}{|OPT_{f,g}(S)|} = \frac{|LZ_{f,g}(S)|}{|r_{OPT}(S)|} \geq \Theta\left(\frac{l}{\log l}\right)
\]

Since \(|S_l| = 2^l + l^2 - O(l)\), we have that \(l = \Theta(\log |S_l|)\) for sufficiently long strings.

The experimental results reported in Table 3.6 show that this gap is not negligible in practice too, just look at the entries **Fixed-LZ77** and **BitOptimal-LZ77**. Additionally we can prove that this lower bound is tight up to a \(\log \log |S|\) multiplicative factor, by easily extending to the LZ77-dictionary (which is dynamic), a result proved in [52] for static dictionaries. Precisely, it holds that:

\[
\frac{|LZ_{f,g}(S)|}{|OPT_{f,g}(S)|} \leq \frac{|f(|S|)|+|g(|S|)|}{|f(0)|+|g(0)|},
\]

which is upper bounded by \(O(\log |S|)\) because \(|f(|S|)| = |g(|S|)| = \Theta(\log |S|)\) and \(|f(0)| = |g(0)| = O(1)\). To see this, let us assume that \(LZ_{f,g}(S)\) and \(OPT_{f,g}(S)\) are formed by \(\ell_{iz}\) and \(\ell_{opt}\) phrases respectively. Of course, \(\ell_{iz} \leq \ell_{opt}\) because the greedy parsing is optimal with respect to the number of parsed phrases for \(S\). We then assume the worst-case scenario in which every phrase is encoded by \(LZ_{f,g}(S)\) with the longest encoding (namely, \(f(|S|)\) and \(|g(|S|)|\) bits each) while \(OPT_{f,g}(S)\) uses the shortest one (namely, \(|f(0)|\) and \(|g(0)|\) bits each). Therefore, we have:

\[
\frac{|LZ_{f,g}(S)|}{|OPT_{f,g}(S)|} \leq \frac{\ell_{iz}(|f(|S|)|+|g(|S|)|)}{\ell_{opt}(|f(0)|+|g(0)|)} \leq \frac{|f(|S|)|+|g(|S|)|}{|f(0)|+|g(0)|} = \Theta(\log |S|)
\]

that proves the claim above.

### 3.4 Pruned Parsing Graph

Following the scheme outlined in Section I.3, we model the design of a bit-optimal LZ77-parsing strategy for a string \(S\) as a Single-Source Shortest Path problem (shortly, SSSP-problem) on a weighted DAG \(G(S)\) defined as follows. Graph \(G(S) = \)
3.4. PRUNED PARSING GRAPH

\((V, E)\) has one vertex per symbol of \(S\) plus a dummy vertex \(v_{n+1}\), and its edge set \(E\) is defined so that \((v_i, v_j) \in E\) if \((1) j = i + 1\) or \((2)\) the substring \(S[i : j - 1]\) occurs in \(S\) starting from a (previous) position \(p < i\). Clearly \(i < j\) and thus \(G(S)\) is a DAG. Every edge \((v_i, v_j)\) is labeled with the pair \(\langle d_{i,j}, \ell_{i,j}\rangle\) which is set to \(\langle 0, S[i]\rangle\) in case (1), or it is set to \(\langle p - i, j - i\rangle\) in case (2). The second case corresponds to copying a phrase longer than one single character.

It is easy to see that the edges outgoing from \(v_i\) denote all possible parsing steps that can be taken by any parsing strategy which uses a LZ77-dictionary. Hence, there exists a correspondence between paths from \(v_1\) to \(v_{n+1}\) in \(G(S)\) and LZ77-parsings of the whole string \(S\). We weight every edge \((v_i, v_j) \in E\) with an integer \(c(v_i, v_j) = |f(d_{i,j})| + |g(\ell_{i,j})|\), which accounts for the cost of encoding its label (phrase) via the encoding functions \(f\) and \(g\), and conventionally assume that \(c(v_i, v_j) = +\infty\) if the edge \((v_i, v_j)\) does not exist. This weighting of edges makes sure that the length in bits of the encoded parsing is equal to the cost of the corresponding weighted path in \(G(S)\).

The problem of determining the bit-optimal LZ77-parsing is thus reduced to computing the shortest path from \(v_1\) to \(v_{n+1}\) in \(G(S)\). However, the size of this graph is \(\Theta(n^2)\) in the worst case (take e.g. \(S = a^n\) [79, 51]) making direct SSSP computation over it very inefficient and actually unfeasible in practice even for strings of few Megabytes. In what follows we show that the computation of the SSSP can be restricted to a subgraph of \(G(S)\) whose size depends on the choice of \(f\) and \(g\) satisfying Property 1, and is \(O(n \log n)\) for most known integer-encoding functions.

We use \(FS(v)\) to denote the forward star of a vertex \(v\), namely the set of vertices pointed to by \(v\) in \(G(S)\); and we use \(BS(v)\) to denote the backward star of \(v\), namely the set of vertices pointing to \(v\) in \(G(S)\). Notice that \(G(S)\) is a DAG, since all of its edges \((v_i, v_j)\) are oriented rightward (i.e. \(i < j\)), moreover the sequence of vertices sorted by increasing text positions is a topological order of \(G(S)\). Actually the indices of the vertices in \(FS(v)\) and \(BS(v)\) form a contiguous range:

**Fact 6** Given a vertex \(v_i\) and let \(v_{i+x}\) and \(v_{i-y}\) be respectively the vertex with greatest index in \(FS(v_i)\) and the smallest index in \(BS(v_i)\), it holds

- \(FS(v_i) = \{v_{i+1}, \ldots, v_{i+x-1}, v_{i+x}\}\) and
- \(BS(v_i) = \{v_{i-y}, \ldots, v_{i-2}, v_{i-1}\}\).

Furthermore, \(x, y\) are smaller than the length of the longest repeated substring in \(S\).

**Proof:** By definition of \((v_i, v_{i+x})\), string \(S[i : i + x - 1]\) occurs at some position \(p < i\) in \(S\). Any prefix \(S[i : k - 1]\) of \(S[i : i + x - 1]\) also occurs at that position \(p\), thus \(v_k \in FS(v_i)\). The bound on \(x\) derives from the definition of \((v_i, v_{i+x})\). A similar argument holds for \(BS(v_i)\).

This means that if an edge does exist in \(G(S)\), then there exist also all the edges which are nested within it and are incident into one of its extremes. The
following property relates the indices of the vertices \( v_j \in FS(v_i) \) with the cost of their connecting edge \((v_i, v_j)\), and not surprisingly shows that the smaller is \( j \) (i.e. shorter edge), the smaller is the cost of encoding the phrase \( S[i : j - 1] \):

**Fact 7** Given a vertex \( v_i \), for any pair of vertices \( v_{j'}, v_{j''} \in FS(v_i) \) such that \( j' < j'' \), we have that \( c(v_i, v_{j'}) \leq c(v_i, v_{j''}) \). The same property holds for \( v_{j'}, v_{j''} \in BS(v_i) \).

**Proof:** We have that \( d_{i,j'} \leq d_{i,j''} \) and \( \ell_{i,j'} < \ell_{i,j''} \) because \( S[i : j' - 1] \) is a prefix of \( S[i : j'' - 1] \) and thus the first substring occurs wherever the latter occurs. The property holds because \( f \) and \( g \) satisfy the Increasing Cost Property 1.

Given these monotonicity properties, we are ready to characterize a special subset of the vertices in \( FS(v_i) \) and their connecting edges.

**Definition 8** An edge \((v_i, v_j) \in E\) is called

- \( d\)-maximal iff the next edge from \( v_i \) takes more bits to encode its distance: \( |f(d_{i,j})| < |f(d_{i,j+1})| \).
- \( \ell\)-maximal iff the next edge from \( v_i \) takes more bits to encode its length: \( |g(l_{i,j})| < |g(l_{i,j+1})| \).

Edge \((v_i, v_j)\) is called maximal if it is either \( d\)-maximal or \( \ell\)-maximal: thus \( c(v_i, v_j) < c(v_i, v_{j+1}) \).

Given an edge \((v_i, v_j)\), we will call \( d\)-cost the value \( |f(d_{i,j})| \) (i.e. the amount of bits needed to encode its distance value) and \( l\)-cost the value \( |g(d_{i,j})| \). Before studying the main properties of maximal edges, we need to introduce some formalisms.

Suppose that \( e \) denotes a generic encoding functions mapping the integers in the range \([n]\) into binary strings. We define \( Q(e, n) \) as the number of different codeword lengths generated by \( e \) when applied to integers in the range \([n]\), in symbols \( Q(e, n) = |\{ |e(m)| | m \in [n]\}| \). The encoding \( e \) induces an equivalence relation \( \equiv_e \) on \([n]\) such that \( x \equiv_e y \iff |e(x)| = |e(y)| \). We will always denote with \( I_1^e, I_2^e, \ldots, I_{Q(e,n)}^e \) the subsets of \([n]\) that are equivalence classes for \( \equiv_e \) and assume that the numbering of this sets is such that if \( i < j \) then \( |e(x)| < |e(y)| \), for any \( x \in I_i^e \) and any \( y \in I_j^e \).

Notice that, when \( e \) satisfies Property 1, each \( I_k^e \) is a sub-interval of \([n]\). The number of maximal edges depends on the functions \( f \) and \( g \) (which satisfy Property 1)

**Lemma 9** There are at most \( Q(f, n) + Q(g, n) \) maximal edges outgoing from any vertex \( v_i \).

**Proof:** By Fact 6, vertices in \( FS(v_i) \) have indices in a range \( R \), and by Fact 7, \( c(v_i, v_j) \) is monotonically non-decreasing as \( j \) increases in \( R \). Moreover we know that \( f \) (resp. \( g \)) cannot change more than \( Q(f, n) \) (resp. \( Q(g, n) \)) times, so that the statement follows.
3.5. AN EFFICIENT BIT-OPTIMAL PARSER

To speed up the computation of a SSSP from \( v_1 \) to \( v_{n+1} \), we construct a subgraph \( \tilde{G}(S) \) of \( G(S) \) which is formed by maximal edges only, it is smaller than \( G(S) \) and contains one of those SSSP. The following fundamental theorem ensures that we can restrict the SSSP computation to \( \tilde{G}(S) \) and still finding a shortest path for \( G(S) \):

**Theorem 10** There exists a shortest path in \( G(S) \) from \( v_1 \) to \( v_{n+1} \) that traverses maximal edges only.

**Proof:** By contradiction assume that every such shortest path contains at least one non-maximal edge. Let \( \pi = v_{i_1}v_{i_2} \cdots v_{i_h} \), with \( i_1 = 1 \) and \( i_h = n + 1 \), be one of these shortest paths, and let \( \gamma = v_{i_1} \cdots v_{i_r} \) be the longest initial subpath of \( \pi \) which traverses maximal edges only. Assume w.l.o.g. that \( \pi \) is the shortest path maximizing the value of \( |\gamma| \). We know that \( (v_{r}, v_{r+1}) \) is a non-maximal edge, and thus we can take the maximal edge \( (v_{r}, v_j) \) that has the same cost. By definition of maximal edge, it is \( j > i_{r+1} \); furthermore, we must have \( j < n + 1 \) because we assumed that no path is formed by maximal edges only. Now, since \( G(S) \) is a DAG and indices in \( \pi \) are increasing, it must exist an index \( i_h \geq i_r \) such that the index of that maximal edge \( j \) lies in \([i_h, i_{h+1}]\). Since \( (v_{i_h}, v_{i_{h+1}}) \) is an edge of \( \pi \), it does exist the edge \( (v_j, v_{i_{h+1}}) \) (by Fact 6), and by Fact 7 on \( BS(v_{i_{h+1}}) \) we can conclude that \( c(v_j, v_{i_{h+1}}) \leq c(v_{i_h}, v_{i_{h+1}}) \). Consequently, the path \( v_{i_1} \cdots v_r v_j v_{i_{h+1}} \cdots v_{i_k} \) is also a shortest path but its longest initial subpath of maximal edges consists of \(|\gamma| + 1\) vertices, which is a contradiction!

Theorem 10 implies that the distance between \( v_1 \) and \( v_{n+1} \) is the same in \( G(S) \) and \( \tilde{G}(S) \), with the advantage that computing SSSP in \( \tilde{G}(S) \) can be done faster and in reduced space, because \(|FS(v)| \leq Q(f, n) + Q(g, n)\) (Lemma 9). Thus, subgraph \( \tilde{G}(S) \) consists of \( n + 1 \) vertices and at most \( n(Q(f, n) + Q(g, n)) \) edges. For Elias’ codes [24], Fibonacci’s codes [26], and most practical integer encoders used for search engines and data compressors [78, 92], it is \( Q(f, n) = Q(g, n) = O(\log n) \). Therefore \( |\tilde{G}(S)| = O(n \log n) \), so it is smaller than the complete graph built and used by previous papers [79, 51]. For the encoders used in gzip, it is \( Q(f, n) = Q(g, n) = O(1) \) and \( |\tilde{G}(S)| = O(n) \).

### 3.5 An Efficient Bit-Optimal Parser

In this section we will exploit the reduction achieved in Section 3.4 to compute efficiently the bit-optimal LZ77-parsing of the input text. Theorem 10 implies that computing a bit-optimal LZ77-parsing of a string \( S \) is equivalent to computing a shortest path in graph \( \tilde{G}(S) \) from \( v_1 \) to \( v_{n+1} \).

The graph \( \tilde{G}(S) \) is a DAG having \( n \) vertices and \( O(n(Q(f, n) + Q(g, n))) \) edges. If we could efficiently build \( \tilde{G}(S) \) then using the SSSP algorithm of Fact 1 would immediately yield a bit-optimal LZ77-parsing of \( S \) in \( O(n(Q(f, n) + Q(g, n))) \) time, which is \( O(n \log n) \) for many practical choices of \( f \) and \( g \) (see the final remark of Section
3.4). The main issue with this approach is that naively applying the SSSP algorithm requires the input graph $\tilde{G}(S)$ to be stored in memory. Unfortunately, encoding $\tilde{G}(S)$ in a usual format (like Adjacency Lists) occupy $O(|\tilde{G}(S)|) = O(n(Q(f,n) + Q(g,n)))$ words of memory, that is a superlinear amount of space in the size of $S$ whenever $Q(f,n) + Q(g,n) = \Omega(1)$.

To achieve an efficient solution working in optimal $O(n)$ space we devise a technique to dynamically generate the edges of $\tilde{G}(S)$ as they are inspected during the SSSP computation. In order to compute the shortest path from $v_1$ to $v_{n+1}$ in $\tilde{G}(S)$ we modify the classic SSSP algorithm for DAGs in such a way that, rather than having the input graph $\tilde{G}(S)$ materialized in memory, we use the previous technique to selectively generate the edges of $\tilde{G}(S)$ leaving from the current vertex.

Therefore, the key difficulty in implementing our approach consists of how to generate on-the-fly and efficiently (in time and space) the maximal edges outgoing from vertex $v_i$. We will refer to this problem as the forward-star generation problem, and use FSG for brevity. In what follows we consider the case of integer alphabet with $\sigma \leq n$, and show that FSG takes $O(1)$ amortized time per edge and $O(n)$ space in total 3.5.3. In case of a larger alphabet, we can always remap the symbols of $S$ into alphabet $[n]$ by spending additional proportional to $T_{\text{sort}}(n, \Sigma)$, which as in Section 1.1 denotes the time complexity of sorting an $n$-elements multiset of symbols drawn from alphabet $\Sigma$. Since we have $n$ vertices, with no more than $Q(f,n) + Q(g,n)$ maximal edges each (Lemma 9), we will obtain the following:

**Theorem 11** Given a string $S$ and two integer-encoding functions $f$ and $g$ that satisfy Property 1, there exists a compressor that computes the $(f,g)$-optimal LZ77-parsing of $S$ in $O(n(Q(f,n) + Q(g,n)) + T_{\text{sort}}(n, \Sigma))$ time and $O(n)$ space in the worst case.

Our main result of Theorem 11 shows that the bit-optimal LZ77-parsing of a string $S[1,n]$ can be computed in $O(n \log n)$ time and $O(n)$ optimal space in the worst case, for most of the integer-encoders frequently used in the literature [26]. To the best of our knowledge, this is the first result of such a type that answers positively to the question posed by Rajpoot and Sahinalp in [76, pag. 159].

### 3.5.1 Some Notation and Basic Facts

Before presenting our solution to the FSG-problem we introduce some useful notations and a simple observation about maximal edges.

From the definition of $d$-maximal edges, we know that for each cost class $I_k^f$ induced by $f$, there is at most one $d$-maximal edge of cost $c(I_k^f)$. We adopt the notation introduced in Section 1.5, and use $\text{lcp}(S_i, S_j)$ to denote the longest common prefix of the suffixes $S_i$ and $S_j$ and $\prec / \succ$ to denote the usual lexicographic ordering over strings.

Fix an arbitrary cost class $I_k^f$ induced by $f$, in the following we will drop the superscript and denote it by $I_k$. Since $f$ satisfies Property 1, $I_k$ is a sub-interval of
[n] of the form $I_k = [r, l]$. For any position $h$ in $S$, let us define $W_h = [h - r, h - l]$, in other words, $W_h$ is the window of text position whose distance from $h$ can be encoded in $c(I_k)$ bits. The next fact is crucial to efficiently generate $d$-maximal edges via indexing data structures built over $S$:

**Fact 12** If there exists a $d$-maximal edge outgoing from $v_h$ and having $d$-cost $c(I_k)$, then this edge can be found by determining a position $s \in W_h$ whose suffix $S_s$ shares the maximum longest common prefix with $S_h$.

**Proof:** Among all positions $s$ in $W_h$ take one whose suffix $S_s$ shares the maximum $1\text{cp}$ with $S_h$, and let $q$ be the length of this $1\text{cp}$. Of course, there may exist many such positions, we take just one of them. The edge $(v_h, v_{h+q+1})$ has $d$-cost $c(I_k)$, because $s \in W_h$, and is $d$-maximal because any other position $s' \in W_h$ induces an edge $(v_h, v_{h+q'+1})$ whose length $q' \leq q$, by maximality of $q$. So any edge $(v_h, v_{h+q''})$, with $q'' > q$, must reference a copy before $W_h$, if any.

Hereafter we call the position $s$ of Fact 12 *maximal position* for vertex $v_h$, and note that it does exist only if $v_h$ has a $d$-maximal edge of $d$-cost $c(I_k)$. In particular, from Lemma 12, it follows that once $s$ has been computed the corresponding $d$-maximal edge $(v_h, v_q)$ and its label $(d_{h,q}, \ell_{h,q})$ are uniquely determined by: $q = h + 1\text{cp}(S_s, S_h)$, $d_{h,q} = h - s$ and $\ell_{h,q} = q - h$. In other words, the $d$-maximal edge $(v_h, v_q)$ of $d$-cost $c(I_k)$ actually encode the phrase $S[h : q - 1]$ which is the longest substring starting at $h$ and having a previous copy at distance within $I_k$.

### 3.5.2 Dynamic Generation of Maximal Edges

Here we discuss a first solution to the FSG problem which requires $O(\log n)$ time to generate each maximal edge, then in subsection 3.5.3 we will achieve $O(1)$ time per maximal edge using a more sophisticated approach. The solution proposed here is asymptotically slower than the one of subsection 3.5.3, but has the advantage of being much simpler, in fact we have used it to implement a bit-optimal LZ77-parser (see the experimental data in Section 3.5.3).

We will treat $d$-maximal edges and $\ell$-maximal edges using different methods. Once $d$-maximal edges outgoing from the current vertex $v_i$ are known, the computation of the $\ell$-maximal edges is then easy because it suffices to further decompose the edges between successive $d$-maximal edges, say between $(v_i, v_{h_{k-1}+1})$ and $(v_i, v_{h_k})$, according to the distinct values assumed by the encoding function $g$ on the lengths in the range $[h_{k-1}, \ldots, h_k - 1]$. This takes $O(1)$ time per $\ell$-maximal edge, because it needs some algebraic calculations, and the corresponding copied substring can then be inferred as a prefix of $S[i : h_k - 1]$.

So, let us concentrate on the computation of $d$-maximal edges outgoing from vertex $v_i$. Our solution deploys as basic ingredients the suffix array of $S$, denoted by $SA$, and a data structure for computing the $1\text{cp}$s of two suffixes in $O(1)$ time.
(Section 1.1). From Theorems 3 and 4 we know that $SA$ can be built in time $T_{sort}(n, \sigma) + O(n)$ and the 1cp data structure can be built from $SA$ in additional $O(n)$ time and encoded in $O(n)$ bits.

Thanks to fact 12 and the subsequent discussion, the FSG problem reduces to computing the maximal positions of the current vertex $v_i$ with respect to every possible cost class for the encoding $f$. Let us fix one such cost class $I_k$ and focus on the computation of the maximal position for $I_k$. If such maximal position does not exist, our algorithm will generate an arbitrary position. The net result is that we will generate a supergraph of $\tilde{G}(S)$ which is still guaranteed to have the size stated in Lemma 9. Notice that the correctness of the SSSP computation is preserved, since the shortest path distance from $v_1$ and $v_{n+1}$ cannot be increased by inserting non-maximal edges into $\tilde{G}(S)$.

Consider the window $W_i = [i - r, i - l]$ of all the positions whose distance from $i$ can be encoded in $c(I_k)$ bits, as defined in the previous subsection. The maximal position $s$ is the one that maximizes $1\text{cp}(S_i, S_s)$ among all positions of $W_i$. Let $R_i$ denote the set of suffixes starting in $W_i$. We say that a suffix $S_j$ of $R_i$ is the predecessor in the set $R_i$ of suffix $S_i$ iff (i) $S_j \prec S_i$ and (ii) there is no suffix $S_x$ of $R_i$ such that $S_j \prec S_x \prec S_i$. We define the successor in $R_i$ of $S_i$ in a symmetrical way, by changing $\prec$ with $\succ$.

We maintain $R_i$ in a dynamic data structure supporting the following two queries:

- pred$(x)$: return $p \in W_i$ such that $S_p$ is the predecessor in $R_i$ of $S_x$
- succ$(x)$: return $p \in W_i$ such that $S_p$ is the successor in $R_i$ of $S_x$

In our solution we adopt a simple balanced binary search tree over $R_i$, where each suffix is represented by its starting position in $W_i$. Notice that we can lexicographically compare two suffixes of $S$ in $O(1)$ time, by checking which one of the two has highest rank in $SA$. Therefore, using the balanced search tree, the operations pred$(\cdot)$, succ$(\cdot)$ and insertion/removal of a suffix can be supported with $O(\log n)$ lexicographic comparisons among suffixes, which in total require $O(\log n)$ time. The space occupied by the tree is $O(|W_i|) = O(|I_k|)$.

When the current position advance to $i + 1$ the tree is updated to contain the suffixes of the set $R_{i+1}$. Since $R_{i+1} = (R_i \setminus S_{i-r} \cup S_{i+1-l})$, it is necessary to just remove $S_{i-r}$ and insert $S_{i+1-l}$ in the tree, thus spending $O(\log n)$ time.

It is easy to see that the suffix of $R_i$ sharing the longest maximal prefix with $S_i$ must be either its lexicographic predecessor or successor. Therefore we can determine the maximal position $s$ by computing pred$(i)$ and succ$(i)$ and choosing the one giving bigger 1cp with $S_i$. Since 1cp’s can be computed in $O(1)$ time, this solution spends $O(\log n)$ time for computing each maximal position.

Overall, the time required for generating each maximal edge using this solution is $O(n(Q(f, n) \log n + Q(g, n)))$. The total space occupied to store a balanced search tree for each cost class is $O(\sum_k |I_k|)$, which is bounded by $O(n)$, since the cost classes for $f$ are a partition of $[n]$. 
3.5.3 Achieving $O(1)$ amortized time per edge

This section further investigates the FSG problem, presenting an improved solution running in $O(1)$ time per maximal edge and $O(n)$ space. To achieve this result we deploy two key ideas:

1. The first idea aims at achieving the optimal $O(n)$ working-space bound. The algorithm proceeds in $Q(f,n)$ parallel passes, one per class $I_k$ of possible d-costs for the edges in $\tilde{G}(S)$. During the $k$th pass, we logically partition the vertices of $\tilde{G}(S)$ in blocks of $|I_k|$ contiguous vertices, say $v_{i_k}, v_{i_k+1}, \ldots, v_{i_k+|I_k|-1}$, and compute all d-maximal edges which spread out from that block and have copy-distance within $I_k$ (thus they all have the same d-cost, say $c(I_k)$). These edges are kept in memory until they are used by our bit-optimal parser, and discarded as soon as the first vertex of the next block, i.e. $v_{i_k+|I_k|}$, needs to be processed. The next block of $|I_k|$ vertices is then fetched and the process repeats. All passes are executed in parallel to guarantee that all d-maximal edges of $v_i$ are available when processing this vertex. There are $n/|I_k|$ distinct blocks at each pass, and each d-maximal edge of a vertex is considered in some pass (because it has d-cost in some $I_k$). The space is $\sum_{k=1}^{Q(f,n)} |I_k| = O(n)$ because we keep one d-maximal edge per vertex at any pass.

2. The second key idea aims at computing the d-maximal edges for the current block of $|I_k|$ contiguous vertices in $O(|I_k|)$ time and space. This is what we address below, being the most sophisticated technicality of our solution. As a result, we show that the time complexity of FSG is $\sum_{k=1}^{Q(f,n)} (n/|I_k|)O(|I_k|) = O(n Q(f,n))$, i.e., $O(1)$ amortized time per d-maximal edge. Combining this fact with the previous observation on the computation of the $\ell$-maximal edges, we get Theorem 11.

Let us consider the $k$th pass of FSG in which we assume that $I_k = [l,r]$. Recall that all distances in $I_k$ can be $f$-encoded in the same number of, say, $c(I_k)$ bits. Let $B = [i, i + |I_k| - 1]$ be the block of (indices of) vertices for which we wish to compute on-the-fly the $d$-maximal edges of cost $c(I_k)$. This means that the $d$-maximal edge from vertex $v_h$, $h \in B$, represents a phrase that starts at $S[h]$ and has a copy starting in the window (of indices) $W_h = [h-r, h-l]$. Thus the distance of that copy can be $f$-encoded in $c(I_k)$ bits, and so we will say that the edge has d-cost $c(I_k)$. Since this computation must be done for all vertices in $B$, it is useful to consider the window $W_B = W_i \cup W_{i+|I_k|-1}$ which merges the first and last window of positions that can be the (copy-)reference of any $d$-maximal edge outgoing from $B$. Note that $|W_B| = 2|I_k|$ (see Figure 3.1) and it spans all positions where the copy of a $d$-maximal edge outgoing from $B$ can occur.

So, let us assume that we are given the trie $T_B$. We notice that the maximal position $s$ for a vertex $v_i$ in $B$ having d-cost $c(I_k)$ can be computed by finding the leaf of $T_B$ which is labeled with an index $s$ that belongs to the range $W_h$ and has
the maximum lcp with the leaf labeled h. This actually corresponds to find the leaf whose label $s \in W_h$ and has the deepest lca (Section 1.1) with the leaf labeled $h$. We need to answer this query in $O(1)$ amortized time per vertex $v_h$, since we aim at achieving an $O(|I_k|)$ time complexity over all vertices in $B$. This is not easy because this is not the classic lca-query since we do not know $s$, which is actually the position we are searching for! The solution of subsection 3.5.2 computes $h$ by using proper predecessor/successor queries on a suitable dynamic set of suffixes in $W_h$. Unfortunately, this requires at least $\omega(1)$ time because of well-known lower bounds [4]. Therefore, in order to answer this query in constant (amortized) time per vertex of $B$, we deploy proper structural properties of the Trie $\mathcal{T}_B$ and the problem at hand.

**Definition 13** For any node $u \in \mathcal{T}_B$, we define $a(u)$ as the minimum index in $B$ such that $v_{a(u)}$ is a descendant of $u$ in $\mathcal{T}_B$.

We make now a crucial statement about maximal positions. Assume for simplicity that the window $W_h$ strictly precedes $B$ for any $h \in B$ (our algorithm deals with these cases too, see the proof of Lemma 15).

**Fact 14** If $s$ is a maximal position of an arbitrary position $h$, and $u$ is the lca of the leaves labeled $h$ and $s$ in $\mathcal{T}_B$, then it must be $h = a(u)$.

In fact assume, by contradiction, that $a(u) = h' < h$ does exist. By definition $h' \in B$ and thus $v_{h'}$ would not have a $d$-maximal edge of $d$-cost $c(I_k)$ because it could copy from the closer $h'$ a possibly longer phrase, instead of copying from the farther set of positions in $W_h$. Computing the value $a()$ for all nodes $u$ in $\mathcal{T}_B$ takes $O(|\mathcal{T}_B|) = O(|I_k|)$ time and space via a traversal of the trie $\mathcal{T}_B$. Thanks to observation 14, we can compute the maximal position of $a(u)$ for each node $u$ of $\mathcal{T}_B$ by searching it among the indices of the leaves descending from $u$. When this process is repeated for each node $u$ it generates the maximal positions of every node for which one of these exists.

We cannot resolve a node $u$ by trivially checking each leaf in the subtree of $u$, because this would take quadratic time complexity overall. To achieve a linear-time solution we have to deploy another observation. Let us define $W_B'$ and $W_B''$ as the
first and the second half of \( \mathcal{W}_B \), respectively. Observe that any window \( W_k \) has its left extreme in \( \mathcal{W}''_B \) and its right extreme in \( \mathcal{W}''_B \) (see Figure 3.1). Therefore the window \( W_a(u) \) containing the maximal position \( s \) for \( v_a(u) \) overlaps both \( \mathcal{W}'_B \) and \( \mathcal{W}''_B \). If \( s \) does exist for \( v_a(u) \), then \( s \) belongs to either \( \mathcal{W}'_B \) or to \( \mathcal{W}''_B \), and the leaf labeled \( s \) descends from \( u \). Hence the maximum (resp. minimum) among the elements in \( \mathcal{W}'_B \) (resp. \( \mathcal{W}''_B \)) that label leaves descending from \( u \) must belong to \( W_a(u) \).

This suggests to compute for each node \( u \) the rightmost position in \( \mathcal{W}'_B \) and the leftmost position in \( \mathcal{W}''_B \) that label a leaf descending from \( u \), denoted respectively by \( \text{max}(u) \) and \( \text{min}(u) \). This takes \( O(|I_k|) \) time with a post-order visit of \( \mathcal{T}_B \). We can now efficiently compute \( \text{mp}[h] \) as the maximal position for \( v_h \), if it exists, or otherwise set \( \text{mp}[h] \) arbitrarily. The pseudo-code of our strategy is reported in Algorithm 1.

**Algorithm 1** Generate maximal position for each position of \( B 

```plaintext
for each node \( u \) of \( \mathcal{T}_B \) in post-order do
  if \( \text{mp}[a(u)] = \text{nil} \) then
    if \( \text{min}(u) \in W_a(u) \) then
      \( \text{mp}[a(u)] \leftarrow \text{min}(u) \)
    else if \( \text{max}(u) \in W_a(u) \) then
      \( \text{mp}[a(u)] \leftarrow \text{max}(u) \)
    else
      \( \text{mp}[a(u)] \leftarrow a(\text{parent}(u)) \)
  end if
end for
```

We initially set all \( \text{mp} \)'s entries to \( \text{nil} \); then we visit \( \mathcal{T}_B \) in post-order and perform, at each node \( u \), the following two checks whenever \( \text{mp}[a(u)] = \text{nil} \): If \( \text{min}(u) \in W_a(u) \), we set \( \text{mp}[a(u)] = \text{min}(u) \); if \( \text{max}(u) \in W_a(u) \), we set \( \text{mp}[a(u)] = \text{max}(u) \). At the end of the visit, if \( \text{mp}[a(u)] \) is still \( \text{nil} \) we set \( \text{mp}[a(u)] = a(\text{parent}(u)) \) whenever \( a(u) \neq a(\text{parent}(u)) \). This last check is needed (see proof of Lemma 15) to manage the case in which \( S[a(u)] \) can copy the phrase starting at its position from position \( a(\text{parent}(u)) \) and, additionally, we have that \( B \) overlaps \( \mathcal{W}_B \) (which may occur depending on \( f \)). Since \( \mathcal{T}_B \) has size \( O(|I_k|) \), the overall algorithm requires \( O(|I_k|) \) time and space in the worst case, and hence Theorem 11 follows. Correctness of algorithm 1 follows from lemma below.

**Lemma 15** For each position \( h \in B \), if there exists a \( d \)-maximal edge outgoing from \( v_h \) and having \( d \)-cost \( c(I_k) \), algorithm 1 correctly sets \( \text{mp}[h] \) to its maximal position.

**Proof:** Recall that \( B = [i, i + |I_k| - 1] \) and consider the longest path \( \pi = u_1 u_2 \ldots u_z \) in \( \mathcal{T}_B \) that starts from the leaf \( u_1 \) labeled with \( h \in B \) and goes upward until the traversed nodes satisfy the condition \( a(u_j) = h \), here \( j = 1, \ldots, z \). By definition of \( a \)-value 13, we know that all leaves descending from \( u_z \) and occurring in \( B \) are
labeled with an index which is larger than \( h \). Clearly, if \( \text{parent}(u_z) \) does exist, then it is \( a(\text{parent}(u_z)) < h \). There are two cases for the final value stored in \( \text{mp}[h] \).

**Case 1.** Suppose that \( \text{mp}[h] \in W_h \). We want to prove that \( \text{mp}[h] \) is the index of the leaf which has the deepest \( \text{lca} \) with \( h \) among all the other leaves labeled with an index in \( W_h \). Let \( u_x \in \pi \) be the node in which the value of \( \text{mp}[h] \) is assigned. Then, by our algorithm it is \( a(u_x) = h \). Assume now that there exists at least another index in \( W_h \) whose leaf has a deeper \( \text{lca} \) with leaf \( h \). This \( \text{lca} \) must lie on \( u_1 \ldots u_x \), say \( u_i \). Since \( W_h \) is a window having its left extreme in \( \mathcal{W}_B \) and its right extreme in \( \mathcal{W}_B' \), the value \( \min(u_i) \) or \( \max(u_i) \) must lie in \( W_h \) and thus the algorithm has set \( \text{mp}[h] \) to one of these positions, because of the post-order visit of \( \mathcal{T}_B \) and the check on \( \text{mp}[a(u_x)] = \text{nil} \). Therefore \( \text{mp}[h] \) must be the index of the leaf having the deepest \( \text{lca} \) with \( h \), and thus by Fact 12 it is its maximal position.

**Case 2.** Suppose that \( \text{mp}[h] \notin W_h \) and, thus, it cannot be a maximal position for \( v_h \). We have to prove that it does not exist a \( d \)-maximal edge outgoing from the vertex \( v_h \) with cost \( c(I_k) \). Let \( S_s \) be the suffix in \( W_h \) having the maximum \( \text{lcp} \) with \( S_h \), and let \( l \) be the \( \text{lcp} \)-length. Values \( \min(u_i) \) and \( \max(u_i) \) do not belong to \( W_h \), for any node \( u_i \in \pi \) with \( a(u_i) = h \), otherwise \( \text{mp}[h] \) would have been assigned with an index in \( W_h \) (contradicting the hypothesis). Thus the value of \( \text{mp}[h] \) remains \( \text{nil} \) up to node \( u_x \). This implies that no suffix descending from \( u_x \) starts in \( W_h \) and, in particular, \( S_x \) does not descend from \( u_x \). Therefore, the \( \text{lca} \) between leaves \( h \) and \( s \) is a node in the path from \( \text{parent}(u_z) \) to the root of \( \mathcal{T}_B \), and the \( \text{lcp} (S_{a(\text{parent}(u_z))}, S_h) \geq \text{lcp}(S_s, S_h) = l \). Since, by definition 13, \( a(\text{parent}(u_z)) < a(u_z) \) and it belongs to \( B \) this position is nearer to \( h \) than any other position in \( W_h \) and shares a longer prefix with \( S_h \). So we found a longer edge from \( v_h \) with smaller \( d \)-cost. This way \( v_h \) has no \( d \)-maximal edge of cost \( c(I_k) \) in \( \mathcal{G}(S) \).

### 3.5.4 Building \( \mathcal{T}_B \) optimally

In the discussion above we left the explanation on how to build \( \mathcal{T}_B \) in \( O(|I_k|) \) time and space, thus within a time complexity which is independent of the length of the indexed suffixes and the alphabet size. To achieve this result we deploy the crucial fact that the algorithm of Section 3.5.3 does not make any assumption on the ordering of the edges in \( \mathcal{T}_B \), because it just computes (sort of) \( \text{lca} \)-queries on its structure.

At preprocessing time we build the suffix array of the whole string \( S \) and a data structure that answers constant-time \( \text{lcp} \)-queries between pair of suffixes. These data structures can be built in \( O(n) \) time and space, when \( \sigma = O(n) \), or with an additional \( T_{\text{sort}}(n, \Sigma) \) time, in the case of larger alphabet (see Theorem 3).

Let us first assume that \( B \) and \( \mathcal{W}_B \) are contiguous and form the range \([i, i + 3|I_k| - 1]\). If we had the sorted sequence of suffixes starting in \( S[i, i + 3|I_k| - 1] \), we could easily build \( \mathcal{T}_B \) in \( O(|I_k|) \) time and space by deploying the above \( \text{lcp} \)-data structure. Unfortunately, it is unclear how to obtain from the suffix array of the
whole $S$, the sorted sub-sequence of suffixes starting in the range $[i, i + 3|I_k| - 1]$ by taking $O(|B| + |W_B|) = O(|I_k|)$ time (notice that these suffixes have length $\Theta(n-i)$). We cannot perform a sequence of predecessor/successor queries because they would take $\omega(1)$ time each [4]. Conversely, we resort the key observation above that $T_B$ does not need to be ordered, and thus devise a solution which builds an unordered $T_B$ in $O(|I_k|)$ time and space, passing through the construction of the suffix array of a transformed string. The transformation is simple. We first map the distinct symbols of $S[i, i + 3|I_k| - 1]$ to the first $O(|I_k|)$ integers. This mapping does not need to reflect their lexicographic order, and thus can be computed in $O(|I_k|)$ time by a simple scan of those symbols and the use of a table $T$ of size $\sigma < n$. Then, we define $\hat{S}$ as the string $S$ which has been transformed by re-mapping some of the symbols according to table $T$ (namely, those occurring in $S[i, i + 3|I_k| - 1]$). We can prove the following Lemma.

**Lemma 16** Let $S_i, \ldots, S_j$ be a contiguous sequence of suffixes in $S$. The re-mapped suffixes $\hat{S}_i \ldots \hat{S}_j$ can be lexicographically sorted in $O(j - i + 1)$ time.

**Proof:** Consider the string of pairs $w = \langle \hat{S}[i], b_i \rangle \ldots \langle \hat{S}[j], b_j \rangle $, where $b_h$ is 1 if $\hat{S}_{h+1} > \hat{S}_{j+1}$, $-1$ if $\hat{S}_{h+1} < \hat{S}_{j+1}$, or 0 if $h = j$. The ordering of the pairs is defined component-wise, and we assume that $\$ is a special “pair” larger than any other pair in $w$. For any pair of indices $p, q \in [1 \ldots j - i]$, it is $\hat{S}_{p+i} > \hat{S}_{q+i}$ iff $w_p > w_q$. In fact, suppose that $w_p > w_q$ and set $r = \text{lcp}(w_p, w_q)$. We have that $w[p + r] = \langle \hat{S}[p + i + r], b_{p+i+r} \rangle > \langle \hat{S}[q + i + r], b_{q+i+r} \rangle = w[q + i + r]$. Hence $\hat{S}_{p+i+r} > \hat{S}_{q+i+r}$, by definition of the $b$’s. Therefore $\hat{S}_{p+i} > \hat{S}_{q+i}$, since their first $r$ symbols are equal. This implies that sorting the suffixes $\hat{S}_i \ldots, \hat{S}_j$ reduces to computing the suffix array of $w$, and this takes $O(|w|)$ time given that the alphabet size is $O(|w|)$. Clearly, $w$ can be constructed in that time bound because comparing $\hat{S}_z$ with $\hat{S}_{z+1}$ takes $O(1)$ time via an $\text{lcp}$-query on $S$ and a check at their first mismatch.

Lemma 16 allows us to generate the compact trie of $\hat{S}_i, \ldots, \hat{S}_{i+3|I_k|-1}$, which is equal to the (unordered) compacted trie of $S_i, \ldots, S_{i+3|I_k|-1}$ after replacing every $1\text{D}$ assigned by table $T$ with its original symbol in $S$. We finally notice that if $B$ and $W_B$ are not contiguous (as instead we assumed above), we can use a similar strategy to sort separately the suffixes in $B$ and the suffixes in $W_B$, and then merge these two sequences together by deploying the $1\text{cp}$-data structure mentioned at the beginning of this section.

### 3.6 Experimental Results

In this section we report our experimental study on bit-optimal LZ77-parsing. We have implemented a variant of the algorithm of subsection 3.5.2, called BitOptimal-LZ77,
and compared it with some state-of-the-art compression tools. The results are reported in Table 3.6. The experiments were run over a set of few freely available text collections.

<table>
<thead>
<tr>
<th>Compressor</th>
<th>english 33%</th>
<th>C/C++/Java src 33%</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip -9</td>
<td>37.52%</td>
<td>23.29%</td>
</tr>
<tr>
<td>bzip2 -9</td>
<td>28.40%</td>
<td>19.78%</td>
</tr>
<tr>
<td>boosterOpt</td>
<td>20.62%</td>
<td>17.36%</td>
</tr>
<tr>
<td>Fixed-LZ77</td>
<td>26.19%</td>
<td>24.63%</td>
</tr>
<tr>
<td>Rightmost-LZ77</td>
<td>23.81%</td>
<td>20.14%</td>
</tr>
<tr>
<td>BitOptimal-LZ77</td>
<td>21.62%</td>
<td>17.62%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compressor</th>
<th>HTML 8%</th>
<th>Avg Dec. time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip -9</td>
<td>20.09%</td>
<td>0.7</td>
</tr>
<tr>
<td>bzip2 -9</td>
<td>10.63%</td>
<td>6.3</td>
</tr>
<tr>
<td>boosterOpt</td>
<td>3.89%</td>
<td>20.2</td>
</tr>
<tr>
<td>Fixed-LZ77</td>
<td>4.98%</td>
<td>0.8</td>
</tr>
<tr>
<td>Rightmost-LZ77</td>
<td>4.27%</td>
<td>0.9</td>
</tr>
<tr>
<td>BitOptimal-LZ77</td>
<td>3.87%</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Table 3.1: Each text collection consists of 50 Mbytes of data. All the experiments were executed on a 2.6 GHz Pentium 4, with 1.5 GB of main memory, and running Fedora Linux.

For testing the effectiveness of the bit-optimal parsing strategy, we have also compared our method with two compressors based on the LZSS scheme, called Fixed-LZ77 and Rightmost-LZ77. The algorithm Fixed-LZ77 uses an unbounded window and encode the distances and lengths of each phrase with a fixed number of bits. Its compression performance shows that an unbounded window may introduce a significant compression gain wrt to a bounded one, as used by gzip and bzip2 (see e.g. HTML), due to the presence in current (Web/text) collections of surprisingly many long repetitions at large distances. This provide additional support to our unbounded window assumption, even if our results extend to the bounded window case too.

Then we consider Rightmost-LZ77, which uses an unbounded window and variable-length encoders to represent distances and lengths. In order to minimize the number of bits in the encodings of distances, the parser selects the rightmost copy of each parsed phrase. An algorithm for efficiently locating the rightmost copy is described in Chapter 4. As expected, this technique combined with the use of variable-length encoders improves Fixed-LZ77, thus sustaining in practice the starting point of our theoretical investigation.

Finally, we tested our bit-optimal compressor BitOptimal-LZ77 finding that it improves Fixed-LZ77, as theoretically predicted in Lemma 20. Surprisingly, BitOptimal-LZ77 significantly improves bzip2 (which uses a bounded window) and
comes close to the booster (which uses an unbounded window [31]). Additionally, since BitOptimal-LZ77 adopts the same decompression algorithm of gzip, it retains its fast decompression speed which is at least one order of magnitude faster than decompressing bzip2’s or booster’s compressed files. This is a nice combination which makes BitOptimal-LZ77 practically relevant for a wide range of applications in which the paradigm is “compress once & decompress many times” (like in Web search engines and IR systems), or where the decompression system is less powerful than the compressor one (like a server that distributes data to clients, possibly mobile phones).
Chapter 4

Extensions

4.1 Overview

This chapter further extends the study of the bit-optimal parsing problem for dictionary-based compressors by considering two related questions:

1. In Chapter 3 we considered the LZ77 compression scheme, and devised an efficient bit-optimal parser based on a reduction to a shortest path computation over a particular DAG. We were able to prove specific properties of the parsing graph that allow to speed-up the shortest path computation by pruning a significant fraction of the edges. In this chapter, we will extend this approach to the dictionary construction scheme adopted by the ACB compressor (Section 4.2). We will prove that the considerations of Section 3.4 also apply to the case of ACB, this will allow us to prune the parsing graph using the same technique devised for LZ77. In this way we reduce the design of an efficient bit-optimal parsers for ACB to the problem of dynamically generating the pruned parsing graph, for which we provide a solution working in optimal linear space and efficient time.

2. Given a parsing of the input text, there can be more than one codeword representing a phrase. In order to minimize the number of bits output by the final compressor, the parser should obviously select for each phrase the codeword having the smallest encoding, and the question is how to efficiently compute it. In Section 4.3, we will propose an efficient solution for this problem when the parsing is the one produced by the greedy LZ77-parser and codewords are encoded with variable-length encoders satisfying property 1. For any codeword \((d_i, \ell_i)\) of the LZ77 dictionary, the value of \(\ell_i\) is fixed by the length of the phrase while the value of \(d_i\) depends on the position of the copy of the phrase, and the encoding size of the codeword is minimized when the value of \(d_i\) is the smallest possible. Thus, for any phrase of the parsing, finding the codeword with shortest encoding is equivalent to locate the closest previous occurrence
of that phrase in $S$. We will present an elegant, yet simple, algorithm which computes the closest copy of each phrase in the greedy LZ77-parsing of the input string. This method works in $O(n(1 + \frac{\log \sigma}{\log \log n}))$ overall time and $O(n)$ space (Theorem 20), which is optimal in terms of time/space performance when the alphabet has size $\text{polylog}(n)$ (hence almost all texts of practical interest). We have implemented this technique and discussed its compression performance in Section 3.6, (see Rightmost-LZ77). Notice that, for any choice of the distances and lengths encoders $f$ and $g$, this method cannot improve the bit-optimal LZ77-parser in compression, but its running time is inferior to that of the algorithm of Theorem 11 whenever the factor $Q(f, n) + Q(g, n)$ is larger than $\log \sigma / \log \log n$.

### 4.2 Bit-Optimal ACB Compression

Recall that ACB (Section 2.3) is a dynamic-dictionary compressor which differs from LZ77 in the definition of codewords. As LZ77, ACB encodes each phrase with a pair of integers consisting of a distance and a length component; the main difference with LZ77 is in the notion of distance. Let us denote with $S^i$ the context of the input string $S$ that ends in position $i$, that is $S^i = S[1 : i - 1]^R$ (recall the notation $x^R$ for denoting the reverse of string $x$). We conventionally put $S^1 = \$, where $\$ is an auxiliary symbol lexicographically smaller than any other symbol of the alphabet.

While compressing the string $S$, the algorithm maintains the sequence of contexts $S^1, S^2, \ldots, S^m$, where $m$ is the current parsing position, sorted in lexicographic order. Then, if the next phrase is copied from position $p$, the corresponding distance value is given by the difference between the offsets of $S^p$ and $S^{m-1}$ in the lexicographic ordering of the contexts.

We assume that each codeword $(\delta, \gamma)$ is encoded by the binary string $\delta_s f(|\delta|) g(\gamma)$, where $f$ and $g$ are two integer encoding functions satisfying property 1 and $\delta_s$ is a single bit encoding the sign of $\delta$; the encoding of the parsing is then obtained by concatenating the encodings of each codeword. Our aim is to efficiently compute a parsing of the input text that achieves optimality in the bit-length of its encoding.

Notice that the only difference with computing the bit-optimal LZ77-parsing is in how codewords are defined, while the phrases available to the parser at each step are the same, since the dictionaries of ACB and LZ77 consists of the same set of phrases.

#### 4.2.1 Pruned Parsing Graph

Not surprisingly, we reduce the computation of the bit-optimal parsing to a shortest path computation over a weighted parsing graph $\mathcal{G}(S)$. The graph $\mathcal{G}(S)$ has the same edge- and vertex-set of the graph defined in Section 3.4, since in both cases the dictionary consists of the same set of phrases.
4.2. BIT-OPTIMAL ACB COMPRESSION

We label each edge \((v_i, v_j)\) with the integer pairs \(\langle \delta_{i,j}, \lambda_{i,j} \rangle\), where \(\delta_{i,j}\) and \(\lambda_{i,j}\) are the distances and length components of the codeword assigned to phrase \(S[i : j - 1]\). The weight assigned to \((v_i, v_j)\) is \(c(v_i, v_j) = 1 + \left| f(|\delta_{i,j}|) \right| + \left| g(\lambda_{i,j}) \right|\), which accounts for the cost of encoding the codeword using \(f\) and \(g\) (plus one bit of sign). This weighting scheme ensures that the length in bits of a parsing is equal to the cost of the corresponding path from \(v_1\) to \(v_{n+1}\) in \(G(S)\), which allow us to reduce the problem of computing a bit-optimal parsing for ACB on input string \(S\) to a single-source shortest-path (SSSP) computation over \(G(S)\).

The graph \(G(S)\) still satisfy all the good properties of the parsing graph of LZ77. Since the structure of the graph is the same, the assertion of Fact 6 still holds; that is, for any vertex \(v\), the indices of the vertices in \(FS(v)\) and \(BS(v)\) form a contiguous range. The following theorem proves that also Fact 7, which refer to the costs of edges, remains true:

**Theorem 17** Given a vertex \(v_i\), for any pair of vertices \(v_{j'}, v_{j''}\) such that \(j' \leq j''\) we have:

1. If \(v_{j'}, v_{j''} \in FS(v_i)\) then \(c(v_i, v_{j'}) \leq c(v_i, v_{j''})\)
2. If \(v_{j'}, v_{j''} \in BS(v_i)\) then \(c(v_{j'}, v_i) \geq c(v_{j''}, v_i)\)

**Proof:**

Let \((\delta', \lambda')\) and \((\delta'', \lambda'')\) be respectively the labels assigned to the edges connecting \(v_i\) with \(v_{j'}\) and \(v_i\) with \(v_{j''}\). Assume first then \(v_{j'}, v_{j''} \in FS(v_i)\) and that \(S[i : j'']\) has a copy starting in position \(p \leq i\), associated with context \(S^p\). Then also its prefix \(S[i : j']\) has a copy occurring at \(p\) and associated with the same context, which implies that \(\delta' \leq \delta''\). Since trivially \(\lambda' \leq \lambda''\), it follows that \(c(v_i, v_{j'}) \leq c(v_i, v_{j''})\) by the increasing cost property of the encoders.

Now assume that \(v_{j'}, v_{j''} \in BS(v_i)\). Suppose that the copy of \(S[j' : i]\) starts in position \(p \leq j'\), associated with context \(S^p\). Then \(S[j' : i]\) has a copy starting in position \(t = p + j'' - j'\). Notice that contexts \(S^{j'}\) and \(S^{j''}\) share the prefix \(w = S[j' : j'' - 1]^R\), therefore any of the at least \(\delta''\) contexts falling between them in lexicographic order is prefixed by \(w\). If we remove such prefix from one of these contexts, the remainder is a context falling between \(S^{j'}\) and \(S^p\) in lexicographic order. As a consequence, there are at least \(\delta''\) contexts between \(S^{j'}\) and \(S^p\), which implies that \(\delta'' \geq \delta''\) and hence, by the increasing cost property of the encoder, \(c(v_{j''}, v_i) \leq c(v_{j'}, v_i)\).

Since both Facts 6 and 7 can be restated unchanged for \(G(S)\), we can reuse identically the same pruning strategy described in Section 3.4. Basically, we borrow the notion of maximal edges (Definition 8) and say that an edge \((v_i, v_j)\) is maximal when either \(\left| f(|\delta_{i,j}|) \right| < \left| f(|\delta_{i,j+1}|) \right|\) (also called \(\delta\)-maximal) or \(\left| g(\lambda_{i,j}) \right| < \left| g(\lambda_{i,j+1}) \right|\) (also called \(\lambda\)-maximal). The statements of Lemma 9 and Theorem 10 hold by the same proof; hence, as inferred in Section 3.4, computing the bit-optimal parsing of
CHAPTER 4. EXTENSIONS

S reduces to finding a shortest path from \( v_1 \) to \( v_{n+1} \) in \( \tilde{G}(S) \), which is the subgraph of \( G(S) \) consisting of maximal edges only. In the following, we address the latter problem using the same approach used for LZ77 that is, instead of storing \( \tilde{G}(S) \) in memory, which would take \( \Omega(n) \) space, we dynamically generate the edges of \( \tilde{G}(S) \) as they are inspected in the shortest path computation; we refer this task as the FSG (Forward Star generation) problem. The final result that we will achieve is a bit-optimal parser for ACB working in optimal \( O(n) \) space and \( O(n(Q(f, n) \log n + Q(g, n))) \) time (Theorem 18).

4.2.2 Generating Maximal Edges Efficiently

In this section we show how to dynamically generate the maximal edges outgoing from the current vertex, as the sequence of vertices is scanned by increasing text position. As observed in the LZ77 case, computing the \( \ell \)-maximal edges outgoing from the current vertex trivially requires \( O(1) \) time per edge, since the lengths of \( \ell \)-maximal edges are fixed by the encoding function \( g \). We will therefore concentrate on the computation of \( \delta \)-maximal edges. In subsection 4.2.3, we will describe an auxiliary data structure built on the input text and requiring \( O(n) \) space that, given two indices \( i \) and \( j \), can efficiently evaluate the \( \delta \)-cost of edge \((v_i, v_j)\), or return \( \infty \) if \( v_i \) and \( v_j \) are not linked by an edge. Let us call such operation a \( \delta \)-cost query. Using this tool, the computation of \( \delta \)-maximal can be easily performed incrementally with the following technique.

Adopting the notation of Section 3.4, we denote by \( I_1, I_2, \ldots, I_{Q(f,n)} \) the cost classes for function \( f \) and by \( c(I_k) \) the encoding cost of class \( I_k \). Assume that \( v_i \) is the currently inspected vertex. We show how to maintain the set of indices \( M = \{m_1, m_2, \ldots, m_{Q(f,n)}\} \) with the property that, for any \( k \), \( m_k \) is the largest possible index such that \( c(v_i, v_{m_k}) = c(I_k) \). Notice that this gives a solution to the FSG problem, in fact if there exists a \( \delta \)-maximal edge \((v_i, v_t)\) of cost \( c(I_k) \) then, by definition of maximality, it must be \( t = m_k \). When the scan advances to vertex \( v_{i+1} \) we have to update each index \( m_k \) of \( M \). By Theorem 17 we know that the \( \delta \)-costs of edges outgoing from \( v_{i+1} \) are monotonically increasing in their lengths, therefore the value of \( m_k \) can be updated by simply incrementing it until we find the largest index \( m_k \) such that the \( \delta \)-cost of \((v_{i+1}, v_{m_k})\) is \( c(I_k) \), where the latter condition is checked by querying the auxiliary data structure. Overall this procedure makes at most \( O(n) \) \( \delta \)-cost queries for any cost class of \( f \). Notice that this technique has quite general applicability, since it only requires the parsing graph to satisfy the monotonicity property of Theorem 17 (we will see another example of its usage in Chapter 6).

In the next subsection we present a linear-space data structure supporting \( \delta \)-cost queries in \( O(\log n) \) time which can be built in \( O(n \log n) \) time. This will immediately yield the analogous of Theorem 11 for ACB:
4.2. BIT-OPTIMAL ACB COMPRESSION

**Theorem 18** Given a string $S$ and two integer-encoding functions $f$ and $g$ that satisfy Property 1, the $(f,g)$-optimal parsing of $S$ for the ACB dictionary can be computed in $O(n(Q(f, n)\log n + Q(g, n)))$ time and $O(n)$ space in the worst case.

**Proof:** The time complexity of the above scheme is dominated by the $O(nQ(f, n))$ call to $\delta$-cost query, plus $O(nQ(g, n))$ total time for generating $\lambda$-maximal edge. Therefore, using the data structure of subsection 4.2.3 yields an algorithm for the FSG-problem working in $O(n(Q(f, n)\log n + Q(g, n)))$ time and $O(n)$ space. We can then compute a shortest path in $\tilde{G}(S)$ in the same asymptotical time and space by scanning the sequence of vertices and relaxing the edges outgoing from the current vertex, which are dynamically generated using the above method.

4.2.3 Fast Encoding of Phrases

To complete the proof of Theorem 18 we will now describe a data structure to support $\delta$-cost queries in $O(\log n)$ time and requiring $O(n)$ words of storage. Our solution reduces the evaluation of $\delta$-costs to a particular geometric query over a set of points, which can be solved using well-known computational geometric tools.

Given a set of points $S \subset N \times N$, a two-dimensional range-counting query over $S$ asks to find the number of points in $S$ contained in a rectangle $[x, x'] \times [y, y']$, where $x, x, y, y'$ are integer parameters of the query. In [62], the authors present a data structure supporting two-dimensional range-counting queries over a set of $n$ points in $O(\log n)$ time. This data structure can be built in $O(n \log n)$ time and requires $O(n)$ words of memory. As usual, we denote with $\prec$ the lexicographic ordering of string.

Let $PA[1, n]$ be the array containing the sequence of contexts of $S$ sorted in lexicographic order, i.e. $S^{PA[1]} \prec S^{PA[2]} \prec \ldots \prec S^{PA[n]}$. We build a static data structure over the input text $S$ consisting of the following components:

- (1) The array $IPA[1, n]$ such that $IPA[i] = j$ iff $S^{IPA[i]}$ appears in the $j$-th element of $PA[1, n]$ (i.e. the inverse array of $PA[1, n]$)

- (2) The arrays $P[1, n], R[1, n]$ such that, for any $i$, $P[i]$ and $R[i]$ are respectively the predecessor and successor of $S[1 : i]^R = S^{i+1}$ in lexicographic order among the set of contexts $\{S^j \mid j \leq i\}$

- (3) The data structure of [62] for two-dimensional range-counting built over the set of points $C = \{(i, PA[i]) \mid 1 \leq i \leq n\}$

The space occupancy of components (1), (2) and (3) is $O(n)$ memory words. To build the data structure we first construct $PA[1, n]$, which is trivially derived from the suffix array of $S^R$, then we can build each component in total $O(n \log n)$ time: (1) is constructed in $O(n)$ time by inverting $PA[1, n]$, (2) can be easily built in $O(n \log n)$ time by inserting the elements of $IPA[1, n]$ in a Balanced Search-Tree
and hence we have proved that our data structure can support \( \delta \) queries in \( O(n \log n) \) time (see [62] for details).

Suppose that we are looking for the \( \delta \)-cost of edge \((v_i, v_j)\). The string \( S[i : j] \) can occur in many different places inside \( S[1 : i] \). We refer the contexts that precede each previous occurrence of \( S[i : j] \) as candidate contexts. The value of \( \delta_{i,j} \) may encode the distance from the current context \( S^j \) to any of the candidate contexts and we need to select among them the one that minimizes \( \delta_{i,j} \), i.e.: the closest one to \( S^i \) in lexicographic order, which we will call the reference context. Therefore we have to face two main issues: (i) determine the reference context and (ii) given the reference context, compute the corresponding value of \( \delta_{i,j} \). Point (ii) is easily reduced to a range-counting query over the set \( C \), which can be resolved in \( O(\log n) \) time using the data structure of (3). In fact, assume that we have selected \( S^i \) as reference context for the copy of \( S[i : j] \). The value of \( \delta_{i,j} \) is equal the number of contexts \( S^h \) such that \( h < i \) and \( S^i \prec S^h \prec S^j \) which corresponds to the number of points of \( C \) contained in the rectangle \([IPA[i], IPA[j]] \times [0, i] \).

We are left with showing how to efficiently determine the reference context. Fortunately, this task is made simple by the following:

**Fact 19** If \( S^i \) is the reference context for \( S[i : j] \) then either \( P[j] = t + (j - i) \) or \( S[j] = t + (j - i) \)

**Proof:** Put \( u = t + (j - i) \), we will prove that \( S^u \) is the predecessor or successor of \( S^j \) in lexicographic order among the set of contexts \( \{S^h | h < j\} \) if respectively \( S^u \prec S^j \) or \( S^u \succ S^j \); the thesis then follows by the definitions of \( P[1, n] \) and \( R[1, n] \).

Let us consider the case \( S^u \prec S^j \) (the case \( S^u \succ S^j \) is symmetric). Notice that \( S^u \) and \( S^j \) are both prefixed by \( w = S[i : j]^R \). By absurd, if \( S^q \) is another context with \( q < j \) such that \( S^u \prec S^q \prec S^j \) then \( S^q \) must be prefixed by \( w \) and the relative order of the contexts remains the same if we delete their common prefix \( w \), thus \( S^j \prec S^{(q-j+i)} \prec S^i \). This contradict the fact that \( S^j \) is the reference context, in fact \( S^{(q-j+i)} \) is a candidate context (it is followed by \( S[i : j] \)) and is closer than \( S^q \) to \( S^i \) in lexicographic order. This means that \( S^q \) cannot exists, thus \( S^u \) is the predecessor of \( S^j \) in lexicographic order among the set of contexts \( \{S^h | h < j\} \).

Thanks to Fact 19, to determine the reference context we only need to search among the two contexts \( S^{t_1} \) and \( S^{t_2} \), where \( t_1 = P[j] - j + i \) and \( t_2 = P[j] - j + i \). Therefore we compute the corresponding value of \( \delta_{i,j} \) for both of them, as described in the solution for point (ii), and return as reference context the one giving the minimum; all these operations can be done in \( O(\log n) \) time. In conclusion, combining the solutions for points (i) and (ii), once the reference context and the corresponding \( \delta_{i,j} \) have been determined, the \( \delta \)-cost of edge \((v_i, v_j)\) can be obtained as \(|f(\delta_{i,j})|\), and hence we have proved that our data structure can support \( \delta \)-cost queries in \( O(\log n) \) time.
4.3 Bit-Optimal Greedy Parsing

In this section we describe how to efficiently determine the codeword of smallest encoding for each phrase computed by a greedy LZ77-parser, thus addressing question 2 of Section 4.1. This result has been published in [35].

Let us consider the greedy LZ77-parser with unbounded window (see Section 2.2), and assume that we encode the distance and length components of each codeword using respectively two integer encoders, denoted by $f$ and $g$, which satisfy the Increasing Cost Property (possibly $f = g$). We denote by $\text{LZ}_{f,g}(S)$ the binary encoding of the greedy LZ77-parsing of $S$. Thus, in $\text{LZ}_{f,g}(S)$ any phrase $w_i$ is encoded in $|f(d_i)| + |g(\ell_i)|$ bits. Given that the parsing is the greedy one, $\ell_i$ is in some sense fixed (being the length of the longest copy), so we minimize $|\text{LZ}_{f,g}(S)|$ by minimizing the distance $d_i$ of $w_i$’s copy in $S$. If $p_i$ is the starting position of $w_i$ in $S$ (namely $S[p_i, p_i + \ell_i - 1] = w_i$), many copies of the phrase $w_i$ could be present in $S[1, p_i - 1]$. To minimize $|\text{LZ}_{f,g}(S)|$ we should choose the copy which is the closest one to $p_i$, and thus requires the minimum number of bits to encode its distance $d_i$ (recall the unbounded window assumption). Surprisingly enough, known implementations of greedy parsers are time optimal but not bit-optimal, because they select an arbitrary or the leftmost occurrence of the longest copied phrase (see [22] and references therein), or they select the closest copy but take $O(n \log n)$ suboptimal time [3, 61]. In this section we propose an elegant, yet simple, algorithm which computes at each parsing step the closest copy of the longest dictionary phrase in $O(n(1 + \log \log n))$ overall time and $O(n)$ space (Theorem 20). This is optimal in terms of time/space performance when the alphabet has size $\text{polylog}(n)$ (hence almost all texts of practical interest).

Our solution requires the suffix tree $\mathcal{ST}(S)$ (abbreviated $\mathcal{ST}$), preprocessing to support constant-time $\text{lca}$-queries, and the greedy LZ77-parsing of $S$ which consists of, say, $k \leq n$ phrases. These two objects can be both constructed in linear time from $S$ (see Sections 2.2 and Theorem 2). We say that a node $u$ of $\mathcal{ST}$ is marked iff the string spelled out by the root-to-$u$ path in $\mathcal{ST}$ is equal to some phrase $w_i$. In this case we use the notation $u_{w_i}$ to denote the node marked by phrase $w_i$ which starts at position $p_i$ of $S$. Since the same node may be marked by different phrases, but any phrase marks just one node, the total number of marked nodes is bounded by the number of phrases, hence $k$. Furthermore, if a node is assigned with many phrases, since the greedy LZ77-parsing takes the longest one, it must be the case that every such occurrences of $w_i$ is followed by a distinct character. So the number of phrases assigned to the same marked node is bounded by $\sigma$.

All marked nodes can be computed in $O(k)$ time by executing $k$ $\text{lca}$-queries on $\mathcal{ST}$. Let us now define $\mathcal{ST}_C$ as the contracted version of $\mathcal{ST}$, namely a tree whose internal nodes are the marked nodes of $\mathcal{ST}$ and whose leaves are the leaves of $\mathcal{ST}$. The parent of any node in $\mathcal{ST}_C$ is its lowest marked ancestor in $\mathcal{ST}$. It is easy to see that $\mathcal{ST}_C$ consists of $O(k)$ internal nodes and $n$ leaves, and that it can be built in $O(n)$ time via a top-down visit of $\mathcal{ST}$.
Given the properties of suffix trees, we can now rephrase our problem as follows:
for each position \( p_i \), we need to compute the largest position \( x_i \) which is smaller
than \( p_i \) and whose leaf in \( ST_C \) lies within the subtree rooted at \( u_{p_i} \). Our algorithm
processes the input string \( S \) from left to right and, at each position \( j \), it maintains
the following invariant: the parent \( v \) of any leaf in \( ST_C \) stores the maximum position
\( h < j \) such that the leaf labeled \( h \) is attached to \( v \). Maintaining this invariant is
trivial: after that position \( j \) is processed, \( j \) is assigned to the leaf parent of the
leaf labeled \( j \) in \( ST_C \). The key point now is how to compute the position \( x_i \) of
the rightmost-copy of \( w_i \) whenever we discover that \( j \) is the starting position of a
phrase (i.e. \( j = p_i \) for some \( i \)). In this case, the algorithm visits the subforest of
\( ST_C \) rooted at \( u_j \) and computes the maximum position stored in its internal nodes.
By the invariant, this position is the rightmost copy of the phrase \( w_i \). This process
takes \( O(n + \sigma \sum_{i=1}^{k} #(u_{p_i})) \) time, where \( #(u_{p_i}) \) is the number of internal nodes in
the subrooted at \( u_{p_i} \) in \( ST_C \). In fact, by construction, there can be at most \( \sigma \)
repetitions of the same phrase in the parsing of \( S \), and for each of them the algorithm
performs a visit of the corresponding subtree.

As a final step we prove that \( \sum_{i=1}^{k} #(u_{p_i}) = O(n) \). By properties of suffix trees,
the depth of \( u_{p_i} \) is smaller than \( \ell_i = |w_i| \), and each (marked) node of \( ST_C \) is visited
as many times as the number of its (marked) ancestors in \( ST_C \) (with their multiplicities).
For each (marked) node \( u_{p_i} \), this number can be bounded by \( \ell_i = O(|w_i|) \).
Summing up on all nodes, we get \( \sum_{i=1}^{k} O(|w_i|) = O(n) \). Thus, the above algorithm
requires \( O(\sigma \times n) \) time, which is trivially optimal whenever \( \sigma = O(1) \).

Now we will show how to further reduce the time complexity to \( O(n(1 + \log \sigma / \log \log n)) \)
by properly combining a slightly modified variant of the tree covering procedure of
[40] with a dynamic Range Maximum Query data structure [70, 91] applied on properly composed arrays of integers. Notice that this improvement leads to an algorithm
requiring optimal \( O(n) \) time for alphabets of size poly-logarithmic in \( n \).

Given \( ST_C \) and an integer parameter \( P \geq 2 \) (in our case \( P = \sigma \)) this procedure
covers the \( k \) internal nodes of \( ST_C \) in a number of connected subtrees, all of which
have size \( \Theta(P) \), except the one which contains the root of \( ST_C \) that has size \( O(P) \).
Any two of these subtrees are either disjoint or intersect at their common root. (We refer to Section 2 of [40] for more details.) In our modification we impose that there
is no node in common to two subtrees, because we move their common root to the
subtree that contains its parent. None of the above properties change, except for
the fact that each cover could now be a subforest instead of subtree of \( ST_C \). Let
\( F_1, F_2, \ldots F_t \) be the subforests obtained by the above covering, where we clearly have
that \( t = O(k/P) \).

We define the tree \( ST_{SC} \) whose leaves are the leaves of \( ST_C \) and whose internal
nodes are the above subforests. With a little abuse of notation, let us refer with
\( F_i \) to the node in \( ST_{SC} \) corresponding to the subforest \( F_i \). The leaf \( l \) having \( u \) as
parent in \( ST_C \), is thus connected to the node \( F_i \) in \( ST_{SC} \), where \( F_i \) is the forest that
contains the node \( u \). Notice that roots of subtrees in any subforest \( F_i \) have common
parent in $\mathcal{ST}_C$.

The computation of the rightmost copy for a phrase $p_i$ is now divided in two phases. Let $F_i$ be the subforest that contains $u_{p_i}$, the node spelled out by the phrase starting at $S[p_i]$. In the first phase, we compute the rightmost copy for the phrase starting at $p_i$ among the descendants of $u_{p_i}$ in $\mathcal{ST}_{SC}$ that belong to subforests different from $F_i$. In the second phase, we compute its rightmost copy among the descendants of $u_{p_i}$ in $F_i$. The maximum between these two values will give the rightmost copy for $p_i$, of course. To solve the former problem, we execute our previous algorithm on $\mathcal{ST}_{SC}$. It simply visits all subforests descendant from $F_i$ in $\mathcal{ST}_{SC}$, each of them maintaining the rightmost position among its already scanned leaves, and returns the maximum of these value. Since groups of $P = \sigma$ nodes of $\mathcal{ST}_C$ have single nodes in $\mathcal{ST}_{SC}$, in this case our previous algorithm requires $O(n)$ time.

The latter problem is solved with a new algorithm exploiting the fact that the number of nodes in $F_i$ is $O(\sigma)$ and resorting to dynamic Range Maximum Queries (RMQ) on properly defined arrays [70]. To be precise, we assign to each node of $F_i$ an unique identifier in $[m]$ that corresponds to the time of its visit in a depth-first traversal of $F_i$. Notice that the nodes in the subtree rooted at some node $u$ receive integers spanning the whole range from the starting time to the ending time of the DFS-visit of $u$. We use an array $A_{F_i}$ that has an entry for each node of $F_i$. Initially, all entries are set to $-\infty$. The entry corresponding to any node has index equal to the time of its visit. We build on each array $A_{F_i}$ a dynamic data structure that answers range maximum queries. For this purpose we use a simple balanced tree augmented with the maximum of the descending leaves in each nodes of $F_i$. This way Range-Max queries and updates on $A_{F_i}$ take $O(\log \sigma)$ time in the worst case. Now, we proceed, as in our first algorithm, by processing string $S$ from left to right. When a position $j$ of $S$ is processed, we identify the subforest $F_i$ containing the father of the leaf labeled $j$ in $\mathcal{ST}_C$ and we set to $j$ the corresponding entry in $A_{F_i}$ (this induces a change in the underlying RMQ data structure). If $j$ is the starting position of a phrase, we identify the subforest $F_x$ containing the node $u_j$ and compute its rightmost copy in $F_x$, by resorting to a RMQ on $A_{F_x}$. The left and right indexes for the range query are, respectively, the starting and ending time of the visit of $u_j$ in $F_x$.

It is easy to notice that the overall complexity of the algorithm is dominated by the $O(n)$ updates to the RMQ data structures and the $O(k)$ queries onto them (recall that $k$ is the number of phrases of the LZ77-greedy parsing of $S$). Our algorithm then takes $O(n \log \sigma)$ time and $O(n)$ space. A further improvement can be obtained by adopting an idea similar to the one in [91][Section 5] to reduce the height of that balanced tree and, consequently, our time complexity by a factor $O(\log \log n)$. This proves the following

**Theorem 20** ([35]) *Given a string $S$, there exists an algorithm that computes the greedy parsing of $S$ and reports the rightmost copy of each phrase in the LZ77-dictionary taking $O(n(1 + \frac{\log \sigma}{\log \log n}))$ time and $O(n)$ space.*
Part III

Reorganizing Data for Compression
Chapter 5
The PPC-paradigm

Reorganizing data to improve compressibility is an important principle for the design of lossless compression algorithms. The reorganization process generally consists in permuting the input data to form a new string which is then partitioned into a sequence of contiguous substrings that are finally compressed individually by a given base compressor. We call this scheme the Permute-Partition-Compress paradigm, or PPC-paradigm for brevity. This paradigm and its instantiations have received considerable attention in the data compression community, its virtues have been exploited in many different areas of compression: like Table Compression, compression of massive collections of texts, and in compression based on the Burrows-Wheeler Transform (Sections 5.2,5.3,5.4).

This chapter serves as an introduction to the PPC-paradigm. We will briefly survey the above cited applications, recognizing the central role of parsing algorithms in each of them. In this setting, the goal of the parser is to compute a partition of the permuted data resulting in the best possible final compression. This provides motivation to study the optimal partitioning problem which will be the main topic of investigation of Chapter 6.

5.1 Theory and Practice of PPC

The PPC-paradigm is a general scheme for the design of compression algorithms. It is based on the idea of reorganizing input data in a form which is more convenient for a given base compressor. In general, given a base compressor $C$, an algorithm that follows the PPC-paradigm compress the input string $S$ in three consecutive steps:

1. Rearrange the symbols of $S$ to form a new string $S'$ (Permuting)

2. Compute a partition of the transformed string $S'$ into a sequence of contiguous substrings say $S' = S'_1 S'_2 \cdots S'_k$ (Partitioning)
3. Individually compress the substrings so found with $C$, and return the concatenation as the compressed output $C(S'_1)C(S'_2)\cdots C(S'_k)$ (Compression)

Notice that the PPC is clearly at least as powerful as the classic data compression approach that applies $C$ to the entire $S$: just take the identity permutation and set $k = 1$. Intuitively, “good” algorithms for the Permuting and Partitioning steps should maximize the redundancy inside each substring of the partition by grouping together similar pieces of input. This ensures that the resulting sequence of substrings will be highly compressible, even using a very weak compressor $C$.

The potential of this approach has been highlighted by recent theoretical and experimental results. These results have investigated the PPC-paradigm under various angles by considering: different data formats (strings [31], trees [32], tables [11], etc.), different granularities for the items of $S$ to be permuted (chars, node labels, columns, blocks [6, 59], files [14, 83, 84], etc.), different permutations (see e.g. [41, 88, 84, 14]), different base compressors to be boosted (0th order compressors, gzip, bzip2, etc.).

We survey below some of the most significative applications of the PPC-paradigm to data compression. In all of them, one of the primary goal is finding an instantiation of the permuting and partitioning steps that achieves the best possible compression. Unfortunately, optimizing the permuting step turn out to be a very difficult problem in general, which is usually dealt using heuristic techniques that are outside the scope of this thesis (see for instance the discussion related to Table Compression in Section 5.2). We will focus instead on computing the optimal partition of a string $S$ with respect to a base compressor $C$. This problem, called optimal partitioning (Section 6.1) can be solved in polynomial time using dynamic programming, but the only known solution are very slow (cubic time). In Chapter 6 we will improve this result, showing the first efficient approximation algorithm for this problem.

## 5.2 Table Compression

Tables are large collections of fixed-length records, which in practice can grow to terabytes in size. The goals of table compression are to be fast, online, and effective: eventual compression ratios of at least 100:1 are desirable.

A straightforward method for compressing a table is to compress the string derived from scanning the table in row-major order. Buchsbaum et al. [10] were the first to suggest the use data rearrangement for boosting Table Compression, their approach has been further investigated in [11]. They observed experimentally that partitioning the table into contiguous intervals of columns and compressing each interval separately in this fashion can achieve significant compression improvement.

They also observed heuristically that certain rearrangements of the columns prior to partitioning further improve compression by grouping dependent columns more closely. For example, in a table of addresses and phone numbers, the area code can
often be predicted by the zip code when both are defined geographically therefore, by arranging the corresponding columns consecutively, we give the base compressor the possibility to exploit this correlation.

To better illustrate why permuting and partitioning may improve compression, let us consider the example of Figure 5.1. Observe that the first three columns of the table, taken in row-major order, form a repetitive string that can be very easily compressed and the same is true for the last two columns. Therefore it is advantageous to compress the first three and the last two columns separately.

Buchsbaum et al. [10] raised the problem of permuting and partitioning the columns optimally, in the sense that individually compressing contiguous groups of them gives the shortest compressed output. Unfortunately, computing the most compressible partition with reordering of the columns seems intractable. No polynomial algorithms are known for it, and some of its variants are known to be MAX-SNP-hard [11]. If we restrict to partitioning alone (without reordering of the columns), then there exists a simple polynomial time solution [10], based on Dynamic Programming. The authors showed that optimal partitioning without column rearrangement can improve gzip compression by a factor 2 on telephone usage data. However, this approach has cubic time complexity and therefore is unfeasible even on small input files. Successively, Giancarlo et al. [11] devised a linear time greedy algorithm which however did not have any guaranteed bounds on the quality of the produced result. Concerning the rearrangement process, the authors of [11] proposed efficient heuristics, based on a reduction to the classic TSP problem. They empirically showed that good TSP heuristics can effectively reorder the columns, yielding additional improvements of 5 to 20 relative to partitioning alone.

5.3 Text Compression and the Burrows-Wheeler Transform

5.3.1 Overview

In the context of classic text compression, the most notable application of the PPC-paradigm is the design of compression boosters [31, 29, 42]. Informally, a boosting technique is a method that, when applied to a particular class of algorithms, yields
improved algorithms. The problem of compression boosting consists of designing a technique that improves the compression performance of a class of algorithms.

The works of [42] first related the question of compression boosting to the PPC-paradigm. Their approach strongly relies on the ability of the Burrows-Wheeler Transform to arrange consecutively symbols of the input text preceded by the same context. At high level, their technique is an instantiation of the PPC paradigm in which the permuting step is implemented by the BWT. The net result is a tool for boosting the performance of the base compressor \( C \) from 0th order-entropy bounds to \( k \)th order entropy bounds, simultaneously over all \( k \geq 0 \). In a sense, this approach allow to turn a “memoryless” base compressor that uses no context information into one that always select the best possible context.

Technically speaking, the new algorithm uses the starting compressor \( C \) as a black box and operates by partitioning the BW-transformed text into contiguous substrings which are then separately compressed with \( C \). The time/space complexity of the decompression process remains essentially the same, since the only additional work required is for inverting the BWT, which can be done in \( O(n) \) time and space.

From a theoretical perspective, this boosting technique offers two additional advantages: 1) when applied to good 0th order base compressors (namely Huffman or Arithmetic) it yields compressors that are theoretically superior, in terms of entropic bounds, to some of the best existing ones, such as \( LZ77, LZ78 \), and the ones derived from the Burrows-Wheeler Transform 2) the theoretical analysis of the new compressor is straightforward.

On the other hand, the main bottleneck of the approach in [42] is that it requires to compute an optimal partition of the BW-transformed text. They proposed a solution based on dynamic programming requiring \( O(n^2) \) time, which makes their results of purely theoretical interest. Successively, Ferragina et al. [31] observed that optimal partitioning can be replaced by a context-based grouping of that, although not optimal, achieve the same worst-case bound on the compress size. They proved that such partitioning of the \( \text{BWT} \) can be computed by a simple greedy algorithm working in \( O(n) \) time, assuming an alphabet of constant size (see Theorem 22).

### 5.3.2 Compression Boosting and the PPC-paradigm

Compression boosting is made possible by the following important property that relates Burrows-Wheeler Transform with \( k \)th order empirical entropy \( H_k \):

**Property 2 ([42])** If there existed an ideal compressor \( A \), such that, for any parsing \( S = S_1S_2 \ldots S_p \) of a string \( S \), one have:

\[
|A(S)| = \sum_{i=1}^{p} |S_i|H_0(S_i)
\]

then for any string \( S \):
5.3. TEXT COMPRESSION AND THE BURROWS-WHEELER TRANSFORM

\[ |A(BWT(S))| = |S|H_k(S) \]

holds simultaneously for any value of \( k \).

That is, \( A(BWT(s)) \) would achieve, exactly, the best possible performance, simultaneously for all \( k \). Property 2 suggests how to combine the BWT transform with a partitioning technique, to obtain a new class of compression methods based on the BWT transform. We assume that \( C \) is a base compressor such that, when compress in input a string \( x \), it leaves a special symbol $ (not in \Sigma) to the end if the string to tell the decompressor when to stop the decoding process. Many data compression algorithms can be adapted to work in this way.

Consider the following data compression algorithm based on the PPC-paradigm:

(A) Compute \( \hat{S} = BWT(S) \);
(B) Optimally partition \( \hat{S} \) with respect to the base compressor \( C \);
(C) Compress, separately, each piece of the partition and concatenate the results in output.

We denote such an algorithm as \( BWT^C_{OPT} \). The following theorem [42] is a consequence of property 2:

**Theorem 21 ([42])** Let \( C \) be a base compressor satisfying the assumptions stated above and such that \( |C(x)| \leq \lambda|x|H_0(x) + \mu|x| + c \), where \( \lambda, \mu \) and \( c \) are non-negative constants. Then, given a string \( S \), for all \( k > 0 \), the output-size in bits of compressor \( BWT^C_{OPT} \) on \( S \) is bounded by \( \lambda|S|H_k(S) + \mu|S| + O(c|\Sigma|^k) \).

In particular, when \( C \) is the classic Huffman encoding we have \( \lambda = \mu = 1 \) and \( c = |\Sigma|\log |\Sigma| \) and therefore \( |BWT^C_{OPT}(S)| \leq |S|H_k(S) + |S| + S(|\Sigma|^{k+1} \log \Sigma) \). Theorem 21 states that optimally partitioning the Burrows-Wheeler Transform yields a general boosting method for a base compressor \( C \). Starting from these premises [42] attacked the computation of the optimal partitioning of \( S \) via a Dynamic Programming approach, which turned to be very costly (at least \( O(n^2) \)); then [31] (and subsequently many other authors, see e.g. [30, 64, 32]) proposed partitioning techniques which are not optimal but, nonetheless, achieve the same \( k \)th order-entropy bounds of Theorem 21.

The main idea of [31] is to restrict the attention on a particular subclass of the partitions of \( BWT(S) \). A subset \( \mathcal{L} \) of the nodes of a tree is called a leaf cover if every leaf of the tree has a unique ancestor in \( \mathcal{L} \). Any leaf cover \( \mathcal{L} = \{u_1, u_2, \ldots, u_p\} \) of the suffix tree of \( S \) naturally induces a partition of \( BWT(S) \) into \( p \) blocks, where each block contains all the symbol of \( S \) having as context the locus of \( w \), for some node \( w \in \mathcal{L} \) (we refer to [31] for more details). The authors of [31] proved that:
(i) The leaf cover whose corresponding induced partition has minimum cost can be computed in time $O(n\sigma)$, with a simple greedy algorithm;

(ii) If the partition induced by this leaf cover is used in $BWT_{OPT}$ in place of the optimal one, then the final compressed output respects the same entropy-bounds of Theorem 21.

Let us call $F_{OPT}$ the algorithm obtained by substituting in $BWT_{C_{OPT}}$ the slow DP-based approach with the algorithm mentioned in point (i). Then, exploiting points (i) and (ii), it is possible to prove that:

**Theorem 22 ([31])** Let $C$ be a base compressor compressing any string $x$ in at most $|C(x)| \leq \lambda|x|H_0(x) + \mu|x| + c$, where $\mu$ and $c$ are non-negative constants. Then, given a string $S$, for all $k > 0$, the number of bits produced by $F_{OPT}$ on $S$ is bounded by $|S|H_k(S) + \mu|S| + O(c|\Sigma|^k)$. The new algorithm uses additional $O(|S| \log |S|)$ bits of space and introduces $O(n\sigma)$ overhead over the running time of $C$.

### 5.4 Space-Constrained Compression

There is another scenario in which the PPC-paradigm is successfully deployed to improve the compressibility of an input string and occurs when $S$ is a single (possibly long) file on which we wish to apply classic data compressors, such as gzip, bzip2, ppm, etc. [92]. Note that how much redundancy can be detected and exploited by these compressors depends on their ability to “look back” at the previously seen data. However, such ability has a cost in terms of memory usage and running time, and thus most compression systems provide a facility that controls the amount of data that may be processed at once — usually as a block size. For example the classic tools gzip and bzip2 have been designed to have a small memory footprint, from few tens to some hundreds KBs. More recent and sophisticated compressors, like ppm [92] and the family of BWT-based compressors [29], have been designed to use block sizes of up to a few hundreds MBs. But using larger blocks to be compressed at once does not necessarily induce a better compression ratio!

As an example, let us take $C$ as the simple Huffman or Arithmetic coders and use them to compress the string $S = 0^n1^n/2$: There is a clear difference whether we compress individually the two halves of $S$ (achieving an output size of about $O(\log n)$ bits) or we compress $S$ as a whole (achieving $n + O(\log n)$ bits). The impact of the block size is even more significant as we use more powerful compressors, such as the $k$th order entropy encoder ppm which compresses each symbol according to its preceding $k$-long context. In this case take $S = (2^k0)^{n/(2^k+1)}(2^k1)^{n/(2^k+1)}$ and observe that if we divide $S$ in two halves and compress them individually, the output size is about $O(\log n)$ bits, but if we compress the entire $S$ at once then the output size turns to be much longer, i.e. $2^kk + 0(\log n)$ bits. Therefore the choice of the block size cannot be underestimated and, additionally, it is made even more...
problematic by the fact that it is not necessarily the same along the whole file we are compressing because it depends on the distribution of the repetitions within it.

This problem is even more challenging when $S$ is obtained by concatenating a collection of files via any permutation of them: think to the serialization induced by the Unix tar command, or other more sophisticated heuristics like the ones discussed in [83, 14, 74, 84]. In these cases, the partitioning step looks for homogeneous groups of contiguous files which can be effectively compressed together by the base-compressor $C$. More than before, taking the largest memory-footprint offered by $C$ to group the files and compress them at once, is not necessarily the best choice because real collections are typically formed by homogeneous groups of dramatically different sizes (e.g. think to a Web collection and its different kinds of pages). In all those cases we are interested in optimally partitioning the input for compression text and, again, this provides additional motivation to study efficient algorithms for this task.
Chapter 6

Fast Approximate Optimal Partitioning

6.1 The Optimal Partitioning Problem

In Chapter 5 we introduced a new emerging paradigm in data compression, called Permute-Partition-Compress or PPC. The basic idea consists of permuting the input data to form a new string which is then partitioned into a sequence of contiguous substrings that are finally compressed individually with a base compressor. It follows from this discussion (see 5.1) that one of the fundamental problems in the design of PPC-based compressor is computing the most compressible partition of the input string. We call it the optimal partitioning problem, formalized as follows.

Let \( C \) be the base compressor we wish to boost, and let \( S \) be the input string we wish to partition and then compress. Notice that we are assuming that \( S \) has been (possibly) permuted in advance, and we are concentrating on the last two steps of the PPC paradigm. Given a partition \( P \) of the input string into contiguous substrings, say \( S = S_1 S_2 \cdots S_k \), we denote by \( \text{Cost}(P) \) the compression cost of this partition and measure it as \( \sum_{i=1}^{k} |C(S_i)| \).

**Problem 1 (optimal partitioning)** For a given input string \( S \), compute the partition \( P_{\text{opt}} \) achieving the minimum compression cost with respect to \( C \), namely

\[
P_{\text{opt}} = \min_P \text{Cost}(P)
\]

The optimal partition \( P_{\text{opt}} \) can be easily computed via Dynamic-Programming [11, 43]. Define \( E[i] \) as the cost of the optimum partitioning of \( S[1 : i] \), and set \( E[0] = 0 \). Then, for each \( i \geq 1 \), we can compute \( E[i] \) as the \( \min_{0 \leq j \leq i-1} E[j] + |C(S[j + 1 : i])| \). At the end \( E[n] \) gives the cost of \( P_{\text{opt}} \), which can be explicitly determined by standard back-tracking over the DP-array. Unfortunately, this solution requires to run \( C \) over \( \Theta(n^2) \) substrings of average length \( \Theta(n) \), for an overall \( \Theta(n^3) \) time cost which is clearly unfeasible even on small input sizes \( n \).
The main result presented in this chapter is the first efficient approximation scheme for the optimal partitioning problem \[34\]. This chapter is organized as follows. Section 6.2 presents a general overview of our approximation scheme, which deploy the notion of entropy-based compressive estimates introduced in Section 6.3. In the subsequent sections we detail our approximation algorithms, which follows the solution pattern described in Chapter 6.1. First, Section 6.4 describes our reduction from the optimal partitioning problem of \( S \) to a shortest-path problem over a weighted DAG in which edges represent substrings of \( S \) and edge costs are entropy-based estimations for the compression of these substrings via \( C \). Then, Sections 6.5 will address the problem of incrementally and efficiently computing those edge costs as they are needed by the shortest-path computation, distinguishing the two cases of 0th order estimators (Section 6.5.2) and \( k \)th order estimators (subsection 6.5.3), and the situation in which \( C \) is a BWT-based compressor and \( S \) is a collection of files (Section 6.6).

6.2 An Approximation Scheme

The key idea to improve the running time of the exact DP-based approach 6.1 is to allow approximate results. To this aim, we make two significant relaxation to the original problem:

(1) Instead of applying \( C \) over each substring of \( S \), we use an entropy-based estimation of \( C \)’s compressed output (described in Section 6.3) that can be computed efficiently and incrementally by suitable dynamic data structures;

(2) We relax the requirement for an exact solution to the optimal partitioning problem, and aim at finding a partition whose cost is no more than \((1 + \epsilon)\) worse than \( P_{\text{opt}} \), where \( \epsilon \) is a user-defined positive constant.

Some comments about items (1) and (2) are in order. Item (1) takes inspiration from the heuristics proposed in [10, 11], but it is executed in a more principled way because our entropy-based cost functions reflect the real behavior of modern compressors, and our dynamic data structures allow the efficient estimation of those costs without their re-computation from scratch at each substring (as instead occurred in [10, 11]).

Item (2) reduces the problem to efficiently computing a partition whose cost is a \((1 + \epsilon)\)-approximation of the optimal partition’s cost \( P_{\text{opt}} \), where \( \epsilon \) is a user-defined positive constant. In what follow, we call \((1 + \epsilon)\)-approximation of a value \( x \), for a given \( \epsilon > 0 \), any value \( \tilde{x} \) such that \( x \leq \tilde{x} \leq (1 + \epsilon)x \). Notice that by arbitrarily setting the value of \( \epsilon \) the user can control the distance of the algorithm’s output from optimality.

We show that computing (an approximation of) the optimal partition boils down to a Single Source Shortest path computation over a weighted DAG consisting of \( n \)
6.3. COMPRESSIVE ESTIMATES

nodes and $O(n^2)$ edges whose costs are derived from item (1). We prove some interesting structural properties of this graph that allow us to restrict the computation of the shortest-path to a subgraph consisting of $O(n \log_{1+\epsilon} n)$ edges only. The technical part of this chapter (Section 6.5) will show that we can build this graph on-the-fly as the shortest-path computation proceeds over the DAG via the proper use of time-space efficient dynamic data structures. The final result will be to show that we can $(1 + \epsilon)$-approximate $P_{\text{opt}}$ in $O(n \log_{1+\epsilon} n)$ time and $O(n)$ space, for both 0th order (like Huffman and Arithmetic, Section 1.2) and $k$th order compressors (such as ppm [92]). We will also extend these results to the class of BWT-based compressors, when $S$ is a collection of texts. We remark that the result on 0th order entropy is interesting from both the experimental side, since Huffword compressor is the standard choice for the storage of Web pages [92], and from the theoretical side since it can be applied to the compression booster of [31] to fast obtain an approximation of the optimal partition of BWT($S$) in $O(n \log_{1+\epsilon} n)$ time. This may be better than the algorithm of [31] both in time complexity, since that takes $O(n \sigma)$ time, and in compression ratio. The case of a large alphabet (namely, $|\Sigma| = \Omega(\text{poly} \log n)$) is particularly interesting whenever we consider either a word-based BWT [71] or the XBW-transform over labeled trees [31].

6.3 Compressive Estimates

Our approach relies on entropy-based bounds to estimate the output size of base compressor $C$. As discussed in Section 1.2, the empirical entropy can provide reasonable bounds on the performance of many compression algorithms. When $C$ is a $k$th order statistical compressor we can bound the compress size of a string $S$ in terms of $H_0(S)$ (the 0th order empirical entropy of $S$) as:

$$|C_0(S)| \leq \lambda n H_k(S) + f_k(n, \sigma)$$  \hspace{1cm} (6.1)

Here $\lambda$ is a positive constant and $f_k(n, \sigma)$ is a function including the extra costs of encoding the source model and/or other inefficiencies of $C$. In the following we will assume that the function $f_k(n, \sigma)$ can be computed in constant time given $n$ and $\sigma$. As an example, we have seen in Section 1.2, that when $C$ is Huffman we have $k = 0$, $f_0(n, \sigma) = \sigma \log \sigma + n$ bits and $\lambda = 1$, while for Arithmetic coding it is $k = 0, f_0(n, \sigma) = \sigma \log n + O(1)$ bits and $\lambda = 1$.

Many authors have provided upper bounds in terms of $k$th order empirical entropy for sophisticated data-compression algorithms, such as the Lempel-Ziv algorithms (Section 2.4), bzip2 [66, 31, 48], and ppm.
6.4 The Parsing Graph and a Pruning Strategy

The optimal partitioning problem we stated in Section 6.1 can be reduced to a single source shortest path computation (abbreviated SSSP) over a particular directed acyclic graph $G(S)$ which is defined as follows. The graph $G(S)$ has a vertex $v_i$ for each position $i$ in $S$, plus an additional vertex $v_{n+1}$ marking the end of the string, and an edge connecting vertex $v_i$ to vertex $v_j$ for any pair of indices $i$ and $j$ such that $i < j$. Each edge $(v_i, v_j)$ has associated the cost $c(v_i, v_j) = |C(S[i : j - 1])|$ that corresponds to the size in bits of the substring $S[i : j - 1]$ compressed by $C$.

We remark the following crucial, but easy to prove, property of the cost function defined on $G(S)$:

\textbf{Fact 23} For any vertex $v_i$, it is $0 < c(v_i, v_{i+1}) \leq c(v_i, v_{i+2}) \leq \ldots \leq c(v_i, v_{n+1})$

There is a one-to-one correspondence among paths in $G(S)$ from $v_1$ to $v_{n+1}$ and partitions of $S$: every edge $(v_i, v_j)$ in the path identifies a contiguous substring $S[i : j - 1]$ of the corresponding partition. Therefore the cost of a path is equal to the (compression-)cost of the corresponding partition. Thus, we can find the optimal partition of $S$ with respect to $C$ by computing the shortest path in $G(S)$ from $v_1$ to $v_{n+1}$. Unfortunately there are two main drawbacks with this simple approach:

1. the number of edges in $G(S)$ is $\Theta(n^2)$, thus making the shortest-path computation inefficient (i.e. $\Omega(n^2)$ time) if executed directly over $G(S)$;

2. the computation of the each edge cost might take $\Theta(n)$ time over most $S$’s substrings, if $C$ is run on each of them from scratch.

In the following sections we address successfully both these two drawbacks. First, we sensibly reduce the number of edges in the graph $G(S)$ to be examined during the shortest-path computation and show that we can obtain a $(1 + \epsilon)$-approximation using only $O(n \log_{1+\epsilon} n)$ edges, where $\epsilon > 0$ is a user-defined parameter (subsection 6.4.1). Second, we show some sufficient properties that $C$ needs to satisfy in order to compute efficiently every edge’s cost. These properties hold for some well-known compressors—e.g. 0th order compressors, PPM-like and bzip-like compressors—and for them we show how to compute each edge cost in constant or polylogarithmic time (Sections 6.5.2–6.5.3–6.6).

6.4.1 A pruning strategy

The aim of this section is to provide a strategy that reduces the number of edges in the DAG $G(S)$ without increasing the shortest path distance among its leftmost and rightmost nodes, $v_1$ and $v_{n+1}$, by more than a fixed error threshold.

Given an error threshold $\epsilon > 0$, we will denote with $G_\epsilon(S)$ the subgraph produced by our pruning technique which is defined as follows: it contains all edges $(v_i, v_j)$ of $G(S)$, for $1 \leq i < j \leq n + 1$, such that at least one of the following two conditions holds:
1. there exists a positive integer \( k \) such that \( c(v_i, v_j) \leq (1 + \epsilon)^k < c(v_i, v_{j+1}) \);
2. \( j = n + 1 \).

In other words, by fact 23, we are keeping for each integer \( k \) the edge of \( \mathcal{G}(S) \) that approximates at the best the value \((1 + \epsilon)^k\) from below. Given this, we will call \( \epsilon \)-maximal the edges of \( \mathcal{G}_\epsilon(S) \). Clearly, each vertex of \( \mathcal{G}_\epsilon(S) \) has at most \( \log_{1+\epsilon} n = O(\frac{1}{\epsilon} \log n) \) outgoing edges, which are \( \epsilon \)-maximal by definition. Therefore the total size of \( \mathcal{G}_\epsilon(S) \) is at most \( O(\frac{n}{\epsilon} \log n) \). Hereafter, we will denote with \( d_{\mathcal{G}}(\cdot, \cdot) \) the shortest path distance between any two nodes in a graph \( \mathcal{G} \).

The following lemma states a basic property of shortest path distances over our special DAG \( \mathcal{G}(S) \):

**Lemma 24** For any triple of indices \( 1 \leq i \leq j \leq q \leq n + 1 \) we have:

1. \( d_{\mathcal{G}(S)}(v_j, v_q) \leq d_{\mathcal{G}(S)}(v_i, v_q) \)
2. \( d_{\mathcal{G}(S)}(v_i, v_j) \leq d_{\mathcal{G}(S)}(v_i, v_q) \)

**Proof:** We prove just 1, since 2 is symmetric. It suffices by induction to prove the case \( j = i + 1 \). Let \((v_i, w_1)(w_1, w_2)...(w_{h-1}, w_h), \) with \( w_h = v_q \), be a shortest path in \( \mathcal{G}(S) \) from \( v_i \) to \( v_q \). By fact 23, \( c(v_j, w_1) \leq c(v_i, w_1) \) since \( i \leq j \). Therefore the cost of the path \((v_j, w_1)(w_1, w_2)...(w_{h-1}, w_h) \) is at most \( d_{\mathcal{G}(S)}(v_i, v_q) \), which proves the claim.

The correctness of our pruning strategy relies on the following theorem:

**Theorem 25** For any string \( S \), the shortest path in \( \mathcal{G}_\epsilon(S) \) from \( v_1 \) to \( v_{n+1} \) has a total cost of at most \((1 + \epsilon) d_{\mathcal{G}(S)}(v_1, v_{n+1}) \).

**Proof:** We prove a stronger assertion, namely that \( d_{\mathcal{G}_\epsilon(S)}(v_i, v_{n+1}) \leq (1+\epsilon) d_{\mathcal{G}(S)}(v_i, v_{n+1}) \) for any index \( 1 \leq i \leq n + 1 \), which yields the thesis by taking \( i = 1 \). This is clearly true for \( i = n + 1 \), because in that case \( d_{\mathcal{G}_\epsilon(S)}(v_{n+1}, v_{n+1}) = d_{\mathcal{G}(S)}(v_{n+1}, v_{n+1}) = 0 \).

Now let us inductively prove that, for any non negative integer \( k \), if the statement is true for any \( i > k \) then it is true also for \( i = k \). Consider a shortest path \( \pi \) in \( \mathcal{G}(S) \) from \( v_i \) to \( v_{n+1} \) and let \((v_k, v_{t_1})(v_{t_1}, v_{t_2})... (v_{h-1}, v_h) \) be the sequence of edges traversed by \( \pi \), with \( t_h = n + 1 \). By the definition of \( \epsilon \)-maximal edge, it is possible to find an \( \epsilon \)-maximal edge \((v_k, v_r)\) with \( t_1 \leq r \), such that \( c(v_k, v_r) \leq (1 + \epsilon) c(v_k, v_{t_1}) \). By Lemma 24 we know that \( d_{\mathcal{G}(S)}(v_r, v_{n+1}) \leq d_{\mathcal{G}(S)}(v_{t_1}, v_{n+1}) \). By induction, \( d_{\mathcal{G}_\epsilon(S)}(v_r, v_{n+1}) \leq (1 + \epsilon) d_{\mathcal{G}(S)}(v_r, v_{n+1}) \). Combining this with the triangle inequality we get the thesis.
6.5 Space and time efficient algorithms for generating $G_\epsilon(S)$

Theorem 25 ensures that, in order to compute a $(1+\epsilon)$ approximation of the optimal partition of $S$, it suffices to compute the SSSP in $G_\epsilon(S)$ from $v_1$ to $v_{n+1}$. This can be easily computed in $O(|G_\epsilon(S)|) = O(\frac{n}{\epsilon} \log n)$ time since $G_\epsilon(S)$ is a DAG, as a consequence of Fact 1.

However, generating $G_\epsilon(S)$ in efficient time is a non-trivial task for several reasons. First of all, the original graph $G(S)$ contains $\Omega(n^2)$ edges, so that we cannot trivially check each edge $(v_i, v_j)$ to determine whether it is $\epsilon$-maximal or not, because this would take $\Omega(n^2)$ time. Moreover, as pointed out at the beginning of this section, we cannot compute the cost of an edge $(v_i, v_j)$ by executing $C(S[i : j-1])$ from scratch at each edge, since this would require time linear in the substring length, and thus $\Omega(n^3)$ time over all $S$’s substrings. Finally, we cannot materialize $G_\epsilon(S)$ (e.g. its adjacency lists) because it consists of $\Omega(n \text{polylog}(n))$ vertices, and thus its space occupancy would be super-linear in the original input size.

The rest of this section is devoted to design an algorithm which overcomes the three limitations above. Our algorithm deviates from the classic linear-time algorithm (Fact 1) in that materializes $G_\epsilon(S)$ on-the-fly, as its vertices are examined, by spending polylogarithmic time per edge. The actual time complexity per edge will depend on the entropy-based cost function we will use to estimate $|C(S[i : j-1])|$ (see Section 6.3) and on the dynamic data structure we will adopt to compute that estimation efficiently.

6.5.1 A General Scheme

The key tool we will use to make a fast estimation of the edge costs will be a dynamic data structure built over the input string $S$ and requiring $O(|S|)$ space. We will state the main properties of this data structure in an abstract form to design a general framework for solving our problem; in the next sections we will provide implementations for this data structure and thus obtain real time/space bounds for our problem. So, let us assume to have a dynamic data structure that maintains a set of sliding windows over $S$ denoted by $w_1, w_2, \ldots, w_{\log_{1+\epsilon} n}$. The sliding windows are substrings of $S$ which start at the same position $l$ but have different lengths: namely, $w_i = S[l : r_i]$ and $r_1 \leq r_2 \leq \ldots \leq r_{\log_{1+\epsilon} n}$. The data structure must support the following three operations:

1. Remove() moves the starting position $l$ of all sliding windows one position to the right (i.e. $l + 1$);
2. Append($w_i$) moves the ending position of the sliding window $w_i$ one position to the right (i.e. $r_i + 1$);
3. Size($w_i$) computes and returns the value $|C(S[l : r_i])|$. 
It is not difficult to notice that this data structure would be enough to generate \( \epsilon \)-maximal edges via a single pass over \( S \). More precisely, let \( l \) be the current position reached by our scan of \( S \), and hence let \( v_l \) be the currently examined vertex of \( G(S) \) in our shortest-path computation. We maintain the following invariant: sliding windows correspond to maximal edges, that is, the edge \((v_l, v_{l+r})\) is the \( \epsilon \)-maximal edge satisfying \( c(v_l, v_{l+r}) \leq (1 + \epsilon)^t < c(v_l, v_{l+r+1}) \). Hence, the set \( \{(v_l, v_{l+r_1}), (v_l, v_{l+r_2}), \ldots, (v_l, v_{l+r_{\log_2 n}})\} \) contains all the \( \epsilon \)-maximal edges going out from \( v_l \). Initially all indices are set to 0. For maintaining the invariant, when the scan advances to the next position \( l + 1 \), we call operation \text{Remove}() once to increment index \( l \) and, for each \( t = 1, \ldots, \log_2(n) \), we call operation \text{Append}(w_t) \) until we find the largest \( r_t \) such that \text{Size}(w_t) = c(v_l, v_{l+r_t}) \leq (1 + \epsilon)^t \). The key issue here is that \text{Append} and \text{Size} are paired so that our data structure should take advantage of the rightward sliding of \( r_t \) for computing \( c(v_l, v_{l+r_t}) \) efficiently. Just one character is entering \( w_t \) to its right, so we need to deploy this fact for making the computation of \text{Size}(w_t) \) fast (given its previous value). Here comes into play our second contribution that consists of adopting the entropy-bounded estimates of the compressibility of a string, mentioned in Section 6.3, to estimate indeed the edge costs \( C(w_t) \). This idea is crucial because we will be able to show that these functions do satisfy some structural properties that admit a fast incremental computation, as the one required by \text{Append} + \text{Size}. These issues will be discussed in the following sections, here we just state that, overall, the shortest-path computation over \( G_\epsilon(S) \) takes \( O(n) \) calls to operation 1, and \( O(n \log_{2+\epsilon} n) \) calls to operations 2 and 3.

**Theorem 26** If we have a dynamic data structure occupying \( O(n) \) space and supporting operation \text{Remove} in time \( L(n) \), and operations \text{Append} and \text{Size} in time \( R(n) \), then we can compute the shortest path in \( G_\epsilon(S) \) from \( v_1 \) to \( v_{n+1} \) taking \( O(n L(n) + R(n)(n \log_{1+\epsilon} n)) \) time and \( O(n) \) space.

### 6.5.2 0th Order Compressors

In this section we explain how to implement the data structure above whenever \( C \) is a 0th order compressor, and thus \( H_0 \) is used to provide a bound to the compression cost of \( G(S) \)'s edges (see Section 6.3). The key point is actually to show how to efficiently compute \text{Size}(w_i) \) as the sum of \( |S[l : r_i]| H_0(S[l : r_i]) = \sum_{c \in \Sigma} n_c \log((r_i - l + 1)/n_c) \) (see its definition in Section 1.2) plus \( f_0(r_i - l + 1, |S[l : r_i]|) \), where \( n_c \) is the number of occurrences of symbol \( c \) in \( S[l : r_i] \) and \( \sum_{c \in \Sigma} \) denotes the number of different symbols in \( S[l : r_i] \).

The first solution we are going to present is very simple and uses \( O(\sigma) \) space per window. The idea is the following: for each window \( w_i \) we keep in memory an array of counters \( A_i[c] \) indexed by symbol \( c \) in \( \Sigma \). At any step of our algorithm, the counter \( A_i[c] \) stores the number of occurrences of symbol \( c \) in \( S[l : r_i] \). For any window \( w_i \), we also use a variable \( E_i \) that stores the value of \( \sum_{c \in \Sigma} A_i[c] \log A_i[c] \). It is easy to notice that:
Therefore, if we know the value of $E_i$, we can answer to a query $\textbf{Size}(w_i)$ in constant time. So, we are left with showing how to implement efficiently the two operations that modify $l$ or any $r$ value and, thus, modify appropriately the $E$'s value. This can be done as follows:

1. **Remove()**: For each window $w_i$, we subtract from the appropriate counter and from variable $E_i$ the contribution of the symbol $S[l]$ which has been evicted from the window. That is, we decrease $A_i[S[l]]$ by one, and update $E_i$ by subtracting $(A_i[S[l]] + 1) \log(A_i[S[l]] + 1)$ and then summing $A_i[S[l]] \log A_i[S[l]]$. Finally we set $l = l + 1$.

2. **Append()**: We add to the appropriate counter and variable $E_i$ the contribution of the symbol $S[r_i + 1]$ which has been appended to window $w_i$. That is, we increase $A_i[S[r_i + 1]]$ by one, then we update $E_i$ by subtracting $(A[S[r_i + 1]] - 1) \log(A[S[r_i + 1]] - 1)$ and summing $A[S[r_i + 1]] \log A[S[r_i + 1]]$. Finally we set $r_i = r_i + 1$.

In this way, operation **Remove** requires constant time per window, hence $O(\log_{1+\epsilon} n)$ time overall. Operation **Append** takes constant time. The space required by the counters $A_i$ is $O(\sigma \log_{1+\epsilon} n)$ words. Unfortunately, the space complexity of this solution can be too much when it is used as the basic-block for computing the $k$th order entropy of $S$ as we will do in Section 6.5.3. In fact, we would achieve $\min(\sigma^{k+1} \log_{1+\epsilon} n, n \log_{1+\epsilon} n)$ space, which may be superlinear in $n$ depending on $\sigma$ and $k$.

The rest of this section is therefore devoted to provide an implementation of our dynamic data structure that takes the same query time above for these three operations, but within $O(n)$ space, which is independent of $\sigma$ and $k$. The new solution still uses $E$'s value but the counters $A_i$ are computed on-the-fly by exploiting the fact that all windows share the same value of $l$. We keep an array $B$ indexed by symbols whose entry $B[c]$ stores the number of occurrences of $c$ in $S[1 : l]$. We can keep these counters updated after a **Remove** by simply decreasing $B[S[l]]$ by one. We also precompute an array $R$ with an entry for each position in $S$. The entry $R[j]$ stores the number of occurrences of symbol $S[j]$ in $S[1 : j]$. The number of elements in both $B$ and $R$ is no more than $n$, hence they take $O(n)$ space.

These two arrays are enough to correctly update the value $E_i$ after **Append**, which is in turn enough to estimate $H_0$ (see Eqn 6.2). In fact, we can compute the value $A_i[S[r_i + 1]]$ by computing $R[r_i + 1] - B[S[r_i + 1]]$ which correctly reports the number of occurrences of $S[r_i + 1]$ in $S[l \ldots r_i + 1]$. Once we have the value of $A_i[S[r_i + 1]]$, we can update $E_i$ as explained in the above item 2.

We are left with showing how to support **Remove()** whose computation requires to evaluate the value of $A_i[S[l]]$ for each window $w_i$. Each of these values can be computed as $R[t] - B[S[l]]$ where $t$ is the last occurrence of symbol $S[l]$ in $S[l : r_i]$. The problem here is given by the fact that we do not know the position $t$. We solve
6.5. SPACE AND TIME EFFICIENT ALGORITHMS FOR GENERATING $G_c(S)$

this issue by resorting to a doubly linked list $L_c$ for each symbol $c$. The list $L_c$ links together the last occurrences of $c$ in all those windows, ordered by increasing position. Notice that a position $j$ may be the last occurrence of symbol $S[j]$ for different (but consecutive) windows. In this case we force that position to occur in $L_{S[j]}$ just once. These lists are sufficient to compute values $A_i[S[l]]$ for all the windows together. In fact, since any position in $L_{S[l]}$ is the last occurrence of at least one sliding window, in each of them can be used to compute $A_i[S[l]]$ for the appropriate indices $i$. Once we have all values $A_i[S[l]]$, we can update all $E_i$'s as explained in the above item 1. Since list $L_{S[l]}$ contains no more than $\log_{1+\epsilon} n$ elements, all $E_i$'s can be updated in $O(\log_{1+\epsilon} n)$ time. Notice that the number of elements in all the lists $L$ is bounded by the length of the string. Thus, they are stored using $O(n)$ space.

It remains to explain how to keep lists $L$ correctly updated. Notice that only one list may change after a Remove() or an Append$(w_i)$. In the former case we have possibly to remove position $l$ from list $L_{S[l]}$. This operation is simple because, if that position is in the list, then $S[l]$ is the last occurrence of that symbol in $w_i$ (recall that all the windows start at position $l$, and are kept ordered by increasing ending position) and, thus, it must be the head of $L_{S[l]}$. The case of Append$(w_i)$ is more involved. Since the ending position of $w_i$ is moved to the right, position $r_{i}+1$ becomes the last occurrence of symbol $S[r_{i}+1]$ in $w_i$. Recall that Append$(w_i)$ inserts symbol $S[r_{i}+1]$ in $w_i$. Thus, it must be inserted in $L_{S[r_{i}+1]}$ in its correct (sorted) position, if it is not present yet. Obviously, we can do that in $O(\log_{1+\epsilon} n)$ time by scanning the whole list. This is too much, so we show how to spend only constant time. Let $p$ be the rightmost occurrence of the symbol $S[r_{i}+1]$ in $S[0 : r_{i}]$ (it is easy to precompute and store the last occurrence of symbol $S[j+1]$ in $S[1 : j]$ for all $j$s in linear time and space). If $p < l$, then $r_{i}+1$ must be inserted in the front of $L_{S[r_{i}+1]}$ and we have done. In fact, $p < l$ implies that there is no occurrence of $S[r_{i}+1]$ in $S[l : r_{i}]$ and, thus, no position can precede $r_{i}+1$ in $L_{S[r_{i}+1]}$. Otherwise (i.e. $p \geq l$), we have that $p$ is in $L_{S[r_{i}+1]}$, because it is the last occurrence of symbol $S[r_{i}+1]$ for some window $w_j$ with $j \leq i$. We observe that if $w_j = w_i$, then $p$ must be replaced by $r_{i}+1$ which is now the last occurrence of $S[r_{i}+1]$ in $w_i$; otherwise $r_{i}+1$ must be inserted after $p$ in $L_{S[r_{i}+1]}$ because $p$ is still the last occurrence of this symbol in the window $w_j$. We can decide which one is the correct case by comparing $p$ and $r_{i-1}$ (i.e., the ending position of the preceding window $w_{r_{i-1}}$). In any case, the list is kept updated in constant time.

**Lemma 27** Let us assume that Size is estimated via the $0$th order entropy of $S$ (as detailed in Section 6.3). Then there exists a dynamic data structure that takes $O(n)$ space and supports Remove in $R(n) = O(\log_{1+\epsilon} n)$ time, and Append and Size in $L(n) = O(1)$ time.

By combining Lemma 27 with Theorem 26 we obtain:
Theorem 28  Given a string $S$ drawn from an alphabet of size $\sigma = \text{poly}(n)$, we can find an $(1 + \epsilon)$-optimal partition of $S$ with respect to a 0th order statistical compressor in $O(n \log_2(n))$ time and $O(n)$ space, where $\epsilon$ is any positive constant.

We point out that this result can be applied to the compression booster [31] described in Section 5.3 to fast obtain an approximation of the optimal partition of $\text{BWT}(S)$. This may be better than the algorithm of [31] both in time complexity, since that algorithm took $O(n\sigma)$ time, and in compression ratio (since the algorithm of [31] does not find the optimal partition). The case of a large alphabet (namely, $\sigma = \Omega(\text{poly}(\log n))$) is particularly interesting whenever we consider either a word-based $\text{BWT}$ [71] or the $\text{XBW}$-transform over labeled trees [31]. We notice that our result is interesting also for the Huffword compressor which is the standard choice for the storage of Web pages [92]; here $\Sigma$ consists of the distinct words constituting the Web-page collection.

6.5.3  $k$th Order Compressors

In this section we make one step further and consider the more powerful $k$th order statistical compressors, for which do exist $H_k$ bounds for estimating the size of their compressed output (see Section 6.3). Here $\text{Size}(w_i)$ must return $|C(S[l : r_i])|$ which is estimated by $(r_i - l + 1)H_k(S[l : r_i]) + f_k(|S[l : r_i]|, \sigma)$.

Let us denote with $S_q[1 : n - q]$ the string whose $i$-th symbol $S[i]$ is equal to the $q$-gram $S[i : i + q - 1]$. Actually, we can remap the symbols of $S_q$ to integers in $[1, n]$ without modifying the 0th order empirical entropy of its substrings. This is always possible because the number of distinct $q$-grams occurring in $S_q$ cannot be more than $n$, the length of $S$. Thus $S_q$’s symbols take $O(\log n)$ bits and $S_q$ can be stored in $O(n)$ space. There are various ways to compute this remapping. The simpler but, maybe not faster in practice, passes through the suffix array $SA$ of $S$. We assign as identifier to $S_k[i]$ the number of distinct $k$-grams that are lexicographically smaller than the corresponding $k$-gram $S[i : i + k - 1]$. This assignment can be easily done with a single pass on $SA$. Since the suffix array can be constructed in linear time for alphabet having polynomial size as a consequence of Theorem 3, $S_k$ is obtained in linear time and space.

It is well-known that the $k$th order empirical entropy of a string (see definition in Section 1.2) can be expressed as the difference between the 0th order empirical entropy of its $k + 1$-grams and its $k$-grams. This suggests that we can use the solution of the previous section in order to compute the 0th order empirical entropy of the appropriate substrings of $S_{k+1}$ and $S_k$. More precisely, we use two instances of the data structure of Theorem 28 (one for $S_{k+1}$ and one for $S_k$), which are kept synchronized in the sense that, when operations are performed on one data structure, then they are also executed on the other.
Lemma 29  Let us assume that Size is estimated via the $k$th order empirical entropy of $S$. There exists a dynamic data structure that takes $O(n)$ space and supports Remove in $R(n) = O(\log_{1+\epsilon} n)$ time, and Append and Size in $L(n) = O(1)$ time.

Theorem 30  Given a string $S$ drawn from an alphabet of size $\sigma = O(n^c)$, where $c$ is a positive constant, we can find an $(1 + \epsilon)$-optimal partition of $S$ with respect to a $k$th order statistical compressor in $O(n \log_{1+\epsilon} n)$ time and $O(n)$ space.

We point out that this result applies also to the practical case in which the base compressor $C$ has a maximum (block) size $B$ of data it can process at once. In this situation the time performance of our solution reduces to $O(n \log_{1+\epsilon}(B \log \sigma))$.

### 6.6 Extension to BWT-based Compressor

As we mentioned in Section 6.3 we know entropy-bounded estimates for the output size of BWT-based compressors. So we could apply Theorem 30 to compute the optimal partitioning of $S$ for such a type of compressors. Nevertheless, it is also known [29] that such compression-estimates are rough in practice because of the features of the compressors that are applied to the BWT($S$)-string. Typically, BWT is encoded via a sequence of simple compressors such as MTF encoding, RLE encoding (which is optional), and finally a 0th order encoder like Huffman or Arithmetic (Section 1.2.4). For each of these compression steps, a 0-th entropy bound is known [66], but the combination of these bounds may result much far from the final compressed size produced by the overall sequence of compressors in practice [29].

In this section, we propose a solution to the optimal partitioning problem for BWT-based compressors that introduces a $\Theta(\sigma \log n)$ slowdown in the time complexity of Theorem 30, but with the advantage of computing the $(1 + \epsilon)$-optimal solution wrt the real compressed size, thus without any estimation by any entropy-cost functions. Since in practice it is $\sigma = \text{poly}(\log n)$, this slowdown should be negligible. In order to achieve this result, we need to address a slightly different (but yet interesting in practice) problem which is defined as follows. The input string $S$ has the form $S[1]\#1 S[2]\#2 \ldots S[m]\#n$ where each $S[i]$ is a string (called page) drawn from an alphabet $\Sigma$, and $\#1, \#2, \ldots, \#n$ are special characters greater than any symbol of $\Sigma$. A partition of $S$ must be page-aligned, that is it must form groups of contiguous pages $S[i] \#i \ldots S[j] \#j$, denoted also $S[i : j]$. Our aim is to find a page-aligned partition whose cost (as defined in Section 6.1) is at most $(1 + \epsilon)$ the minimum possible cost, for any fixed $\epsilon > 0$. Moreover, to simplify things we will drop the RLE encoding step of a BWT-based algorithm.

We notice that this problem generalizes the table partitioning problem [11], since we can assume that $S[i]$ is a column of the table.

We start by observing that a close analog of Theorem 26 holds for this variant of the optimal partitioning problem, which implies that a $(1 + \epsilon)$-approximation of the optimum cost (and the corresponding partition) can be computed using a data
structure supporting operations Append, Remove, and Size; with the only difference that the windows \( w_1, w_2, \ldots, w_m \) subject to the operations are groups of contiguous pages of the form \( w_i = S[l : r_i] \).

It goes without saying that there exist data structures designed to dynamically maintain a dynamic text compressed with a BWT-based compressor under insertions and deletions of symbols (see [36] and references therein). But they do not fit our context for two reasons: (1) their underlying compressor is significantly different from the scheme above; (2) in the worst case, they would spend linear space per window yielding a super-linear overall space complexity.

Instead of keeping a given window \( w \) in compressed form, our approach will only store the frequency distribution of the integers in the string \( w' = \text{MTF}(\text{BWT}(w)) \) since this permits us to compute the 0th order entropy of \( w' \) and thus, by the above assumption, its overall compressed size. Since the MTF produces a sequence of integers from 0 to \( \sigma \), we can store their number of occurrences for each window \( w_i \) into an array \( F_{w_i} \) of size \( \sigma \). The update of \( F_{w_i} \) due to the insertion or the removal of a page in \( w_i \) incurs two main difficulties: (1) how to update \( w'_i \) as pages are added/removed from the extremes of the window \( w_i \), (2) perform this update implicitly over \( F_{w_i} \), because of the space reasons mentioned above. Our solution relies on two key facts about the BWT and the MTF transforms:

1. Since the pages are separated in \( S \) by distinct separators, inserting or removing one page into a window \( w \) does not alter the relative lexicographic order of the original suffixes of \( w \) (see [36]).

2. If a string \( x' \) is obtained from string \( x \) by inserting or removing a char \( c \) into an arbitrary position, then \( \text{MTF}(x') \) differs from \( \text{MTF}(x) \) in at most \( \sigma \) symbols. More precisely, if \( c' \) is the next occurrence in \( x \) of the newly inserted (or removed) symbol \( c \), then the MTF has to be updated only in the first occurrence of each symbol of \( \Sigma \) among \( c \) and \( c' \).

We describe a data structure supporting operations Append\((w)\) and Remove\()()\) when the base compressor is BWT-based, and the input string \( S \) is the concatenation of a sequence of pages \( S[1], S[2], \ldots, S[n] \) separated by unique separator symbols \#1, \#2, \ldots, \#m, \) which are lexicographically larger than any other symbol of the alphabet \( \Sigma \). We denote with \( \text{SA}[1, n] \) and \( \text{ISA}[1, n] \) respectively the suffix array of \( S \) and its inverse. Given a range \( I = [a, b] \) of positions of \( S \), an occurrence of a symbol of \( \text{BWT}(S) \) is called active\(\{a,b\}\) if it corresponds to a symbol in \( S[a : b] \). For any range \( [a, b] \subset [n] \) of positions in \( S \), we define \( \text{RBWT}(S[a : b]) \) as the string obtained by concatenating the active\(\{a,b\}\) symbols of \( \text{BWT}(S) \) preserving their relative order. In the following, we will not indicate the interval when it will be clear from the context. Notice that, due to the presence of separators, \( \text{RBWT}(S[a : b]) \) coincides with \( \text{BWT}(S[a : b]) \) when \( S[a : b] \) spans a group of contiguous pages (see [36] and references therein). Moreover, \( \text{MTF}(\text{RBWT}(S[a : b])) \) is the string obtained by performing the MTF algorithm on \( \text{RBWT}(S[a : b]) \). We will call the symbol \( \text{MTF}(\text{RBWT}(S[a : b]))[i] \) the MTF encoding of symbol \( \text{RBWT}(S[a : b])[i] \).
For each window \( w \), our solution will not explicitly store neither \( \text{RBWT}(w) \) or \( \text{MTF}(\text{RBWT}(S[a:b])) \) since this might require a superlinear amount of space. Instead, we maintain only an array \( F_w \) of size \( \sigma \) whose entry \( F_w[e] \) keeps the number of occurrences of encoding \( e \) in \( \text{MTF}(\text{RBWT}(w)) \). The array \( F_w \) enouh to compute the 0th order entropy of \( \text{MTF}(\text{RBWT}(w)) \) in \( \sigma \) time (or eventually the exact cost of compressing it with Huffman in \( \sigma \log \sigma \) time).

We are left with showing how to correctly keep updated \( F_w \) after a \( \text{Remove}(w) \) or an \( \text{Append}(w) \). In the following we will concentrate only on \( \text{Append}(w) \) since \( \text{Remove}(w) \) is symmetrical. The idea to perform \( \text{Append}(w) \), where \( w = S[l:r] \), is to conceptually insert the symbols of the next page \( S[r+1] \) into \( \text{RBWT}(w) \) one at time from left to right. Since the relative order of symbols of \( \text{RBWT}(w) \) is preserved in \( \text{BWT}(S) \), it is more convenient to work with active symbols of \( \text{BWT}(S) \) by resorting to a data structure, whose details are given later, which is able to efficiently answer the following two queries with parameters \( c \) and \( h \), where \( c \in \Sigma \), \( I = [a,b] \) is a range of positions in \( S \) and \( h \) is a position in \( \text{BWT}(S) \):

- \( \text{Prev}_c(I, h) \): locate the last active\([a,b]\) occurrence in \( \text{BWT}(S)[0:h-1] \) of \( c \);
- \( \text{Next}_c(I, h) \): locate the first active\([a,b]\) occurrence in \( \text{BWT}(S)[h+1:n] \) of \( c \).

This data structure is built over the whole string \( S \) and requires \( O(|S|) \) space.

Let \( c \) be the symbol of \( S[r_i+1] \) we have to conceptually insert in \( \text{RBWT}(S[a:b]) \). We can compute the position (say, \( h \)) of this symbol in \( \text{BWT}(S) \) by resorting to the inverse suffix array of \( S \). Once we know position \( h \), we have to determine what changes in \( \text{MTF}(\text{RBWT}(w)) \) the insertion of \( c \) has produced and update \( F_w \) accordingly. It is not hard to convince ourselves that the insertion of symbol \( c \) changes no more than \( \sigma \) encodings in \( \text{MTF}(\text{RBWT}(w)) \). In fact, only the first active occurrence of each symbol in \( \Sigma \) after position \( h \) may change its \( \text{MTF} \) encoding. More precisely, let \( h_p \) and \( h_n \) be respectively the last active occurrence of \( c \) before \( h \) and the first active occurrence of \( c \) after \( h \) in \( \text{BWT}(w) \), then the first active occurrence of a symbol after \( h \) changes its \( \text{MTF} \) encoding if and only if it occurs active both in \( \text{BWT}(w)[h_p:h] \) and in \( \text{BWT}(w)[h:h_n] \). Otherwise, the new occurrence of \( c \) has no effect on its \( \text{MTF} \) encoding. Notice that \( h_p \) and \( h_n \) can be computed with queries \( \text{Prev}_c \) and \( \text{Next}_c \). In order to correctly update \( F_w \), we need to recover for each of the above symbols their old and new encodings. The first step consists on finding the last active occurrence before \( h \) of each symbols in \( \Sigma \) using \( \text{Prev} \) queries. Once we have these positions, we can recover the status of the \( \text{MTF} \) list, denoted \( \lambda \), before encoding \( c \) at position \( h \). This is simply obtained by sorting the symbols ordered by decreasing position. In the second step, for each distinct symbol that occurs active in \( \text{BWT}(w)[h_p:h] \), we find its first active occurrence in \( \text{BWT}(w)[h:h_n] \). Knowing \( \lambda \) and these occurrences sorted by increasing position, we can simulate the \( \text{MTF} \) algorithm to find the old and new encodings of each of those symbols.

This provides an algorithm to perform \( \text{Append}(w) \) by making \( O(\sigma) \) queries of types \( \text{Prev} \) and \( \text{Next} \) for each symbol of the page to append in \( w \). To complete the
proof of the time bounds in Theorem 31 we have to show how to support queries of type \texttt{Prev} and \texttt{Next} in $O(\log n)$ time and $O(n)$ space. This is achieved by a straightforward reduction to a classic geometric range-searching problem. Given a set of points $P = \{(x_1, y_1), (x_2, y_2), \ldots, (x_p, y_p)\}$ from the set $[n] \times [n]$ (notice that $n$ can be larger than $p$), such that no pair of points share the same $x$- or $y$-value, there exists a data structure \cite{63} requiring $O(p)$ space and supporting the following two queries in $O(\log p)$ time:

- **rangemax**$([l, r], h)$: return among the points of $P$ contained in $[l, r] \times [-\infty, h]$ the one with maximum $y$-value
- **rangemin**$([l, r], h)$: return among the points of $P$ contained in $[l, r] \times [h, +\infty]$ the one with minimum $y$-value

Initially we compute $ISA$ and $SA$ in $O(n \log \sigma)$ time then, for each symbol $c \in \Sigma$, we define $P_c$ as the set of points $\{(i, ISA[i+1])| S[i] = c\}$ and build the above geometric range-searching structure on $P_c$. It is easy to see that \texttt{Prev}_c(I, h) can be computed in $O(\log n)$ time by calling **rangemax**$(I, h)$ on the set $P_c$, and the same holds for \texttt{Next}_c by using **rangemin** instead of **rangemax**, this completes the reduction and the proof of the following theorem:

**Theorem 31** If $C$ is a BWT-based compressor, we can compute an $(1+\epsilon)$-approximate solution to the optimal partitioning problem for a sequence of pages, of total length $n$ and alphabet size $\sigma$, in $O(n(\log_{1+\epsilon} n)\sigma \log n)$ time and $O(n + \sigma \log_{1+\epsilon} n)$ space.
Future Works

We have successfully addressed the question of devising parsing algorithms that achieve optimality in the output-size of the underlying compressor when used to compress a string with the Lempel-Ziv compression scheme, or to find an optimal partition for the PPC compression paradigm. To finish, let us mention some of the main research directions into which our work can be extended.

In Part II we have addressed the question of bit-optimality in LZ77-parsing. One of the main questions left open is whether the result of Theorem 11 can be extended to the case where codewords are encoded by statistical encoding functions like Huffman or Arithmetic encoders. They do not necessarily satisfy Property 1 because it might be the case that $x < y$ but $|f(x)| > |f(y)|$, because the integer $y$ occurs more frequently than the integer $x$ in the parsing of $S$. We argue that it is not trivial to design a bit-optimal compressor for these encoding functions because their codeword lengths change as it changes the set of distances and lengths used in the parsing process. Moreover, our experimental studies of bit-optimal LZ77-parsers have shown interesting results but we believe that an algorithmic-engineering effort is still required to tune the performance of the proposed bit-optimal algorithms, by possibly designing integer encoders which are suited for the data collections in input.

In Part III we have explored the optimal partitioning problem for the PPC-compression paradigm. We have provided the first algorithm which is guaranteed to compute in $O(n \log_{1+\epsilon} n)$ time a partition of the input string whose compressed output is guaranteed to be no more than $(1 + \epsilon)$-worse the optimal one, where $\epsilon$ may be any positive constant. In our opinion, there exists at least two interesting ways of extending this work: one is investigating algorithms for computing the exact optimal partition in $o(n^2)$ time, and the other is performing an experimental study of our algorithms in the compression of large datasets.
Bibliography


