UNIVERSIDADE DA CORUÑA

Departamento de Computación

Algorithms and Compressed Data Structures for Information Retrieval

Tesis Doctoral

Doctoranda: Susana Ladra González

Directores: Nieves Rodríguez Brisaboa, Gonzalo Navarro Badino

A Coruña, Abril de 2011

PhD thesis supervised by
Tesis doctoral dirigida por

Nieves Rodríguez Brisaboa
Departamento de Computación
Facultade de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 981 167000 ext. 1243
Fax: +34 981 167160
brisaboa@udc.es

Gonzalo Navarro Badino
Departamento de Ciencias de la Computación
Universidad de Chile
Blanco Encalada 2120 Santiago (Chile)
Tel: +56 2 6892736
Fax: +56 2 6895531
gnavarro@dcc.uchile.cl

A mis padres y hermanas

Acknowledgments

I have left this page of the thesis until the last moment, being, undoubtedly, one of the most difficult pages to write. The rest haven't been easy (Nieves knows), but in this one I test my expertise in the area I work: I must compress into just a few lines my gratitude to all who have helped, encouraged and taught me throughout this thesis. I hope to be lossless in the attempt.

First I would like to make a special mention to my thesis advisors, Nieves and Gonzalo, for all the knowledge you have transmitted to me and the time you devoted to my thesis during the whole process, from the birth of each idea to the very thorough review of the final document. But above all I thank and appreciate the luck of having two advisors so close and accessible.

I also want to thank Sebastiano Vigna and Jorma Tarhio for their reviews of the thesis, improving it with their observations, and all the members of my thesis committee: Isidro Ramos, Ricardo Baeza-Yates, Josep Díaz, Alejandro López-Ortiz and Paolo Ferragina.

Thanks to all the members of the Database Laboratory. Especially to my third advisor behind the scenes, Antonio Fariña, whose help and encouragement have been constant since I started doing research. I've been surrounded by great people, who have been excellent travelling companions, sharing professional and personal experiences, and have become good friends.

During my two research experiences abroad I met a lot of people. Among them I would like to acknowledge for their hospitality and help in research topics to Francisco Claude, Diego Arroyuello, Rodrigo Paredes, Rodrigo Cánovas, Miguel A. Martínez, Veli Mäkinen, Niko Välimäki and Leena Salmela.

I cannot forget all those friends with whom I spent many memorable moments outside research. To my basketball referees colleagues, with whom I have enjoyed every game or evening out. And many friends, with whom I spent less time than I would have liked.

And my biggest gratitude to my family. My parents and sisters. And all the others, for your constant encouragement. Because you have given me everything, and I know you will always do. This thesis is as mine as yours.

Agradecimientos

He dejado para el final esta página de la tesis, sin duda, una de las más difíciles de escribir. Si bien es cierto que el resto han costado lo suyo (Nieves lo sabe), en esta página pongo a prueba mi pericia en el área en la que trabajo: tengo que comprimir en pocas líneas mi agradecimiento a todos los que me han ayudado, animado y enseñado a lo largo de esta tesis. Espero no perder información en el intento.

En primer lugar me gustaría hacer una mención especial a mis directores de tesis, Nieves y Gonzalo, por todos los conocimientos que me habéis transmitido y el tiempo que habéis dedicado a mi tesis en todo su proceso, desde el nacimiento de cada idea hasta la revisión tan minuciosa del texto final. Pero sobre todo agradezco y valoro la suerte de poder contar con unos directores de tesis tan cercanos y accesibles.

Además me gustaría agradecer a Sebastiano Vigna y Jorma Tarhio por sus revisiones de la tesis y sus comentarios, que me han permitido mejorarla, y a todos los miembros del tribunal: Isidro Ramos, Ricardo Baeza-Yates, Josep Díaz, Alejandro López-Ortiz y Paolo Ferragina.

Gracias a todos los miembros del Lab. de Bases de Datos. A mi tercer director de tesis en la sombra, Antonio Fariña, cuya ayuda y ánimo han sido constantes desde mis comienzos en la investigación. Además he estado rodeada de compañeros fantásticos, tanto de viajes como de experiencias laborales y personales, ejerciendo hasta de consejeros en viajes de metro, y se han convertido en amigos.

Durante las dos estancias que he tenido la fortuna de disfrutar he conocido a mucha gente entre los que destaco tanto por su hospitalidad como por la ayuda en temas de investigación a Francisco Claude, Diego Arroyuelo, Rodrigo Paredes, Rodrigo Cánovas, Miguel A. Martínez, Veli Mäkinen, Niko Välimäki y Leena Salmela.

Tampoco puedo olvidar a todos los compañeros y amigos con los que he pasado muchos momentos inolvidables al margen de la investigación. A mis compañeros árbitros, con los que he disfrutado en cada partido o quedada. O a muchos amigos, con los que he pasado menos tiempo del que me hubiese gustado.

Y mi mayor agradecimiento a mi familia. A mis padres y hermanas. Y a los demás, por vuestros ánimos constantes. Porque me lo habéis dado todo, y sé que seguiréis haciéndolo siempre. Esta tesis es tan mía como vuestra.

Abstract

In this thesis we address the problem of the efficiency in Information Retrieval by presenting new compressed data structures and algorithms that can be used in different application domains and achieve interesting space/time properties.

We propose (i) a new variable-length encoding scheme for sequences of integers that enables fast direct access to the encoded sequence and outperforms other solutions used in practice, such as sampling methods that introduce an undesirable space and time penalty to the encoding; (ii) a new self-indexed representation of the compressed text obtained by any word-based, byte-oriented compression technique that allows for fast searches of words and phrases over the compressed text occupying the same space than the space achieved by the compressors of such type, and obtains better performance than classical inverted indexes when little space is used; and (iii) a new compact representation of Web graphs that supports efficient forward and reverse navigation over the graph using the smallest space reported in the literature, and in addition it also allows for extended functionality not usually considered in compressed graph representations.

These data structures and algorithms can be used in several scenarios, and we experimentally show that they can successfully compete with other techniques commonly used in those domains.

Resumen

En esta tesis abordamos el problema de la eficiencia en la Recuperación de Información presentando nuevas estructuras de datos compactas y algoritmos que pueden ser usados en diferentes dominios de aplicación y obtienen interesantes propiedades en espacio y tiempo.

En ella proponemos (i) un nuevo esquema de codificación de longitud variable para secuencias de enteros que permite un rápido acceso directo a la secuencia codificada y supera a otras soluciones utilizadas en la práctica, como los métodos de muestreo que introducen una penalización indeseable en tiempo y espacio; (ii) una nueva representación autoindexada del texto comprimido obtenido por cualquier técnica de compresión orientada a byte y palabra que permite búsquedas eficientes de palabras y frases sobre el texto comprimido usando el mismo espacio que el obtenido por técnicas de compresión de dicho tipo, y que obtiene mejores resultados que índices invertidos clásicos cuando se usa poco espacio; y (iii) una nueva representación compacta de grafos Web que soporta una navegación directa y reversa eficiente usando el menor espacio de la literatura, y además permite una funcionalidad extendida no considerada usualmente por otras representaciones comprimidas de grafos.

Estas estructuras de datos y algoritmos pueden utilizarse en diferentes escenarios, y probamos experimentalmente que compiten exitosamente con otras técnicas comúnmente usadas en esos dominios.

Resumo

Nesta tese abordamos o problema da eficiencia na Recuperación de Información presentando novas estruturas de datos compactas e algoritmos que poden ser usados en diferentes dominios de aplicación e obteñen interesantes propiedades en espazo e tempo.

Nela propoñemos (i) un novo esquema de codificación de lonxitude variable para secuencias de enteiros que permite un rápido acceso directo á secuencia codificada e supera a outras solucións utilizadas na práctica, como os métodos de mostraxe que introducen unha penalización indesexable en tempo e espazo; (ii) unha nova representación autoindexada do texto comprimido obtido por calquera técnica de compresión orientada a byte e palabra que permite buscas eficientes de palabras e frases sobre o texto comprimido usando o mesmo espazo que as técnicas de compresión de dito tipo, e que obtén mellores resultados que índices invertidos clásicos cando se usa pouco espazo; e (iii) unha nova representación compacta de grafos Web que soporta unha navegación directa e reversa eficiente usando o menor espazo da literatura, e que ademais permite unha funcionalidade estendida non considerada usualmente por outras representacións comprimidas de grafos.

Estas estruturas de datos e algoritmos poden utilizarse en diferentes escenarios, e probamos experimentalmente que compiten exitosamente con outras técnicas comunmente usadas neses dominios.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Information Retrieval (IR) is a very active research area focused on studying procedures to help users to locate data of their interest. Since new information needs are constantly arising, easy and fast access to the data is highly demanded. Information retrieval systems are everywhere: in addition to those well-known Web search engines, library and store catalogs, cookbook indexes, and so on are used in our everyday life. These IR systems are manual or computerized processes for storing, organizing and accessing information, such that the *relevant information*, that is, the information that the user needs, can be efficiently provided to the user when it is demanded.

Moreover, in the last years the information on the Web has also increased explosively, introducing new problems on the field. Finding useful information among the billion Web pages that it contains becomes a tedious and challenging task. In order to satisfy their information needs, users might navigate the Web links searching for information of interest. The analysis of those links is also a very interesting research area. For instance, it can help IR systems to find good sources of content for a given query, since a good source of content is generally linked by many pages which are also related with that query.

Information retrieval relies on complex systems that facilitate the access to large volumes of data in order to satisfy the user's information needs. Behind a user friendly interface where the user can write a query, IR systems hide an intricate architecture that includes multiple algorithms and data structures. The efficiency of the whole system depends significantly on this low-level layer, where the information must be represented and indexed so that the relevant information can be found and displayed to the user. In order to make efficient that search, most of the IR systems use some indexing techniques that reduce the sequential scan over the data when searching for the desired information. In addition, indexes and data are also represented in a compact way so that efficient retrieval can be achieved. In this way, data compression and indexing work together in order to improve the efficiency of IR systems.

In this context, our aim is the study of compressed data structures and algorithms to represent data in little space while allowing efficient access to it. Compression techniques do not only aim at reducing the size of the data structure, but they can also add some extra benefits: by reducing its space requirements, the compressed data structure might fit in main memory rather than swapping out to disk, operating in higher and faster levels of the memory hierarchy. Then, a compressed data structure is interesting when it maintains (or improves if possible) all the capabilities and properties of the plain representation of the same data and allows to perform fast queries to the data directly over the compressed form, without decompressing it before its use. In this case, the compressed data structure becomes a more efficient alternative against operating over the plain representation of the data if it does not fit in main memory and it is stored in secondary memory.

In this thesis we address the problem of the efficiency in Information Retrieval by presenting some new general low-level data structures and algorithms that can be adapted to different domains and achieve interesting space/time properties compared to other techniques of the state-of-the-art in those domains.

More concretely, in this thesis we present three proposals that deal with three different problems of Information Retrieval. In order to understand the particularities of each domain, we explain the motivation and context for each problem in the following sections.

Variable-Length Codes

Variable-length coding is present in several techniques used in information retrieval, the best known examples are the compression of the lists of the inverted indexes [WMB99] or some other compact structures such as compressed suffix arrays for text retrieval [Sad03]. Variable-length codes can achieve better compression ratio than using a fixed-length encoding. One of the best known variable-length encoding technique was introduced by Huffman [Huf52].

However, apart from the compression ratio obtained, other important aspect consists in the possibility to access directly to the symbol encoded at any position of the encoded sequence, without the need to decompress all the sequence first. However, it is not possible to access directly to a symbol at a certain position if the sequence is compressed with variable-length codes, since the start of the codeword assigned to that symbol depends on the lengths of all the codewords previous to that position of the sequence.

The classical solution to permit direct access to random positions consists in regularly sampling some symbols of the original sequence and storing the starting positions of their codewords within the encoded sequence. The more samples are stored, the faster the direct access to a single position of the encoded sequence is, but this can lead to an undesirable worsening of the compression ratio. Different variants of this classical solution are used to provide random access to compressed inverted indexes [CM07, ST07].

Hence, it would be interesting to obtain a variable-length encoding scheme that represents sequences of integers in a compact way and supports fast direct access to any position without the need of any extra sampling.

Text Retrieval

Word-based text searching is a classical problem in Information Retrieval. Given a natural language text T composed by a sequence of words from a vocabulary Σ, searching a pattern P, also composed by a sequence of words from Σ, consists in finding all the occurrences of P in T.

There are two general approaches for solving this search problem: sequential and indexed text searching. The sequential searching approach consists in scanning the complete plain representation of T from the beginning to the end, searching for pattern P. This naive technique is only used in practice when the text is small, so it is affordable. If the length of the text T is n and the length of the pattern is m, the number of comparisons among the words of text and the words of the pattern is $O(mn)$. Then, if the size of the text increases, this approach becomes highly inefficient. There are some compression techniques that permit searching for words directly on the compressed text so that the search can be up to eight times faster than searching the plain uncompressed text [MNZBY00]. This speed-up is due to the fact that there are less data to process, since the text is in compressed form. In addition, search times can also be improved with the use of byte-oriented encoding schemes, which permit faster comparisons than bit-oriented encodings. Moreover, some of these compression methods allow the use of efficient pattern-matching algorithms, such as Boyer-Moore [BM77] or Horspool [Hor80], which reduce the portion of text scanned during the search, skipping some bytes of the compressed text. However, the complexity of the search continues being proportional to the size of the text, even if some improvement is obtained.

Hence, it becomes necessary to construct some kind of data structure over the text, an *index*, to reduce that number of comparisons between the whole text and the pattern, so that the search becomes independent of the size of the text. With the indexed text, searching is improved at the expense of increasing the space requirement, due to the index structure. This approach is of great interest in several scenarios, for instance, when the text is so large that a sequential scan is prohibitively costly or many searches (using different patterns) must be performed on the same text.

Compression methods and indexes improve searches separately, but they can also be combined to achieve interesting effects. Classical indexes [BYRN99, WMB99] require a considerable extra space in addition to the text representation, so some compression techniques can be used in order to minimize that extra space. In addition to the compression of the index, the text can also be compressed. If the text is compressed with a technique that allows direct searching for words in the compressed text, then the compressed text supports efficient pattern-matching algorithms, and therefore the scanning of the text, when needed, becomes quicker than over plain text.

Current indexes aim to exploit the text compressibility. This concept has evolved in a more complex concept called *self-indexes*. They are space-efficient indexes that contain enough information to reproduce the whole text, since they are designed to support efficient searches without the need to store the original text in a separate structure. Thus, a self-index is, in itself, an index for the text and its representation, requiring very little memory, close to the compressed text size. In fact, they can be regarded as a compression mechanism that offers added value, as efficient searching capabilities, to the pure reduced space demand. This field is still open to improvements, especially for word-based self-indexes, where very few structures have been proposed.

Web Graph Compression

We also address the problem of information retrieval on the Web, and more particularly, the study of the Web as a graph. The graph representation of the Web, which consists of all the Web pages (nodes) with the hyperlinks between them (directed edges), is commonly used as the basis for multiple algorithms for crawling, searching and community discovery. The Web graph represents all the hyperlinks of the Web, so it can be used to extract relevant information from the study of those links between Web pages, which is called link analysis. For instance, Web graphs can be used to crawl the Web, starting with an initial set of Web pages and following the outgoing links of the new discovered Web pages. It is also used to study the Web structure, such that it is possible, for example, to know if there are local substructures, how many hops there are from one Web page to another or to identify Web communities [RKT99]. It has been proved that the study of the link structure, represented in Web graphs, can be useful to improve the information retrieval on the Web. For example, the *PageRank* algorithm [PBMW99] ranks Web pages according to the number and importance of the pages that link to them; for instance, it is used by *Google* search engine to decide which Web pages are more relevant to a user query. Web graphs are also used to classify pages, to find related pages, or for spam detection, among other tasks. There are entire conferences devoted to graph algorithms for the Web (e.g. *WAW: Workshop on Algorithms and*

Models for the Web-Graph).

The main problem of solving these IR problems using the Web graph representation is the size of the graphs. As they represent the whole Web, which contains billions of Web pages and hundreds of billions of links, Web graphs are large and their plain representation cannot be completely stored in the current main memories. In this scenario, several compressed representations of Web graphs have been proposed [BBH$^+$98, BKM$^+$00, AM01, SY01, RSWW01, RGM03, BV04], most of them allowing some basic navigation over the compressed form so that it is not necessary to decompress the entire Web graph to obtain, for instance, the list of Web pages pointed by a particular Web page. Therefore, Web graphs (or a big part of them) can be stored and manipulated in main memory, obtaining better processing times than the plain form, which must be stored in secondary memory. New strategies can be studied in order to achieve better spaces or times and increase the navigation possibilities over the compressed representation of the Web graph.

1.2 Contributions

In this thesis we propose several new techniques that can be applied to different information retrieval systems. We present some compact data structures and algorithms that operate efficiently in very little space and solve the problems that have been briefly presented in the previous section. We now enumerate each contribution detailing the problem it addresses.

Variable-Length Codes: Directly Addressable Codes

The first of our contributions consists in the design, analysis, implementation and experimental evaluation of a new variable-length encoding scheme for sequences of integers that enables direct access to any element of the encoded sequence. We call our proposal *Directly Addressable Codes (DACs)*. Basically, it divides the bits of the binary representation of each integer of the sequence into several chunks of bits and rearranges these chunks into different levels. The chunks with the least significant bits of each integer are placed in the first level, then the next chunks with the second least significant bits are placed in the second level and so on until the last level, which contains the chunks with the most significant bits. Each integer is encoded with a variable number of chunks: a smaller integer is encoded with fewer bits than a larger integer, so a very compact space can be achieved. Moreover, it is possible to directly access to the first chunk of the code for each integer and continue obtaining the rest of the code in a quick way.

Our proposal is a kind of *implicit* data structure that introduces synchronism in the encoded sequence without using asymptotically any extra space. We show some experiments demonstrating that the technique is not only simple, but also competitive in time and space with existing solutions in several applications, such as the representation of LCP arrays or high-order entropy-compressed sequences.

In addition, we propose an optimization algorithm to obtain the most compact space given the frequency distribution of the sequence of integers that we want to represent, and we explain how our strategy can be generalized and used to provide direct access over any sequence of symbols encoded with a variable-length code by just a rearrangement of the codeword chunks.

The conceptual description of the technique and some of the application results were published in the proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE 2009) [BLN09a].

Text Retrieval: Byte-Oriented Codes Wavelet Tree

Our second contribution is the design, analysis, implementation and experimental evaluation of a new data structure that permits the compact representation and efficient manipulation of natural language texts. Our proposal, called *Byte-Oriented Codes Wavelet Tree (BOC-WT)*, is a tree-shaped structure that maintains the properties of the compressed text obtained by any word-based, byte-oriented prefix-free encoding technique (same compression ratio and comparable compression and decompression times) and drastically improves searches, since some implicit self-indexing capabilities are achieved. This method, inspired by the Direct Addressable Codes, rearranges the bytes of the compressed text obtained by the byte-oriented encoding scheme following a wavelet tree shape [GGV03]. Besides placing the bytes of the codewords in several levels, so direct access is obtained as with the Direct Addressable Codes, the bytes of each level are separated into different branches building a tree. Then, each word is associated with one leaf of this tree and can be searched independently of the length of the text.

BOC-WT obtains efficient time results for counting, locating and extracting snippets when searching for a pattern in a text without worsening the performance as a compression method. In fact, this proposal can be compared to classical inverted indexes and it obtains interesting results when the space usage is not high.

This new data structure was presented in preliminary form at the 31st International Conference on Research and Development in Information Retrieval (SIGIR 2008) [BFLN08].

Web Graph Compression: k^2-tree

The third contribution of this thesis consists in the design, analysis, implementation and experimental evaluation of a new compact representation for Web graphs, called k^2*-tree*. Conceptually, it is a tree that represents the recursive subdivision of the adjacency matrix of the Web graph, but this tree representation can be stored as a couple of bitmaps in a very compact space. This compact data structure supports basic navigation over the Web graph, that is, retrieving the direct and reverse list of neighbors of a page, in addition to some interesting extra functionality. We compare our results with the current methods of the state of the art in the field, and show that our method is competitive with the best alternatives in the literature, offering an interesting space/time tradeoff. Moreover, we show how our first contribution, the Directly Addressable Codes, can be applied in this context as well, improving simultaneously both time and space results. When using DACs, we achieve the smallest representation for Web graphs compared to the methods of the literature that also support direct and reverse navigation over the representation of the graph.

This approach can be generalized to any kind of binary relation so it can be applied in different scenarios in Information Retrieval. For instance, we could consider the relation between documents and terms (keywords) in those documents, so that we could represent the index of a text collection with our proposal.

The main ideas of this work were published in the proceedings of the 16th International Symposium on String Processing and Information Retrieval (SPIRE 2009) [BLN09b].

We note that all the techniques proposed in this thesis are conceived to operate in main memory, mainly due to the random access pattern presented in all of them. This fact can be, at first, regarded as a serious restriction since we want to index huge volumes of data. However, recent hardware developments, such as the availability of 64−bit architectures and the increase of the usage of cluster environments, have led to a scenario where large collections can be entirely addressed on main memory. In this way, there has been a lot of recent research in efficient document retrieval in main memory [SC07, CM07] where indexes are stored completely in main memory. Therefore, the cost of random reads, which is one of the greatest bottleneck of traditional information systems, is minimized. Hence, it is important to focus on another key aspect of the efficiency of information retrieval systems: the huge volume of data. Then, the goal is to process less amount of data, that is, to read fewer bytes. In this scenario, compact data structures achieve great importance and they were the main objective of this thesis.

1.3 Structure of the thesis

First, in Chapter 2, some basic concepts about Information Retrieval and succinct data structures are presented. After that, the remainder of this thesis is organized in three parts corresponding to each contribution and a concluding part for the thesis. Each part is organized in chapters as follows.

Part one is focused on the study of the direct access problem when variable-length codes are used. Given a sequence of integers that we want to represent in a little space, we propose a new variable-length encoding scheme, *Directly Addressable Codes*, that supports direct access to any position of the sequence (achieving constant time per symbol of the target alphabet) in an easy and fast way. In Chapter 3 we present the motivation of the problem, some notation and basic concepts of variable-length codes. We finish the chapter by enumerating some existing solutions from previous work.

Chapter 4 presents our proposal to represent variable-length codes allowing direct access to any position, including an optimization algorithm to improve the space usage of our technique.

Our Directly Addressable Codes are applied over different scenarios in Chapter 5, where the experimental results are shown and discussed. Finally, Chapter 6 summarizes the main contributions and other applications for this proposal.

Part two presents a new data structure, *Byte-Oriented Codes Wavelet Tree (BOC-WT)*, that represents a natural language text in a compressed form and supports efficient searches, close to other indexing structures. Chapter 7 revises the basic properties of some word-based byte-oriented variable-length compression methods, emphasizing their self-synchronization condition.

Chapter 8 explains the Byte-Oriented Codes Wavelet Tree (BOC-WT) in detail and presents the algorithms to compress, decompress and search words and phrases using this new data structure.

Chapter 9 presents the empirical evaluation, comparing the performance of the BOC-WT structures and algorithms with the compression methods of the state of the art. In addition, we analyze its indexing properties by comparing the time results for searches with other indexing structures using the same amount of space. Finally, Chapter 10 discusses the conclusions and other applications.

In ***Part three*** we address the problem of the Web graph compression. We present a compact representation of Web graphs called *k^2-tree*. In Chapter 11 we study the Web graph compression problem and its current state of the art, detailing the main properties that have been identified and exploited to achieve compression.

In Chapter 12 we propose the compact representation of a Web graph called k^2-tree. We describe the conceptual idea of the representation and detail practical implementation aspects. All the algorithms of construction and navigation over the compact representation of the Web graph are included. We also present several variants of the method proposed.

Chapter 13 presents the empirical evaluation of our technique, comparing the performance of the structure and algorithms with the methods of the literature in Web graph compression. Finally, Chapter 14 presents the main conclusions, other applications and future work, including the possibility of the usage of the k^2-tree technique as a more general purpose method for indexing binary relations.

To complete the thesis, we include a concluding chapter in ***Part four***. Chapter 15 summarizes the contribution of the thesis and enumerates some future directions of the research. Finally, Appendix A lists the publications and other research results derived from this thesis, and the works published by other researchers that take our proposals into consideration.

Chapter 2

Previous concepts

Information retrieval systems deal with large collections of information. During the last years the amount of such data has dramatically increased, requiring techniques such as compression and indexing of the data in order to efficiently handle the available information. In this thesis we propose new algorithms and compressed data structures for Information Retrieval, hence we need to introduce some notions of Information Theory that will help us to understand the basis of Data Compression.

This chapter presents the basic concepts that are needed for a better understanding of the thesis. A brief description of several concepts related to Information Theory are first shown in Section 2.1. Section 2.2 introduces the basis of data compression, including the description of one of the most well-known and used compression techniques, the classic Huffman algorithm, in Section 2.2.1. Finally, Section 2.3 describes some succinct data structures to solve basic operations over sequences, which are commonly used to improve the efficiency of other high-level structures.

2.1 Concepts of Information Theory

Information theory deals with the measurement and transmission of information through communication channels. Shannon's work [SW49] settled the basis for the field. It provides many useful concepts based on measuring information in terms of bits or, more generally, in terms of the minimum amount of the complexity of structures needed to encode a given piece of information.

Given a discrete random variable X with a probability mass function p_X and domain $\mathcal{X}$, the amount of *information* or "surprise" associated with an outcome $x \in \mathcal{X}$ is defined by the quantity $I_X(x) = \log_2 \frac{1}{p_X(x)}$. Therefore, if the outcome is less likely, then it causes more surprise if it is observed, that is, it gives us more information. If we observe an outcome with probability 1, then there is no surprise, since it is the expected outcome, and consequently no information is obtained from that observation.

The *entropy* of X measures the expected amount of surprise, that is, the entropy H of X is defined as $H(X) = E[I_X] = \sum_{x \in \mathcal{X}} p_X(x) \log_2 \frac{1}{p_X(X)}$. This is a measure of the average uncertainty associated with a random variable, or, in other words, the average amount of information one obtains by observing the realization of a random variable.

A *code* C of a random variable X is a mapping from $\mathcal{X}$ to $\mathcal{D}^*$, where $\mathcal{D}$ is an alphabet of cardinality D and $\mathcal{D}^*$ is the set of finite-length strings of symbols from $\mathcal{D}$. Hence, the *encoding scheme* or *code* C defines how each *source symbol* $x \in \mathcal{X}$ is encoded. $C(x)$ is called the *codeword* corresponding to x. This codeword is composed by one or more *target symbols* from the *target alphabet* $\mathcal{D}$. The most used target alphabet is $\mathcal{D} = \{0, 1\}$, with $D = 2$, generating binary codes. The number of elements of $\mathcal{D}$ (i.e., D) determines the number of bits (b) that are needed to represent a symbol in the target alphabet $\mathcal{D}$. For instance, bit-oriented codewords, which are sequences of bits, require $b = 1$ bits to represent each of the $D = 2^1$ elements of $\mathcal{D}$. Byte-oriented codewords, which are sequences of bytes, require $b = 8$ bits to represent each one of the $D = 2^8$ elements of the target alphabet $\mathcal{D}$. Different codewords might have different lengths. Let us denote $l(x)$ the length of the codeword $C(x)$, then the expected length of a code C is given by the expression $L(C) = \sum_{x \in \mathcal{X}} p_X(x) l(x)$.

Given a *message* consisting in a finite string of source symbols, the *extension* of a code C is the mapping of that message to a finite string of target symbols. It is obtained by the concatenation of the individual codewords for each source symbol of the message. Hence, $C(x_1, x_2, ..., x_n) = C(x_1)C(x_2)...C(x_n)$. *Coding* consists in substituting each source symbol that appears in the input string by the codeword associated to that source symbol according to the *encoding scheme*. The process of recovering the source symbol that corresponds to a given codeword is called *decoding*.

A code is a *distinct code* if each codeword is distinct from any other, that is, if $x_1 \neq x_2$, $x_1, x_2 \in \mathcal{X}$, then $C(x_1) \neq C(x_2)$. A code is said to be *uniquely decodable* if every codeword is identifiable from a sequence of codewords. A uniquely decodable code is called a *prefix code* (or prefix-free code) if no codeword is a proper prefix of any other codeword. Prefix codes are *instantaneously decodable*, that is, an encoded message can be partitioned into codewords without the need of any *lookahead* examining subsequent code symbols. This property is important, since it enables decoding a codeword without having to inspect the following codewords in the encoded message, so being instantaneously decodable improves the decoding speed. A code is said to be an *optimal* code if it is instantaneous and has minimum average length, given the source symbols and their probabilities. The entropy gives us a lower bound on average code length for any uniquely decodable code of our data stream.

Example Let us consider three different codes C_1, C_2 and C_3 that map the source symbols from the source alphabet $\mathcal{X} = \{a, b, c, d\}$ to target symbols from the target alphabet $\mathcal{D} = \{0, 1\}$ as follows:

x	a	b	c	d
$C_1(x)$	1	11	10	101
$C_2(x)$	00	10	11	110
$C_3(x)$	00	10	111	110

C_1, C_2 and C_3 are distinct codes, since they map from source symbols to codewords one-to-one. However, not all of them are uniquely decodable or prefix free codes. For instance, the following three strings $''aaa''$, $''ab''$ and $''ba''$ map to the target string $''111''$ using code C_1. Hence, C_1 is not uniquely decodable. C_2 is a uniquely decodable code but not a prefix free code, since $C_2(c)$ is prefix of $C_2(d)$. For example, we can uniquely decode string $''11000010''$ to the sequence $''caab''$, but a lookahead is required to obtain the original sequence, since the first bits could be decoded to $''da''$ or $''ca''$. However, by analyzing all the binary string we can observe that the unique valid input sequence is $''caab''$. C_3 is a prefix code, since no codeword is a prefix of another. A string starting by $''1100000\ldots''$ can be univocally and instantaneously decoded to $''daa''$ without examining the following codewords.

2.1.1 Entropy in context-dependent messages

We can encode symbols depending on the context in which they appear. Until now, we have assumed independence of source symbols and their occurrences. However, it is usually possible to model the probability of the next source symbol x in a more precise way, by using the source symbols that have appeared before x.

We define the *context* of a source symbol x as a fixed-length sequence of source symbols that precede x. When the *context* has length m, that is, is formed by the precedent m symbols, we can use an m*-order model.*

The entropy can be defined depending on the order of the model, so that the kth-order entropy H_k is defined as follows:

- *Base-order models* assume an independent uniform distribution of all the source symbols. Hence, $H_{-1} = \log_D n$.
- *Zero-order models* assume independence of the source symbols, whose frequencies are their number of occurrences. Therefore, the zero-order entropy

is defined as already explained, $H_0 = -\sum_{x\in\mathcal{X}} p_X(x) \log_D p_X(x)$.

- *First-order models* obtain the probability of occurrence of the symbol y conditioned by the previous occurrence of the symbol x ($P_{y|x}$) and compute the entropy as: $H_1 = -\sum_{x\in\mathcal{X}} p_X(x) \sum_{y\in\mathcal{X}} P_{y|x} \log_D(P_{y|x})$.
- *Second-order models* model the probability of occurrence of the symbol z conditioned by the previous occurrence of the sequence yx ($P_{z|yx}$) and compute the entropy as:
$H_2 = -\sum_{x\in\mathcal{X}} p(x) \sum_{y\in\mathcal{X}} P_{y|x} \sum_{z\in\mathcal{X}} P_{z|y,x} \log_D(P_{z|y,x})$.
- *Higher-order models* follow the same idea.

Several distinct m-order models can be combined to estimate the probability of the next source symbol. For instance, *Prediction by Partial Matching* (PPM) [CW84, Mof90, BCW90] is a compression technique that combines several finite-context models of order 0 to m.

2.2 Redundancy and Data Compression

Data Compression aims at converting the string of bits that represents the data into a shorter string of bits, such that transmission, storage, or processing requirements are reduced. Compression techniques exploit redundancies in the source message to represent it using less space [BCW90], while maintaining the source information I_X. *Redundancy* is a measure of the difference between the average codeword length of the code used and the value of the entropy, that is, if $l(x)$ is the length of the codeword assigned to symbol x, redundancy can be defined as follows:

$$R = \sum_{x\in\mathcal{X}} p_X(x) l(x) - H = \sum_{x\in\mathcal{X}} p_X(x) l(x) - \sum_{x\in\mathcal{X}} -p_X(x) \ \log_D p_X(x)$$

Noticing that the entropy is determined by the distribution of probabilities of the source message, redundancy is decreased by reducing the average codeword length. A code is said to be a *minimum redundancy code* if it has minimum codeword length.

We now describe Huffman algorithm to construct a minimum-length prefix code.

2.2.1 Classic Huffman Code

The classic Huffman algorithm [Huf52] is a commonly used technique that generates optimal prefix codes for any source. The goal of Huffman code is that the bit rate of the encoded data can come close to the entropy of the data itself, which is achieved by using a code in which the length of each codeword is proportional to its frequency. Hence, the basic idea of Huffman coding consists in assigning short codewords to those symbols with high probabilities and long codewords to those with low probabilities (ideally, length $\log(1/p_i)$).

Huffman algorithm builds a tree that is used in the encoding process, and it generates prefix codes for each symbol. Huffman tree is a full tree where each leaf is associated to a codeword and every intermediate node has D child nodes. Classical Huffman tree is binary ($D = 2$), hence, every node of the tree has either zero or two children. The leaves are labeled with the weights that represent the probabilities associated with the source symbols. Their position (level) in the tree depends on their probability: the number of occurrences of a leaf in a higher level can never be smaller than the number of occurrences of a leaf placed in a lower level.

The Huffman tree is built as follows. Firstly, a list of leaf nodes is created, one node for each distinct input symbol, storing the frequency of the symbol for each node. This list is sorted by frequency. Then, the two least frequent nodes are removed from the list and a new internal node is created storing the sum of the frequencies of the removed nodes. This new node is added to the list. Then, the two least frequent nodes from the list are removed again and the procedure is repeated until there is just one node in the list. The last internal node created is the root of the Huffman tree (and its frequency will be the sum of occurrences of all the input symbols). Assuming that the source symbols are already sorted, the cost of building a Huffman tree is $O(n)$ [MK95] where n is the number of symbols (leaf nodes) in the tree.

Codewords are assigned to each leaf by setting to 0 the left branch of each node and to 1 the right branch. The path from the root node of the Huffman tree to the leaf node where a symbol appears gives the (binary) codeword of that symbol.

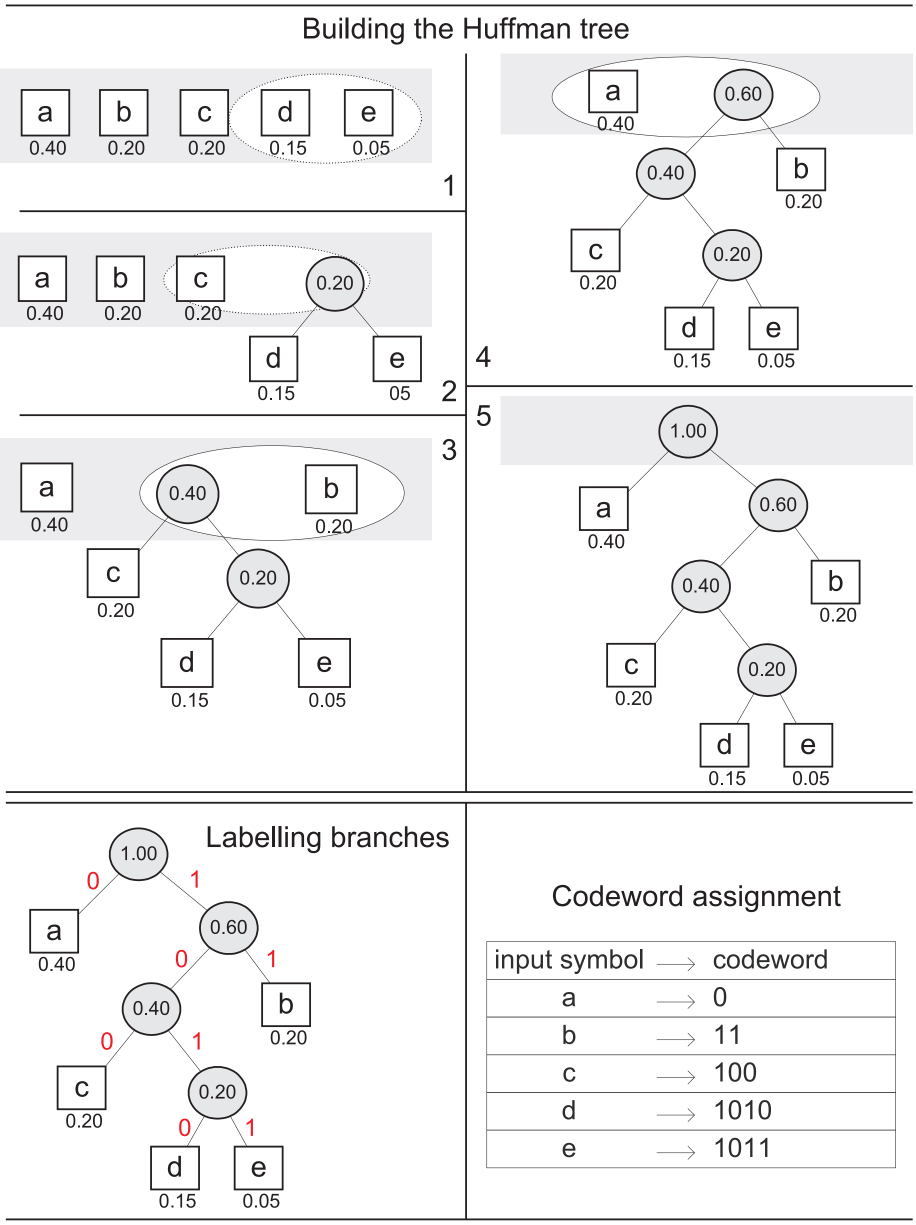
Example Figure 2.1 shows an example of Huffman tree built from a source alphabet $\{a, b, c, d, e\}$ with relative frequencies 0.40, 0.20, 0.20, 0.15 and 0.05 respectively. The figure illustrates the process step by step:

1. First, the list of nodes associated to the input symbols is created.

2. Then, the two least frequent nodes (d and e) are chosen, and they are joined into a new internal node whose frequency is the sum of the frequencies of the two chosen nodes, that is, 0.20.

3. Now, the least frequent nodes are b, c and the internal node just created, since all of them have frequency 0.20. Any two of them can be chosen in the next step. In our example we choose the internal node and c, and join them in a new internal node of frequency 0.40, which is added to the set.

4. The next step consists in joining the previous internal node and b into a new internal node, with frequency 0.60.



input symbol	→	codeword
a	→	0
b	→	11
c	→	100
d	→	1010
e	→	1011

Figure 2.1: Building a classic Huffman tree.

5. Finally, only two nodes remain to be chosen, which are joined into the root node. Notice that the weight associated to the root node is 1, since it represents the sum of the frequencies of all the source symbols.

The branches of the Huffman tree are labelled as explained, and codewords are assigned to the symbols as follows: $a \mapsto 0$, $b \mapsto 11$, $c \mapsto 100$, $d \mapsto 1010$, $e \mapsto 1011$.

The compressed file consists in the concatenation of the codewords for each source symbol of the message to encode. In addition, it must include a header representing the source alphabet and information about the shape of the Huffman tree, such that the decompressor can decode the compressed file. Then, the decompression algorithm reads one bit at a time and traverses the Huffman tree, starting from the root node, in order to obtain the source symbol associated with each codeword. Using the bit value read we can choose either the right or the left branch of an internal node. When a leaf is reached, a symbol has been recognized and it is output. Then the decompressor goes back to the root of the tree and restarts the process.

Canonical Huffman tree

Several Huffman trees can be built over the same sequence of source symbols and probabilities, generating different codes. Huffman's algorithm computes possible codewords that will be mapped to each source symbol, but only their lengths are relevant. Once those lengths are known, codewords can be assigned in several ways. Among all of them, the *canonical Huffman code* [SK64] is the most used one since its shape can be compactly stored.

The canonical code builds the prefix code tree from left to right in increasing order of depth. At each level, leaves are placed in the first position available (from left to right). The following properties hold:

- Codewords are assigned to symbols in increasing length order where the lengths are given by Huffman's algorithm.

- Codewords of a given length are consecutive binary numbers.

- The first codeword c_ℓ of length ℓ is related to the last codeword of length $\ell - 1$ by the equation $c_\ell = 2(c_{\ell-1} + 1)$.

The canonical Huffman tree can be compactly represented with only the lengths of the codewords. It only requires $O(h)$ integers, where h corresponds to the height of the Huffman tree. Hence, the header of the compressed file will include this information in addition to the source alphabet, which is stored sorted by frequency.

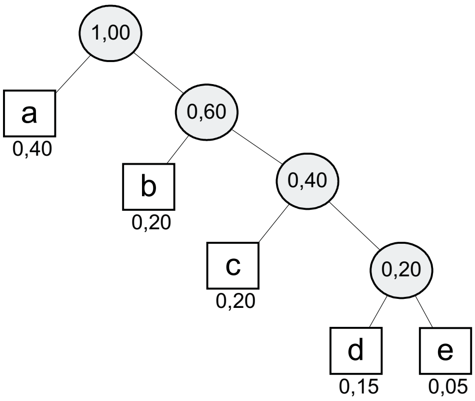


Figure 2.2: Example of canonical Huffman tree.

Figure 2.2 shows the canonical Huffman tree for the previous example. The codeword assignment is now $a \mapsto 0$, $b \mapsto 10$, $c \mapsto 110$, $d \mapsto 1110$, $e \mapsto 1111$.

We have only described the canonical representation for the bit-oriented Huffman approach, but it can also be defined for a byte-oriented approach. More details about how a byte-oriented canonical Huffman code can be built appear in [MK95, MT96].

2.2.2 Classification of compression techniques

Compression techniques represent the original message in a reduced space. This reduction can be performed in a lossy or a lossless way. If a *lossless compression technique* is used, then the data obtained by the decompressor is an exact replica of the original data. On the other hand, *lossy compression techniques* may recover different data from the original. These lossy methods are used in some scenarios, such as image or sound compression, where some loss of source information can be permitted during compression because human visual/auditive sensibility cannot detect small differences between the original and the decompressed data. Some other scenarios, such as text compression, require lossless techniques, and we will focus on them in this thesis.

We can classify compression techniques depending on how the encoding process takes place. Two families are defined: *dictionary* and *statistical* based techniques.

- *Dictionary techniques* replace substrings of the message with an index to an entry in a *dictionary*. Compression is achieved by representing several symbols as one output codeword. The best known examples of dictionary techniques are based on Ziv-Lempel algorithm [ZL77, ZL78], which uses the sliding window model and replaces substrings of symbols by pointers to previous occurrences of the same substring. They build a dictionary during the compression process to store the replaceable substrings. Hence, encoding consists in substituting those substrings, where found, by small fixed-length pointers to their position in the dictionary, and compression is achieved as long phrases are now represented with pointers occupying little space.

- *Statistical methods*, also called *symbolwise* methods, assign codewords to the source symbols such that the length of the codeword depends on the probability of the source symbol. Shorter codes are assigned to the most frequent symbols and hence compression can be achieved. The most known statistical methods are based on the Huffman codes [Huf52] and arithmetic methods [Abr63], and they differ mainly in how they estimate probabilities for symbols. Since statistical compression methods are recurrently used in this thesis, especially in Part II, we will explain these methods more in detail.

A statistical compression method starts by dividing the input message into symbols and estimating their probability. Once this first step is done, it obtains a *model* of the message and an *encoding scheme* can be used to assign a codeword to each symbol according to that representation. Therefore, compression can be seen as a *"modeling + coding"* process and different encoding schemes can be used for the same model of the message. Hence, the model of the message for a statistical technique consists of the vocabulary of different source symbols that appear in the message and their number of occurrences. A good modeling is crucial to obtain good compression, since better compression can be achieved if the estimations of the probabilities are made more accurately, as considering the context of the symbol, as seen in Section 2.1.1.

Depending on the model used, compression techniques can be classified as using a static, semi-static or dynamic model.

- If *static or non-adaptive models* are used, the assignment of frequencies to each source symbol is fixed. The encoding process employs pre-computed probability tables. These probabilities are generally extracted from experience and do not follow the real distribution of the source symbols in the input message, loosing some compression capabilities. However, they can be suitable in specific scenarios. An example of this approach occurs when transmitting data using *Morse* code.

- *Semi-static models* are usually used along with *two-pass* techniques. In the first pass, the whole message is processed to extract all the source symbols that conform the vocabulary and to compute their frequency distribution. Then, an encoding scheme is used to assign a codeword to each source symbol whose length depends on the frequency of the source symbol. In the second pass, the whole message is processed again and source symbols are substituted by their codewords. The compressed text is stored along with a header where the correspondence between the source symbols and codewords is represented. This header will be needed at decompression time. The best known examples are those based on Huffman-based codes [Huf52]. Some semi-static compression techniques for natural language text, such as are Plain Huffman and Tagged Huffman [MNZBY00], or those based on Dense Codes [BFNP07] or Restricted Prefix Bytes [CM05] will be explained in Chapter 7.

- *Dynamic or adaptive models* are usually known as *one-pass* techniques. Hence, they do not perform a first step to compute the frequencies of the source symbols of the message. They start with an initial empty vocabulary and then they read one symbol at a time. For each symbol read, a codeword is assigned depending on the current frequency distribution and its number of occurrences is increased. When a new symbol is read, it is appended to the vocabulary. Hence, the compression process adapts the codeword of each symbol to its frequency as compression progresses. The decompressor adapts the mapping between symbols and codewords in the same way as the compressor. Therefore, this mapping is not included with the compressed data. This property gives one-pass techniques their main advantage: their ability to compress message streams. These models are commonly used along with Ziv-Lempel algorithms [ZL77, ZL78, Wel84] and arithmetic encoding [Abr63, WNC87]. Another compression technique that usually uses adaptively generated statistics is PPM [CW84]. Some compression techniques based on Huffman codes and using a dynamic model have also been presented [Vit87].

2.2.3 Measuring the efficiency of compression techniques

In order to measure the efficiency of a compression technique we take into account two different aspects:

- The performance of the algorithms involved, which can be analyzed by the complexity of the compression and decompression algorithms. The theoretical complexity gives an idea of how a technique will behave, but it may be also useful to obtain empirical results such that we can compare the performance of the technique with other methods in real scenarios. We will measure their performance as *compression and decompression times*, which are usually measured in seconds or milliseconds.

- The compression achieved, which can be measured in many different ways. The most usual one consists in measuring the *compression ratio*. It represents the percentage that the compressed file occupies with respect to the original file size. Assuming that i is the size of the input file (in bytes), that the compressed file occupies o bytes, it is computed as: $\frac{o}{i} \times 100$.

2.3 Rank and select data structures

Succinct data structures aim at representing data (e.g., sets, trees, hash tables, graphs or texts) using as little space as possible while still being able to efficiently solve the required operations over the data. Those representations are able to approach the information theoretic minimum space required to store the original data. Even though these succinct data structures require more complex algorithms than the plain representation in order to retain the original functionality, they improve the overall performance as they can operate in faster levels in the memory hierarchy due to the reduction of space obtained.

One of the first presented succinct data structures consists in the bit-vectors supporting rank/select operations, which are the basis of other succinct data structures. We describe them more in detail. We will also describe some solutions to support these rank and select operations over arbitrary sequences.

2.3.1 Rank and select over binary arrays

Given an offset inside a sequence of bits $B_{1,n}$, we define three basic operations:

- $rank_b$ counts the number of times the bit b appears up to that position. Hence, $rank_0(B, i)$ returns the number of times bit 0 appears in the prefix $B_{1,i}$ and $rank_1(B, i)$ the number of times bit 1 appears in the prefix $B_{1,i}$. If no specification is made, $rank$ stands for $rank_1$ from now on.

- $select_b$ returns the position in that sequence where a given occurrence of bit b takes place. Hence, $select_0(B, j)$ returns the position i of the $j - th$ appearance of bit 0 in $B_{1,n}$ and $select_1(B, j)$. Analogously to $rank$, $select$ stands for $select_1$ if no bit specification is made.

- *access* operation allows one to know if a given position of the sequence contains 0 or 1.

The importance of these operations for the performance of succinct data structures has motivated extensive research in this field [MN07]. Full-text indexes are a good example in which the performance of these two operations is especially relevant [NM07].

Several strategies have been developed to efficiently compute *rank* and *select* when dealing with binary sequences. They are usually based on building auxiliary structures that lead to a more efficient management of the sequence.

The previously explained operations, *rank* and *select*, were defined by Jacobson in one of his first research works devoted to the development of succinct data structures [Jac89b]. In his paper, Jacobson proposed an implementation of *rank* and *select* that was able to compute *rank* in constant time and was used as the basis of a compact and efficient implementation of binary trees.

Given a binary sequence $B[1, n]$ of size n, a two-level directory structure is built. The first level stores $rank(i)$ for every i multiple of $s = \lfloor \log n \rfloor \lfloor \log n/2 \rfloor$. The second level stores $rank'(j)$ for every j multiple of $b = \lfloor \log n/2 \rfloor$, where $rank'(j)$ computes *rank* within blocks of size s. Notice that $s = b\lfloor \log n \rfloor$. To compute $rank_1(B, i)$ we can use these two directory levels to obtain the number of times the bit 1 appears before the block of size s containing the position i. The same happens in the second level of the directory structure. The final result is obtained using *table lookups*: the bits of the subsequence of size b containing the position i that could not be processed with the information of the directories, are used as the index for a table that indicates the number of times bit 1 or 0 appears in them. Therefore *rank* can be computed in constant time. However, with this approach *select* is computed in $O(\log \log n)$, since it has to be implemented using binary searches.

There are n/s superblocks in the first level of the directory, each of them requiring $\log n$ bits to store the absolute *rank* value, that is, the first level of the directory occupies a total space of $n/s \log n = O(n/ \log n)$ bits; there are n/b blocks in the second level of the directory, each of them requiring $\log s$ bits to store the relative *rank* value inside its superblock, that is, the second level of the directory occupies a total space of $n/b \log s = O(n \log \log n/ \log n)$ bits; and there is a lookup table that stores the *rank* value for each position within each stream of length b, that is, the table occupies $O(2^b \cdot b \cdot \log b) = O(\sqrt{n} \log n \log \log n)$ bits. Hence, the space needed by these additional directory structures, which is the sum of the space required by the first level, the second level and the lookup table, is $o(n)$.

Later works by Clark and Munro [Cla96, Mun96] improved these results, obtaining constant time implementations for *rank* and *select* and using $n+o(n)$ bits, where n bits are used for the binary sequence itself and $o(n)$ additional bits for the data structures answering rank and select queries. The solutions proposed by Jacobson, Clark and Munro [Jac89b, Cla96, Mun96] are based on the idea of using additional data structures for efficiently computing *rank* and *select* without taking into account the content of the binary sequence and its statistical properties (number of 1 bits and their positions in the sequence). Pagh [Pag99] and Raman, Raman and Rao [RRR02] explored a new approach working with compressed binary sequences which are also able to efficiently compute *rank* and *select*.

Pagh [Pag99] first explored the possibility of representing the sequence as a set of compressed blocks of the same size, each of them represented by the number of 1 bits it contains and the number corresponding to that particular subsequence. Since with this scheme the number of blocks grows almost linearly with the size of the sequence, an interval compression scheme that clusters suitable adjacent blocks together into intervals of varying length was also proposed.

The compressed representation of binary sequences proposed by Raman *et al.* [RRR02] is based on a numbering scheme. The sequence is divided into a set of blocks of the same size, each of them represented by the number of 1 bits it contains and an identifier, in such a way those blocks with few 1 bits or many 1 bits require shorter identifiers. More concretely, every block is represented as a tuple (c_i, o_i), where the first component, c_i, represents the class of the block, which corresponds to its number of 1s, and the second, o_i, represents the offset of that block inside a list of all the possible blocks in class c_i. If each block has length u, each c_i is represented with $\lceil \log(u+1) \rceil$ bits and each o_i is represented using $\lceil \log \binom{u}{c_i}) \rceil$ bits. This approximation obtains zero-order compression and is currently the best complete representation of binary sequences [MN07] (that is, it supports access, rank and select in constant time for both 0 and 1 bits). This paper also shows how this binary sequence data structure can be used for the optimal representation of k-ary trees and multisets.

Several practical alternatives achieving very close results have been proposed by Okanohara and Sadakane [OS07]. They present four novel rank/select directories: *esp*, *recrank*, *vcode* and *sdarray*, which support fast queries. Each of them is based on different ideas and has different advantages and disadvantages in terms of speed, size and simplicity. The size is small when the bitmap is sparse and can even approach to the zero-th order empirical entropy.

Another research line aims at compression of binary sequences when the number of 1 bits is small. The approach known as *gap encoding* obtains compressed representations of binary sequences encoding the gaps between consecutive 1 bits in the sequence. Several works [Sad03, GHSV06, MN07] present several developments and improvements for this approach.

Practical implementations Two different implementations of Rodrigo González [GGMN05] has been used in this thesis. One of them follows the constant-time classical solution proposed by Jacobson, where precomputed *popcount* tables are used. Popcounting consists in counting how many bits are set in a bit array. By using tables where this counting is already computed for small arrays of 8 bits, rank and select operations can be efficiently solved with a space overhead of 37.5% of the size of the bitarray. The other solution, which is also used in this thesis, requires a parameterizable extra space, which will be usually fixed to 5% of the size of the sequence. This reduction of the overhead is obtained by using a directory of just one level, that is, having a single level of blocks, one per $32 \cdot k$ bits. Notice that a sequential scan to count all the set bits in at most k blocks is required, while the space overhead is $1/k$. Hence, this alternative offers an interesting space/time tradeoff compared to the previous alternatives. We will use this solution with $k = 20$ such that just 5% of extra space is needed, while still computing rank and select in efficient time.

We have also used in Section 9.6.1 the compressed representation implemented by Francisco Claude[1] based on the proposal by Raman, Raman and Rao [RRR02].

2.3.2 Rank and select over arbitrary sequences

Rank, select and access operations can be extended to arbitrary sequences S with an alphabet Σ of size σ. In this case, given a sequence of symbols $S = S_1 S_2 \ldots S_n$ and a symbol $s \in \Sigma$:

- $rank_s(S, i)$ returns the number of times the symbol s appears in the sequence up to position i, that is, in $S[1, i]$.
- $select_s(S, j)$ returns the position of S containing the j-th occurrence of the symbol s.
- $access(S, i)$ returns the i-th symbol of sequence S, that is, S_i. It may be a necessary operation, since S is commonly represented in a compact way.

The strategies used with binary sequences cannot be directly applied to the general case or, if applied, they may require a significant amount of memory. For some scenarios, they can be efficiently adapted, as we probed in Section 9.2, where we propose a simple representation of byte sequences using these approaches. However, rather than directly applying those techniques, most of the approaches for the general case try to adapt them or to transform the problem in such a way that it can be reduced to using *rank* and *select* in binary sequences. We now describe some of the approaches to this problem that are based on the use of binary sequences.

Constant time *rank* and *select* using *bitmaps*

The easiest way to efficiently compute *rank* and *select* in arbitrary sequences consists in using indicator *bitmaps* (binary sequences) for each symbol of the alphabet of symbols Σ [NM07]. For each position of the original sequence, only the bitmap corresponding to its symbol has a 1 bit in that position. Therefore, as we can compute *rank* and *select* over binary sequences in constant time, we can also do it in the case of sequences of bytes. The price to pay for this efficient implementation

[1] It is available at the Compact Data Structures Library (libcds)[2].

is the space used by the bitmap for each symbol of the alphabet Σ and the necessary additional data structures for computing *rank* and *select* in constant time in each one of them. One can use Okanohara and Sadakane rank/select solution to represent each bitmap to reduce the space, at the expense of degrading access time, which must be solved in $O(\sigma)$ time.

Wavelet trees

A wavelet tree [GGV03] consists in a balanced binary tree that divides the alphabet Σ into two parts at each node. In consequence, each symbol from an alphabet $\Sigma = \{s_1, s_2, \ldots, s_\sigma\}$ is associated to a leaf node.

Given a sequence $S = S_1 \ldots S_n$ composed of symbols from the alphabet Σ, a wavelet tree is built as follows. The root of the tree is given a bitmap $B = b_1 \ldots b_n$ of the same length (n) as the sequence of symbols, such that $b_i = 0$ if $S_i \in \{s_1, \ldots, s_{\sigma/2}\}$, and $b_i = 1$ if $S_i \in \{s_{\sigma/2+1}, \ldots, s_n\}$. Those symbols given a 1 in this vector are processed in the right child of the node, and those marked 0 are processed in the left child of the node. This process is repeated recursively in each node until reaching the leaf nodes when the sequence is a repeat of one symbol. In this way, each node indexes half the symbols (from Σ) indexed by its parent node. Each node stores only its bitmap B, and the portion of the alphabet that it covers can be obtained by following the path from the root of the tree to that node. With this information it is not necessary to store the sequence separately, since it can be recovered from these bitmaps.

Figure 2.3 shows a simple example with a sequence of symbols from the alphabet $\Sigma = \{a, b, c, d\}$ (the text is shown only for clarity, but it is not actually stored).

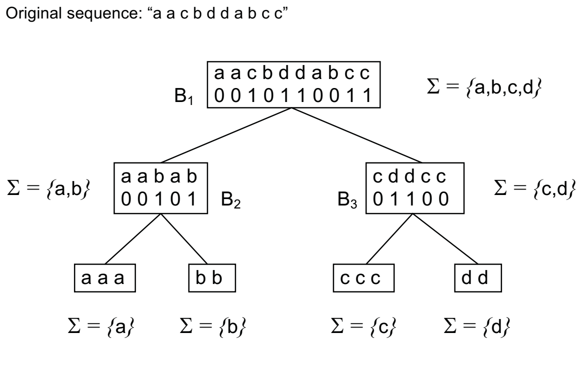


Figure 2.3: Example of wavelet tree.

Access and rank queries over the sequence S are solved via $\log \sigma$ binary ranks over the bitmaps, by performing a top-down traversal of the wavelet tree, while a select query is similarly solved by performing $\log \sigma$ bottom-up binary selects.

Access query: Let us obtain the symbol at position i of the sequence. Bit b_i of the bitmap B at the root node determines whether the symbol is indexed in the left child ($b_i = 0$) or in the left child ($b_i = 1$) of the root node. Besides, $rank_{b_i}(B, i)$ returns the position of the symbol in the bitmap of the child. This process is repeated until the last level is reached. The leaf reached using this procedure corresponds to the symbol represented at the desired position of the sequence.

Rank query: Obtaining the number of times that a symbol s appears up to the position i of the sequence is solved in a similar way as *access*. By applying a binary *rank* in the bitmap of each node we obtain the position in which the binary *rank* is applied at the next level of the tree. Let the path from the root node to the leaf node for symbols s be $b_0 b_1 \dots b_k$ and the bitmaps stored in the nodes from the path be $B^0, B^1, \dots, B^k$. We first count how many times b_0 appears up to position i, that is, $rank_{b_0}(B_0, i)$. This gives us the position i_1 of the second level of the wavelet tree that is used for the rank operation in this level. Hence, we perform $rank_{b_1}(B_1, i_1)$ at the second level to count how many times b_1 appears up to position i_1. We proceed computing rank operations $rank_{b_k}(B_k, i_k)$ up to the last level of the tree. At the leaf, the final bitmap position corresponds to the answer to $rank(s, i)$ in the sequence S.

Select query: Each symbol of the sequence is represented by one unique leaf in the tree. Hence, to compute the position of the i-th occurrence of symbol s in sequence S, we start from the leaf node that represents this symbol s (this leaf node is determined by its position in the alphabet Σ) and traverse the tree up to the root node. Let the path from the root node to the leaf node be $b_0 b_1 \dots b_k$ and the bitmaps stored in the nodes from the path be $B^0, B^1, \dots, B^k$. To compute $select(s, i)$ we proceed as follows. First we calculate $i_k = select_{b_k}(B^k, i)$, so that i_k is the position of the i-th occurrence of s in B^k. We repeat this step in the previous level, obtaining $i_{k-1} = select_{b_{k-1}}(B^{k-1}, i_k)$, and move through the levels of the tree up to the root node. The last $i_0 = select_{b_0}(B^0, i_1)$ returns the position of the i-th occurrence of s in the sequence S.

A practical variant to zero-th order entropy is to give the wavelet tree the shape of the Huffman tree of the sequence or using Raman, Raman and Rao data structures for rank/select operations [GGV03, NM07].

Golynski, Munro, and Rao solution

Golynski, Munro, and Rao [GMR06] presented another representation for arbitrary sequences that supports rank and access operations in $O(\log \log \sigma)$ time and select in $O(1)$; alternatively, they can achieve $O(1)$ time for access, $O(\log \log \sigma)$ for select, and $O(\log \log \sigma \log \log \log \sigma)$ for rank. It uses $n \log \sigma + no(\log \sigma)$ bits.

Their representation follows a similar idea to the one that uses bitmaps for each symbol of the alphabet. Hence, the sequence S is represented using a table T of size $\sigma \times n$ where rows are indexed by $1, \ldots, \sigma$ and columns by positions in the sequence, that is, from 1 to n. One entry of this table $T[s, i]$ indicates whether the symbol $s \in \Sigma$ occurs in position i in the sequence S. A large bitmap A is built by writing T in row major order (note that $|A| = \sigma \cdot n$). A is then divided into blocks of size σ. Restricted versions of rank and select are defined and implemented to answer the operations over these blocks. Sequence A is not directly stored. Instead, a bitmap B is built by writing the cardinalities of all the blocks in unary, that is, the number of 1s inside each block, such that if k_i is the cardinality of block i, then B is built as $B = 1^{k_1}01^{k_2}0 \ldots 1^{k_n}0$. Each block is then represented using two sequences. One of them indicates the positions of all the occurrences for each symbol s in the block in alphabetical order. Hence, this sequence consists in a permutation of the sequence of block positions $1, \ldots, \sigma$. The other sequence is a bitmap called X, of length 2σ. The bitmap X stores the multiplicity of each symbol s inside the block, that is, $X = 01^{l_1}01^{l_2}0 \ldots 1^{l_\sigma}$, where l_s is the multiplicity of symbol s in the block. If these structures are used, rank, select and access operations can be efficiently solved by first locating the corresponding block by means of restricted rank and select operations over sequence B and then examining the block using the permutation and the bitmap X. A complete description of the method can be read in the original paper [GMR06].

Part I

Directly Addressable Variable-Length Codes

Chapter 3

Introduction

Variable-length coding is at the heart of Data Compression [Sto88, BCW90, WMB99, MT02, Sol07]. It is used, for example, by statistical compression methods, which assign shorter codewords to more frequent symbols. It also arises when representing integers from an unbounded universe: Well-known codes like γ-codes and δ-codes, which will be explained in this chapter, are used when smaller integers are to be represented using fewer bits.

A problem that frequently arises when variable-length codes are employed to encode a sequence of symbols is that it is not possible to access directly the i-th encoded element, because its position in the encoded sequence depends on the sum of the lengths of the previous codewords. This is not an issue if the data is to be decoded from the beginning, as in many compression methods. Yet, the issue arises recurrently in the field of compressed data structures, where the compressed data should be randomly accessible and manipulable in compressed form. A partial list of structures that may require direct access to variable-length codes includes Huffman [Mof89] and other similar encodings of text collections, such as Plain Huffman and Tagged Huffman [MNZBY00], End-Tagged Dense Code and (s,c)-Dense Codes [BFNP07] or Restricted Prefix Byte Codes [CM05], compression of inverted lists [WMB99, CM07], compression of suffix trees and arrays (for example the Ψ function [Sad03] and the LCP array [FMN09]), PATRICIA tree skips [Kre10], compressed sequence representations [RRR02, FV07], partial sums [MN08], sparse bitmaps [RRR02, OS07, CN08] and its applications to handling sets over a bounded universe supporting predecessor and successor search, and a long so on. It is indeed a common case that an array of integers contains mostly small values, but the need to handle a few large values makes programmers opt for allocating the maximum space instead of seeking for a more sophisticated solution.

The typical solution to provide direct access to a variable-length encoded sequence is to regularly sample it and store in an array the position of the samples in the encoded sequence, so that decompression from the last sample is necessary. This introduces a space and time penalty to the encoding that often hinders the use of variable-length coding in many cases where it would be beneficial.

In this part of the thesis we propose a new variable-length encoding scheme for sequences of integers that supports direct access without using any extra sampling structure. This new encoding can be efficiently applied to sequences of integers when the number of occurrences of smaller integer values is higher than the frequency of larger integer values, or, more generally, to any sequence of symbols after it has been mapped to a sequence of integers according to the frequency distribution obtained by a statistical modeler[1].

Our proposal, called *Directly Addressable Codes (DACs)*, consists in using a dense variable-length encoding to assign codewords to the integers of the sequence, where each codeword is composed of several chunks. The number of chunks of the assigned codeword depends on the magnitude of the integer, so that smaller integers obtain shorter codewords. Once the codewords have been assigned, their chunks are rearranged in several levels of a data structure in such a way that direct access to any codeword of the encoded sequence is possible. DACs are explained in detail in Chapter 4.

Moreover, the rearrangement strategy proposed, where the different chunks of the codewords are placed in several levels of a data structure, can be seen as a contribution by itself, which provides synchronism to the encoded sequence of symbols obtained after using any variable-length encoding technique. Hence, direct access to any codeword of the encoded sequence is achieved by just using some extra bitmaps. Some examples of this usage are described in Chapter 6.

In this chapter, we study the direct access problem when variable-length codes are used. We have briefly presented the need and usage of these variable-length encoding schemes. The chapter continues by describing some encoding schemes for integers in Section 3.1, and finishes by enumerating some solutions from previous works in Section 3.2.

[1] More concretely, this mapping is done as follows. Firstly, a vocabulary with the different symbols that appear in the sequence is created. Then, this vocabulary is sorted by frequency in decreasing order, hence, the most frequent symbols become the first symbols of the vocabulary. Finally, the sequence of symbols is translated into a sequence of integers by substituting each symbol by the position of the symbol in the sorted vocabulary. With this procedure, a sequence of integers is obtained such that smaller integer values, corresponding to more frequent symbols, appear more times than larger integer values.

3.1 Encoding Schemes for Integers

Let $X = x_1, x_2, \ldots, x_n$ be the sequence of n integers to encode. A way to compress X is to use statistical compression, that is, we order the distinct values of X by frequency, such that a vocabulary of the different integers that appear in the sequence is extracted, and assign shorter codewords to those values x_i that occur more frequently. For instance, we can use Huffman encoding. However, in some applications the smaller values are assumed to be more frequent. In this case one can directly encode the numbers with a fixed instantaneous code that gives shorter codewords to smaller numbers. This strategy has the advantage that it is not necessary to store the vocabulary of symbols sorted by frequency, which may be prohibitive if the set of distinct numbers is too large. Well-known examples are unary codes, γ-codes, δ-codes, and Rice codes [WMB99, Sol07]. Table 3.1 shows how these four techniques, which are explained next, encode the first 10 integers. For the description of each encoding we assume that the integer to encode, x, is a positive integer, that is, $x > 0$.

Unary Codes The unary representation is commonly used within other encodings. A value x is represented as $1^{x-1}0$, that is, $x - 1$ ones followed by a zero. For instance, if we want to encode the integer 5 the codification would be 11110. The final zero allows to delimit the code making it a prefix-free code. Notice that the ones and zeros are interchangeable without loss of generality, so x can be represented as $0^{x-1}1$ as well.

Gamma Codes The γ-code of a given integer x consists in the concatenation of the length of its binary representation in unary code, and the binary representation of x omitting the most significant bit. The codification for 5, whose binary representation is $(101)_2$, would be 11001, where the first bits 110 represent 3 (the length of the binary representation) in unary code and the last bits 01 represent the symbol 101 without its most significant bit. The representation of a symbol x uses $2\lfloor \log x \rfloor + 1$ bits, where $\lfloor \log x \rfloor + 1$ are used to represent the symbol length in unary code and the other $\lfloor \log x \rfloor$ bits are used to represent the symbol without its most significant bit.

Delta Codes The δ-codes are the natural extension of γ-codes for larger symbols. They represent the binary length of the symbol using γ-codes instead of using unary codes. The rest of the encoding does not change. Hence, the representation of a symbol x uses $1 + 2\lfloor \log \log x \rfloor + \lfloor \log x \rfloor$ bits. For example, the representation of the integer 5 will be exactly the same as the one using gamma codes, except for the representation of the length of the symbol, which is encoded with γ-codes now: 10101 (the length of its binary representation, 3, is represented as 101 in γ-encoding instead of using unary codes).

Symbol	Unary code	γ-code	δ-code	Rice code (b = 2)
1	0	0	0	000
2	10	100	1000	001
3	110	101	1001	010
4	1110	11000	10100	011
5	11110	11001	10101	1000
6	111110	11010	10110	1001
7	1111110	11011	10111	1010
8	11111110	1110000	11000000	1011
9	111111110	1110001	11000001	11000
10	1111111110	1110010	11000010	11001

Table 3.1: Examples of variable length encodings for integers 1 to 10.

Rice Codes Rice codes are parameterized codes that receive two values, the integer x to encode and a parameter b. Then x is represented by the concatenation of the quotient and the remainder of the division by 2^b, more precisely, as $q+1$ in unary, where $q = \lfloor (x-1)/2^b \rfloor$, concatenated with $r = x - q \cdot 2^b - 1$ in binary using b bits, for a total of $\lfloor (x-1)/2^b \rfloor + b + 1$ bits.

If we could assign just the minimum bits required to represent each number x_i of the sequence X, the total length of the representation would be $N_0 = \sum_{1 \leq i \leq n} (\lfloor \log x_i \rfloor + 1)$ bits. For example, δ-codes are instantaneous and it can be proved that they achieve a total length $N \leq N_0 + 2n \log \frac{N_0}{n} + O(n)$ bits.

3.1.1 Vbyte coding

Vbyte coding [WZ99] is a particularly interesting code for this thesis. In its general variant, the code is obtained from the $\lfloor \log x_i \rfloor + 1$ bits needed to represent each integer x_i. The binary representation is split into blocks of b bits and each block is stored into a *chunk* of $b+1$ bits. This extra bit, corresponding to the highest bit of the chunk, is 0 if the chunk contains the most significant bits of x_i, and 1 for the rest of the chunks.

For clarity we write the chunks from most to least significant, just like the binary representation of x_i. For example, if we want to represent $x_i = 25$, being $b = 3$ the value of the value of parameter b, then we split the $\lfloor \log 25 \rfloor + 1 = 5$ bits needed for the binary representation of 25, which is $x_i = 25 = 11001_2$, into 2 blocks of 3 bits: the three most significant bits 011 and the three least significant bits 001. For the final Vbyte representation, one extra bit is added to each block, obtaining two chunks of 4 bits each. Hence, the final representation of the integer 25 using Vbyte codes is $\underline{0}011\ \underline{1}001$, where we underlined the extra bits that indicate whether the block contains the most significant bits of the binary representation or not.

Compared to an optimal encoding of $\lfloor \log x_i \rfloor + 1$ bits, this code loses one bit per b bits of x_i, plus possibly an almost empty final chunk, for a total space of $N \leq \lceil N_0(1+1/b) \rceil + nb$ bits. The best choice for the upper bound is $b = \sqrt{N_0/n}$, achieving $N \leq N_0 + 2n\sqrt{N_0/n}$, which is still worse than δ-encoding's performance. In exchange, Vbyte codes are very fast to decode.

The particular case of Vbyte that uses chunks of 8 bits is called *byte codes*. Its decoding function is very fast due to the byte-alignments, becoming an interesting alternative when time efficiency is demanded. However, since this variable-byte coding uses at least 8 bits to encode each integer, using byte codes may not be as space efficient as using a variable-bit scheme.

3.2 Previous Solutions to Provide Direct Access

From the previous section we end up with a sequence of n concatenated variable-length codewords that represent a sequence of integer values. Being usually instantaneous, there is no problem in decoding them in sequence, starting from the first position and decoding all the integers in order. However, there are multiple applications where this is not enough, and direct access to random positions of the encoded sequence is needed. We now outline several solutions proposed in the literature to solve the problem of providing direct access to the sequence, that is, extracting any x_i efficiently, given i.

3.2.1 The classical solution: Sparse sampling

The most commonly used solution consists in sampling the encoded sequence and storing absolute pointers only to the sampled codewords, that is, to each h-th codeword of the sequence. Access to the $(h \cdot k + d)$-th element, for $0 \leq d < h$, is done by decoding d codewords starting from the k-th sample. This involves a space overhead of $\lceil n/h \rceil \lceil \log N \rceil$ bits, where N is the length in bits of the concatenated variable-length codewords that represent the sequence of n integers, and a time overhead of $O(h)$ to access an element, assuming we can decode each symbol in constant time.

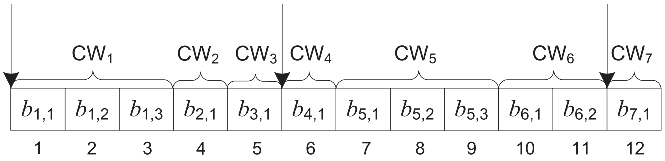

CW1
CW2
CW3
CW4
CW5
CW6
CW7
$b_{1,1}$ $b_{1,2}$ $b_{1,3}$ $b_{2,1}$ $b_{3,1}$ $b_{4,1}$ $b_{5,1}$ $b_{5,2}$ $b_{5,3}$ $b_{6,1}$ $b_{6,2}$ $b_{7,1}$
1 2 3 4 5 6 7 8 9 10 11 12

Figure 3.1: Example of sparse sampling.

Example Assume a sequence of seven integers that has been encoded obtaining the following sequence of codewords:

$$\langle b_{1,1}b_{1,2}b_{1,3}\rangle, \langle b_{2,1}\rangle, \langle b_{3,1}\rangle, \langle b_{4,1}\rangle, \langle b_{5,1}b_{5,2}b_{5,3}\rangle, \langle b_{6,1}b_{6,2}\rangle, \langle b_{7,1}\rangle,$$

that is, the first integer is represented with a codeword of three bits $CW_1 = b_{1,1}b_{1,2}b_{1,3}$, the second integer with just one bit and so on.

If we sample the starting position of every $h = 3$ codewords, then we store the positions of the first bit of the codewords of the first, fourth and seventh integers, which correspond with the following positions: 1, 6, 12. Figure 3.1 illustrates the sparse sampling over this example, where the arrows point to the positions where the sampled codewords start.

Now, if we want to extract the sixth element of the original sequence, we locate the nearest previous sample and start decoding all the integers from that sampled position until we decode the sixth integer. In this case, the nearest previous sample is the second one, corresponding to the fourth integer. There is a pointer to the sixth bit of the encoded sequence, which is where the codeword of the fourth integer starts. Therefore, we decode the fourth codeword from the encoded sequence, also the fifth codeword, and finally the sixth, obtaining the sixth element of the original sequence.

3.2.2 Dense sampling

A different strategy is used by Ferragina and Venturini [FV07], who propose a dense sampling approach to directly access the elements of the encoded sequence.

They represent each integer x_i using just $\lfloor \log x_i \rfloor$ bits and encode the set of integers consecutively following the series of binary strings $\{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$, that is, the infinite sequence of binary strings ordered first by length and then lexicographically by their content, with ϵ denoting the empty string. The integer $x_i = 1$ is represented with 0 bits (with the empty string ϵ). Then, the integer 2 is encoded with the next string "0", 3 with "1", 4 with "00" and so on.

In addition to the sequence of codewords that represent the sequence of integers, they store pointers to the beginning of the codeword of *every* element in the encoded sequence. The encoding scheme is not a uniquely decodable code, but the

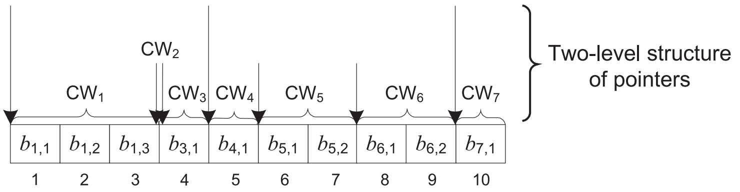

CW$_1$
CW$_2$
CW$_3$
CW$_4$
CW$_5$
CW$_6$
CW$_7$
Two-level structure of pointers
$b_{1,1}$ $b_{1,2}$ $b_{1,3}$ $b_{3,1}$ $b_{4,1}$ $b_{5,1}$ $b_{5,2}$ $b_{6,1}$ $b_{6,2}$ $b_{7,1}$
1 2 3 4 5 6 7 8 9 10

Figure 3.2: Example of dense sampling.

set of pointers gives the beginning and ending positions of each codeword, so any codeword can be decoded without ambiguity in constant time. For instance, two consecutive pointers which point at the same position of the encoded sequence indicate that the codeword has length 0, hence, the integer represented in that position is 1.

A dense sampling implies a large number of pointers. To reduce the space requirements, two levels of pointers are used: absolute pointers every $\Theta(\log N)$ values and relative ones for the rest. Then, the extra space for the pointers is $O(n(\log\log N + \log L))$, being L the length in bits of the longest codeword, and constant-time access is achieved.

Example Assume another sequence of seven integers. The second integer of this sequence is integer 1, so it is encoded with 0 bits (we represent this empty string of bits as ϵ). The rest of the encoded sequence is composed by codewords of different length, as follows:

$$\langle b_{1,1}b_{1,2}b_{1,3}\rangle, \langle\epsilon\rangle, \langle b_{3,1}\rangle, \langle b_{4,1}\rangle, \langle b_{5,1}b_{5,2}\rangle, \langle b_{6,1}b_{6,2}\rangle, \langle b_{7,1}\rangle$$

Using a dense sampling we set pointers to the start of each codeword. This solution is shown in Figure 3.2, where the two-level structure of pointers has been represented with large arrows for the absolute pointers and short arrows for the relative ones.

Now, if we want to extract the sixth integer from the encoded sequence, we must decode the bits represented between the sixth and the seventh pointers. This string of bits can be decoded univocally (its length is extracted from the position of the pointers) in constant time. All the other integers can be obtained in a similar way. For instance, to extract the second integer we proceed as follows: since the second and third pointers point to the same position of the encoded sequence, the second codeword is an empty string of bits, thus the element encoded is integer 1.

3.2.3 Elias-Fano representation of monotone sequences

Given a monotonically increasing sequence of positive integers $X = x_1, x_2, \ldots, x_n$, where all the elements of the sequence are smaller than an integer u, the Elias−Fano representation [Eli74, Fan71] uses at most $2 + \log(u/n)$ bits per element and also permits to directly access any element of the given sequence.

The representation separates the lower $s = \lceil \log(u/n) \rceil$ bits of each element from the remaining upper bits, and stores the lower bits contiguously in a bit array. In addition, the upper bits are represented in a bit array of size $n + x_n/2^s$, setting the bit at position $x_i/2^s + i$ for each $i \leq n$.

Note that we can represent a monotonically increasing sequence $Y = y_1, y_2, \ldots, y_n$, which may include some zero-differences, by setting bits $i + y_i$ in a bit array B of length $B[1, y_n + n]$. Hence, the integer at position i of the sequence Y can be retrieved by computing $y_i = select(B, i) - i$.

Example Let us suppose the following sequence $Y = 4, 8, 15, 15, 23, 42$, where the third and the fourth integers are equal. We can represent this sequence using a bitmap B of length $42 + 6 = 48$ and setting up the bits at positions $4 + 1 = 5$, $8 + 2 = 10$, $15 + 3 = 18$, $15 + 4 = 19$, $23 + 5 = 28$ and $42 + 6 = 48$. Then, the bitmap B is the following:

$$B = 000010000100000001100000000100000000000000000001$$

We can retrieve the third element of the sequence by performing a select operation $y_3 = select(B, 3) - 3 = 18 - 3 = 15$.

Hence, this solution is used by the Elias−Fano representation for the monotone sequence of the upper bits $x_i/2^s$, $i \leq n$, supporting their random retrieval. As the lowest bits are stored using a fixed-length representation, the overall representation of any element of the original sequence X is directly addressable if the bitmap that represents the upper bits supports select operations.

In general, we can represent any sequence of integers, including non-monotone sequences. We just must store the binary representation of each element, excluding the most significant one, concatenated consecutively in a bit array. In addition, we store the positions of the initial bit of each element using the Elias−Fano representation for monotone lists explained above. This techniques achieves high compression when the numbers are generally small. In the worst case scenario, the method does not lose more than one bit per element, plus lower-order terms, compared to N_0.

Chapter 4

Our proposal: Directly Addressable Codes

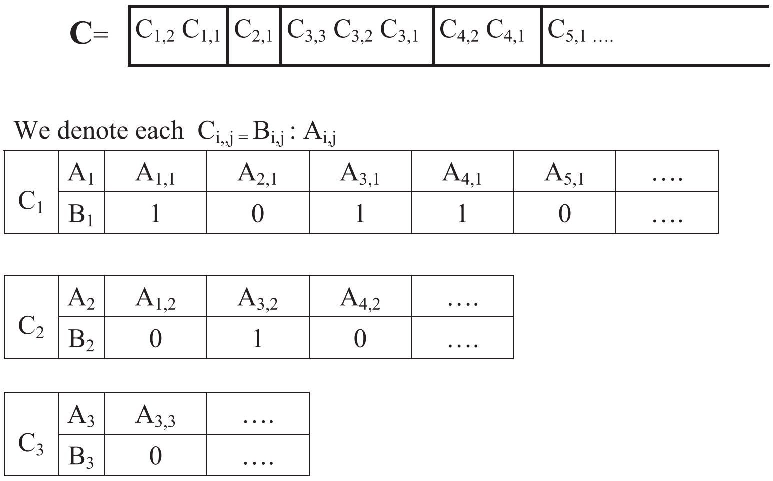
In this chapter we introduce a new variable-length encoding scheme for sequences of integers, called *Directly Addressable Codes (DACs)*, based on a relocation of the chunks of the Vbyte codification, explained in Section 3.1.1, into different levels. Our proposal enables direct access to the i-th codeword of the sequence without the need of any sampling method.

We explain our encoding scheme in Section 4.1, describing the rearrangement we perform to provide direct access to any element of the encoded sequence. We conclude the chapter by presenting in Section 4.2 an algorithm that computes the configuration for the values of the parameters of our proposal that achieves the most compact space.

4.1 Conceptual description

Given a sequence of integers $X = x_1, x_2, \dots, x_n$ we describe a new synchronized encoding scheme that enables direct access to any element of the encoded sequence.

We make use of the generalized Vbyte coding described in Section 3.1.1 with a given parameter b for the size of the blocks. We first encode the x_is into a sequence of $(b+1)$-bit chunks. Next we separate the different chunks of each codeword in different streams. Let us assume that a codeword CW_i is assigned to the integer x_i, such that CW_i needs r chunks $C_{i,r}, \dots, C_{i,2}, C_{i,1}$, where $C_{i,r}$ is the most significant chunk. A first stream, C_1, will contain the $n_1 = n$ least significant chunks (i.e., the $C_{i,1}$ rightmost chunks) of every codeword. A second one, C_2, will contain the n_2 second chunks of every codeword (where n_2 is the number of codewords using more



C= | $C_{1,2}$ $C_{1,1}$ | $C_{2,1}$ | $C_{3,3}$ $C_{3,2}$ $C_{3,1}$ | $C_{4,2}$ $C_{4,1}$ | $C_{5,1}$ ....

We denote each $C_{i,j} = B_{i,j} : A_{i,j}$

C_1	A_1	$A_{1,1}$	$A_{2,1}$	$A_{3,1}$	$A_{4,1}$	$A_{5,1}$	....
	B_1	1	0	1	1	0	....

C_2	A_2	$A_{1,2}$	$A_{3,2}$	$A_{4,2}$	....
	B_2	0	1	0	....

C_3	A_3	$A_{3,3}$	....
	B_3	0	....

Figure 4.1: Rearrangement of codewords using Directly Addressable Codes.

than one chunk). We proceed similarly with C_3, and so on. If the maximum integer of the sequence X is M, we need at most $\lceil \frac{\log M}{b} \rceil$ streams C_k.

Each stream C_k will be separated into two parts. The lowest b bits of the chunks will be stored contiguously in an array A_k (of $b \cdot n_k$ bits), whereas the highest bits of the chunks will be concatenated into a bitmap B_k of n_k bits. Figure 4.1 illustrates the rearrangement of the different chunks of the first five codewords of an example sequence. Notice that the highest bit of every chunk determined in the Vbyte encoding if it was the chunk containing the most significant bits of the binary representation, therefore the bits in each B_k identify whether there is a chunk of that codeword in C_{k+1} or not, that is, if the codeword ends at that level (it is the most significant chunk) or it continues at the next level of the representation.

We set up *rank* data structures on the B_k bitmaps, which answer *rank* in constant time using $O(\frac{n_k \log\log N}{\log N})$ extra bits of space, being N the length in bits of the encoded sequence[1]. As we have shown in Section 2.3.1, solutions to *rank* are rather practical (unlike those for *select*, despite their similar theoretical performance).

The overall structure is composed by the concatenation of the B_ks bitmaps, the A_ks arrays, and pointers to the beginning of the stream of each k. These pointers need at most $\lceil \frac{\log M}{b} \rceil \lceil \log N \rceil$ bits overall (remember that there were $\lceil \frac{\log M}{b} \rceil$ streams C_k), and this space is in practice negligible. In total there are $\sum_k n_k = \frac{N}{b+1}$ chunks

[1] This is achieved by using blocks of $\frac{1}{2} \log N$ bits in the *rank* directories [Jac89a, Cla96, Mun96].

C_1	A_1	00	10	10	01	01	01	11
	B_1	1	0	1	0	1	1	1
C_2	A_2	01	10	01	01	00		
	B_2	0	0	1	0	1		
C_3	A_3	01	01					
	B_3	0	0					

Figure 4.2: Example using Directly Addressable Codes.

in the encoding (note N is a multiple of $b+1$), and thus the extra space for the *rank* data structures is just $O(\frac{N \log\log N}{b \log N})$.

Extraction of the i-th value of the sequence is carried out as follows. We start with $i_1 = i$ and get its first chunk $C_{i,1} = B_1[i_1] : A_1[i_1]$. If $B_1[i_1] = 0$ we are done with $x_i = A_1[i_1]$. Otherwise we set $i_2 = rank(B_1, i_1)$, which gives us the correct position of the second chunk of x_i in C_2, and get $C_{i,2} = B_2[i_2] : A_2[i_2]$. If $B_2[i_2] = 0$, we are done with $x_i = A_1[i_1] + A_2[i_2] \cdot 2^b$. Otherwise we set $i_3 = rank(B_2, i_2)$ and so on.

Example Let sequence X be $X = 4, 2, 10, 1, 21, 5, 19$. If we use a Vbyte encoding with $b = 2$, the codewords of the symbols are:

- $x_1 = 4 \rightarrow CW_1 = \underline{0}01\ \underline{1}00$
- $x_2 = 2 \rightarrow CW_2 = \underline{0}10$
- $x_3 = 10 \rightarrow CW_3 = \underline{0}10\ \underline{1}10$
- $x_4 = 1 \rightarrow CW_4 = \underline{0}01$
- $x_5 = 21 \rightarrow CW_5 = \underline{0}01\ \underline{1}01\ \underline{1}01$
- $x_6 = 5 \rightarrow CW_6 = \underline{0}01\ \underline{1}01$
- $x_7 = 19 \rightarrow CW_7 = \underline{0}01\ \underline{1}00\ \underline{1}11$

Then, we create the first stream C_1 with the least significant chunk of each codeword. The rightmost chunk of CW_1 is $\underline{1}00$, the rightmost chunk of CW_2 is $\underline{0}10$ and so on. C_1 is composed of the two arrays A_1 and B_1 previously defined. A_1 consists of the $b = 2$ last bits of those least significant chunks and B_1 is the bit array containing the first bit of each chunk, determining whether the code continues or not (we underlined this bit to make it clearer). Then $A_1 = 00\ 10\ 10\ 01\ 01\ 01\ 11$, $B_1 = 1\ 0\ 1\ 0\ 1\ 1\ 1$ and $n_1 = n = 7$. In the second stream, C_2, there are only $n_2 = 5$ chunks, since the codewords CW_2 and CW_4 corresponding to the second and fourth elements are encoded with just one chunk, so their codewords are completely contained in the stream C_1. Then, C_2 is composed by the second rightmost chunk of all the codewords that have more than one chunk. It is composed by the second least significant chunk of CW_1, that is $\underline{0}01$, the second least significant chunk of CW_3, that is $\underline{0}10$, and so on. We proceed in the same way for the rest of the streams and we obtain the representation in Figure 4.2.

If we want to extract the integer at position 3, that is x_3, then we proceed as follows. We start with $i_1 = 3$ and get its first chunk $C_{3,1} = B_1[3] : A_1[3] = 110$. Since $B_1[3] = 1$ we know that the codeword is not complete and we must obtain the next chunk placed in the second stream C_2. We set $i_2 = rank(B_1, 3) = 2$, so we know that the second chunk is located in the second position of the arrays of C_2. We can retrieve the second chunk $C_{3,2} = B_2[2] : A_2[2] = 010$. Since $B_2[2] = 0$, we are done and $x_3 = A_1[3] + A_2[2] \cdot 2^2 = (10)_2 + (10)_2 \cdot 2^2 = 2 + 2 \cdot 4 = 10$.

Extraction of a random codeword requires $\lceil \frac{N}{n(b+1)} \rceil$ accesses, where N is the length in bits of the encoded sequence, n is the number of integers of the sequence and b the size of the blocks; the worst case is at most $\lceil \frac{\log M}{b} \rceil$ accesses, which is the maximum number of levels of the representation (M is the maximum integer of the sequence X). Thus, in case the numbers to represent come from a statistical variable-length coding, and the sequence is accessed at uniformly distributed positions, we have the additional benefit that shorter codewords are accessed more often and are cheaper to decode.

The extraction of r consecutive codewords can be performed in a more efficient way than just r independent accesses to the encoded sequence. In particular, we can retrieve the complete original sequence of integers without using any *rank* operation, by just sequentially processing the levels of the representation. Note that all the chunks of the codewords are contiguously stored at each level according to the position in the original sequence of the integer they are representing. Retrieving the original sequence of integers consists in sequentially decoding all the codewords of the representation. By using just one pointer at each level of the representation, which indicates the last chunk read at that level, we can process all the levels in a synchronized way to retrieve the whole sequence. At first, the pointers point to the beginning of each level. The first codeword is decoded and the pointers are moved to the next chunk at those levels containing part of the codeword of the first integer. Then, the second codeword can be decoded considering the chunks pointed by the pointers. This procedure is repeated until all the levels are completely processed, that is, until all the codewords have been decoded and the original sequence of integers has been obtained.

More generally, to extract r consecutive codewords starting from a random position i, $\lceil \frac{\log M}{b} \rceil$ rank operations must be performed to initialize the pointers at each level, that is, the pointer at the first level will point to position $p_1 = i$ of the stream

C_1 and the pointer at level k will point to position $p_k = rank(B_{k-1}, p_{k-1})$ of C_k, for each $k > 1$. Finally, a sequential decoding of r codewords can be performed by just reading the next pointed chunk at each level.

4.1.1 Implementation considerations

We now describe some implementation details that differ in some degree from the theoretical proposal explained before. Most of the decisions were made in order to obtain a better usage of the space of the representation.

The first practical consideration concerns the codification scheme. Vbyte codes split the binary representation of an integer into several chunks. Due to the fact that the most significant bit of the binary representation of any integer is different to zero, the highest chunk of the Vbyte codification cannot be all zeroes. Hence, in this way we lose a value in the highest chunk, namely the one that has all the bits in zero, and consequently, the representation is not obtaining the best possible space usage. To avoid this, we use in our implementation the variant of Vbytes designed for text compression called ETDC [BFNP07], a dense encoding scheme that will be explained in Section 7.2.3. This encoding scheme can make use of all the combinations of chunks and obtains better spaces.

Another variation of the final implementation with regard to the presented proposal is that the last bitmap B_L (where L is the number of levels) is not stored in the final representation of the sequence of integers. This is due to the fact that the bits of this bitmap indicate whether a codeword continues in the next level or not, and since B_L corresponds to the last level, no codeword continues in the next level of the representation. Hence, all the bits in B_L are zeroes, so it can be omitted from the final representation, saving some space.

A further practical consideration refers to the implementation of the *rank* data structures over the bitmaps B_k. We have applied different practical solutions from Section 2.3.1, obtaining excellent times when 37.5% extra space is used on top of B_k, and decent ones using up to 5% extra space [GGMN05]. Then, one parameter of the representation is X, that is, the extra space (in bits) per each bit of the bit array (we will set $X = 0.05$ or $X = 0.375$ depending on the chosen practical implementation).

Hence, the total size of the representation, computed as the sum of the size of each level, is $\sum_{k=1}^{L-1} n_k \cdot (b + 1 + X) + n_L \cdot b$, where the parameters b and X can be modified in order to obtain different space-time tradeoffs. The number of levels L and the number of chunks in each level n_k, $k \leq L$, are determined by the value of b and the frequency distribution of the sequence of integers to encode.

4.2 Minimizing the space

In the previous section we explained DACs by representing a sequence of integers with a fixed parameter b, which remains constant for every level of the representation. However, the value of b could be chosen differently at each level of the representation to fit a concrete application. In particular, this can be used to optimize the overall compression. In this section we propose an algorithm to obtain the optimal number of levels of the representation and the optimal b values to achieve the most compact space as possible for a given sequence of integers.

4.2.1 Optimization Problem

As explained, the fact that the DACs separate the chunks of a code in several independent levels can be exploited by using different b values for each level. Among all the possible combinations of b values, those that obtain the minimal space are particularly interesting. It can be vital to represent the list of integers in a very compact way, especially in some applications with strict space restrictions. Notice that we are just focusing on obtaining the most compact representation, without taking into account its time performance. This optimization can generate a not so efficient representation of the sequence in terms of time if it leads to many different levels (which worsens the access time).

We present an algorithm that, given the sequence of integers to encode and their frequency distribution, returns the optimal values for the parameters of DACs (number of levels and b values for each level) that minimize the space of the encoded sequence.

Following the notation used in Figure 4.1 and the implementation considerations in Section 4.1.1, the total size of the representation is $\sum_{k=1}^{L-1} n_k \cdot (b+1+X) + n_L \cdot b$ if the same fixed parameter b is used for all the levels. Notice that there are two modifiable parameters: b is the size of the blocks in the encoding, and X is the extra space used for the *rank* structure over the bitmaps B_k. The number of levels L and the number of chunks in each level n_k, $k \leq L$, are determined by the value of b and the frequency distribution of the sequence of integers to encode.

If the value of b is not the same for every level, we have different b_k values for each level $k \leq L$ and the size of each level is:

$$\text{size of level } k \text{ in bits} = \begin{cases} n_k \cdot (b_k + 1 + X) & \text{if } k < L \\ n_L \cdot b_L & \text{for the last level} \end{cases} \tag{4.1}$$

Hence, the formula for the total size of the representation of the sequence of integers becomes $\sum_{k=1}^{L-1} n_k \cdot (b_k + 1 + X) + n_L \cdot b_L$.

Therefore, the optimization problem, which consists in obtaining the minimal space usage for the representation of a sequence of integers, can be written as follows: $min \left(\sum_{k=1}^{L-1} n_k \cdot (b_k + 1 + X) + n_L \cdot b_L \right)$. The goal of the optimization is to find the values number of levels L and the values $b_k, \forall k \leq L$, that minimize the total size of the encoding, considering the given fixed parameter X.

To completely understand the formula above, we must mention that the number of chunks n_k of each level k is determined not only by the frequency distribution of the integers of the sequence to encode, but also by the values $b_j, j < k$. This can be easily seen with the next example. Level 2 will contain chunks of the codewords assigned to all the integers that cannot be totally represented at level 1. The integers that are totally represented at level 1 will be those that are smaller than 2^{b_1}. Then, if we choose a very large b_1, there will be more integers that are fully represented at level 1 and are not continued at level 2, decreasing the size of level 2. Otherwise, if a small b_1 is chosen, the number of chunks of level 2 will be larger, since level 2 will contain the representation of many integers that are not fully represented at level 1, since they are bigger than 2^{b_1}. Hence, the size of level 2 depends on the value of b_1, and in general, the size of level k will depend on the sum of the values of b_j for each level j prior to level k.

4.2.2 Optimization Algorithm

In this section we present a dynamic programming algorithm that obtains the values for L and $b_k, k \leq L$, that minimize the size of the representation of the given sequence. The optimization problem can be solved using dynamic programming by noticing that the subproblems are of the form "encode in the best way all the values which are larger or equal to 2^x, ignoring their x lowest bits".

Given the sequence of integers to represent, where the maximum value is M and $m = \lfloor \log(M) \rfloor$, it is trivial to compute a vector fc with the cumulative frequencies of all the integers 2^i, with $0 \leq i \leq m$, such that $fc[i]$ would be the number of times that all the integers lower than 2^i appear in the sequence to be encoded. That is, if $f(i)$ is the frequency of integer i, $fc[i] = \sum_{1}^{2^i-1} f(j)$. In addition, we include an extra value for this vector, $fc[m+1]$ with the cumulative frequency for integer M, that is, $fc[m+1] = n$, since the length of the sequence to encode is equal to the number of times that all the integers appear in the sequence. This vector fc of size $m+2$ will be the only input of the algorithm.

Now, we will show the suitability of the problem for the dynamic programming paradigm. We will prove that our problem exhibits an optimal substructure, that is, that the solution can be obtained by the combination of optimal solutions to its subproblems.

For convenience, let us adopt the notation $< A_m A_{m-1} \cdots A_1 A_0 >$ for the binary representation of the maximum value of the sequence of integers (denoted by M), where A_m is the most significant bit and A_0 is the least significant bit. The binary representation of any integer of the sequence will be $< A_r \cdots A_1 A_0 >$, where $r \leq m$, since any integer of the sequence is lower or equal to M.

We define the subproblem t, $0 \leq t \leq m$ as the problem to obtain the optimal value for L_t and $b_{t_k}, k \leq L_t$, for the representation of the $r - t + 1$ most significant bits $< A_r A_{r-1} \cdots A_t >$ of each integer of the sequence greater or equal to 2^t. Hence, the solution to the subproblem t encodes in the best way all the values greater or equal to 2^t, ignoring their t lowest bits. Following the previous definition, the original problem, which consists in the computation of the optimal values to represent the sequence of integers in the minimal space, is solved when we obtain the solution to the subproblem 0, that is, when we have obtained the optimal values to encode in the best way all the values of the sequence (which are greater or equal to 2^0), ignoring their 0 lowest bits (that is, we compute the complete space usage of the representation).

Analogously to the meaning of the values of L and $b_k, k \leq L$, for the original problem, the value L_t associated with the subproblem t represents the optimal number of levels used to represent the $r - t + 1$ bits of every integer of the sequence (with $r \geq t$), and the values b_{t_k} are the sizes of the blocks in each level k. The minimal number of levels L_t to represent those bits is 1, if we create just one level with blocks of size $b_{t_1} = m - t + 1$ (note that $r = m$ for the maximum integer of the sequence, so it is necessary that the size of the blocks in this unique level is $m - t + 1$ so the $m - t + 1$ most significant bits of M can be represented). The maximum value of L_t is $m - t + 1$ if just 1 bit is represented in each level. Hence, the value of L_t can be $1 \leq L_t \leq m - t + 1$. For each subproblem t the following equation holds: $\sum_{k=1}^{L_t} b_{t_k} = m - t + 1$, that is, the values b_{t_k} define a partition of the bits $< A_m A_{m-1} \cdots A_t >$, such that the optimal solution to the subproblem t obtains the optimal partition that encodes all the integers of the sequence greater or equal to 2^t in the minimal space, ignoring their t lowest bits.

We have seen that the solution to the original problem is obtained when we solve the subproblem 0. We describe now how we can obtain this solution from the optimal values of the higher subproblems. We start by computing the optimal solution to the subproblem m, which is trivial, then the optimal solution to the subproblem $m - 1$, using the already computed solution to the subproblem m, then the optimal solution to the subproblem $m - 2$ from the optimal values of the subproblems m and $m-1$ and so on, up to subproblem 0, whose solution is obtained from the optimal values of the subproblem m, subproblem $m - 1$, ..., subproblem 2 and subproblem 1. That is, the solution to a subproblem t is obtained from all the solutions to the subproblems i with $i = t + 1, \ldots, m$. Therefore, we follow a bottom-up approach where we obtain the solution for the trivial subproblem, that is, when just 1 bit is considered and m bits are ignored, then we obtain the optimal values for the intermediate subproblems using the values obtained for the shorter problems, and finally the optimal value for the complete problem is obtained when all the bits are considered.

Let us show now how to obtain the solution to a subproblem t from the solutions to the subproblems i with $i = t+1, \ldots, m$, which we assume that have been already computed. In the subproblem t a new bit A_t is considered, and the algorithm must decide the optimal partition for $< A_m \cdots A_t >$, that is, the number of levels L_t and the size of the blocks for each level $b_{t_k}, k \leq L_t$. This bit A_t belongs to the first level of the optimal representation of the bits involved in the subproblem t. The size of this first level is b_{t_1}, and $1 \leq b_{t_1} \leq m - t + 1$, as we have previously mentioned. Depending on the size of this first new level, which includes the new bit considered A_t, we can create the solution of the subproblem t as follows:

- If $b_{t_1} = m - t + 1$, then $L_t = 1$ since just one level is created for all the bits involved in the subproblem t.

- If $b_{t_1} = m - t$, one level is created for the bits $< A_{m-1} \cdots A_t >$. There is just one bit that is not included in this level, that is, $< A_m >$. Then, the optimal solution to the subproblem m can be used to solve it.

- If $b_{t_1} = m - t - 1$, one level is created for the bits $< A_{m-2} \cdots A_t >$. The rest of the bits that are not included in this level, that is, $< A_m A_{m-1} >$ must be also partition in levels. Then, the optimal solution to the subproblem $m - 1$ can be used to solve it.

- $\cdots$

- If $b_{t_1} = 2$, one level is created for the bits $< A_{t+1} A_t >$. The rest of the bits $< A_m A_{m-1} \cdots A_{t+2} >$ must be also partition in levels so the optimal solution to the subproblem $t + 2$ can be used.

- If $b_{t_1} = 1$, one level is created for just the bit $< A_t >$, and the solution to the subproblem $t + 1$ is used for the rest of the bits $< A_m A_{m-1} \cdots A_{t+1} >$.

The optimal solution to the subproblem t will be obtained from the comparison of these $m-t+1$ possible solutions, choosing the one that minimizes the space. The procedure of the comparison in order to obtain the best alternative will be explained in more detail later on. Thus, the optimal solution of the subproblem t is obtained from the optimal solution of one of the previous subproblems i, $i = t + 1, \ldots, m$.

We prove now that our problem exhibits an optimal substructure that enables the usage of the dynamic programming paradigm.

Lemma 4.1 *The optimization problem proposed in this section presents an optimal substructure.*

Proof The optimal subdivision in levels for the whole sequence, that is, for the subproblem 0, which takes into account all the bits $< A_m A_{m-1} \cdots A_0 >$, must contain an optimal subdivision of the bits $< A_m A_{m-1} \cdots A_{b_1} >$, where b_1 is the size of the blocks for the first level of the representation. If there was a less costly way to represent all the bits $< A_m A_{m-1} \cdots A_{b_1} >$ of the integers of the sequence (that is, a better solution to the subproblem b_1), substituting that new subdivision in the optimal solution for the representation of the bits $< A_m A_{m-1} \cdots A_0 >$ would produce another representation of the sequence of integers whose cost would be lower than the optimum: a contradiction. Thus, an optimal solution to the optimization problem for a certain sequence of integers contains within it optimal solutions to smaller subproblems.

Another key aspect that enables the applicability of dynamic programming, apart from the optimal substructure of the problem that was proven before, is that the problem has relatively few subproblems: one problem for each bit of the binary representation of the maximum value M, that is, $\lfloor \log M \rfloor + 1$ subproblems.

Example Figure 4.3 illustrates the optimal substructure of our optimization problem with a small example. In this case, the maximum value of the sequence is encoded with 9 bits in its binary representation $< A_8 A_7 \ldots A_1 A_0 >$. We label the most significant bit with A_8 and the least significant bit with A_0 and we will cover all the bits from most to least significant, so we start with the trivial subproblem when $t = m = 8$ (just 1 bit is considered, and the 8 least significant bits are ignored) and the optimal value of the problem is obtained for $t = 0$ (all the bits are considered, 0 are ignored).

Let us consider that the algorithm has already obtained the optimal solution to the subproblems from 8 to 4. We explain now the computation of the optimal solution to the subproblem 3, that is, to encode in the best way all the values which are larger or equal to 2^3, ignoring their 3 lowest bits. We make use of the previous optimal solutions such that there are $m - t + 1 = 8 - 3 + 1 = 6$ possible alternatives to consider and the optimal value to this subproblem is obtained by comparing the size obtained by each one of these different alternatives. Then, to encode all the values greater or equal than $2^3 = 8$ (we ignore the 3 least significant bits, which can encode values from 1 to 7), we must compare the following alternatives:

i) creating a new level with the 6 most significant bits,

ii) maintaining the optimal solution for the 1 most significant bit and creating a new level for the other 5 bits,

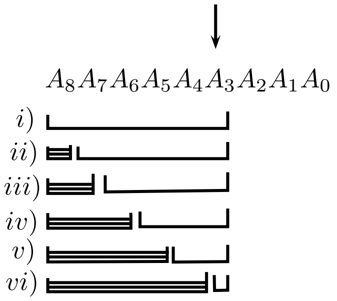


Figure 4.3: Optimal substructure in a small example.

iii) maintaining the optimal solution previously computed for the 2 most significant bits and creating a new level with the other 4 bits,

iv) maintaining the optimal solution previously computed for the 3 most significant bits and creating a new level with the other 3 bits,

v) maintaining the optimal solution previously computed for the 4 most significant bits and creating a new level with the other 2 bits, and

vi) maintaining the optimal division for the first 5 bits and creating a new level for the sixth bit.

Once we have shown that the optimal value of a subproblem t is obtained from the solutions to the subproblems $t{+}1, \ldots, m$, we must define the value of this optimal solution to the subproblem t in terms of the optimal solutions to the subproblems. Let $s[i]$ be the minimum size of the representation obtained for the subproblem i, $t < i \leq m$. The optimal space to represent the whole sequence of integers would be $s[0]$.

We can define $s[t]$ recursively as follows. For the subproblem t, $m{-}t{+}1$ solutions are computed and compared: $m - t$ of them are obtained from the subproblems $i = t + 1, \ldots, m$ and there is one more solution obtained trivially:

- Each of the $m - t$ solutions of the subproblem t obtained by means of one subproblem $i, i = t + 1, \ldots, m$ consists in maintaining the levels and the sizes of the levels created for the subproblem i and adding a new level for the bits $< A_{i+1} \cdots A_t >$, whose size of blocks is $i - t$. The size in bits for this new level will be $(fc[m + 1] - fc[t]) \cdot (i - t + 1 + X)$, following the formula (4.1), where the number of chunks of the level created n_{t_1} is $fc[m + 1] - fc[t]$, that is, there is one chunk per each integer of the sequence that is greater or equal than 2^t, the size of the blocks for this level is $b_{t_1} = i - t$ and $L > k = 1$.

- The trivial subdivision for the subproblem t, which is not solved by means of any of the previous subproblems, consists in creating one unique level with all the bits $< A_m \cdots A_t >$. Consequently, since this level includes the most significant bit A_m, it is the last level of the representation. Thus, the bit array B_k is not stored, and the size in bits of the level is $(fc[m+1]-fc[t])\cdot(m-t+1)$, following the formula (4.1) with $n_{t_1} = fc[m+1] - fc[t]$, $b_{t_1} = m - t + 1$ and $k = L = 1$.

Following the above, the recursive definition for the minimum cost of the representation of the sequence of integers for the subproblem t, that is, to encode in the best way all the values which are greater or equal to 2^t, ignoring their t lowest bits, becomes:

$$\begin{aligned} s[t] = \min\{ &\min_{t<i\leq m} \{s[i] + (fc[m+1] - fc[t]) \cdot (i - t + 1 + X)\}, \\ &(fc[m+1] - fc[t]) \cdot (m - t + 1)\} \end{aligned} \tag{4.2}$$

If the minimal value is obtained from the upper part of the expression (4.2), then the solution of the problem is obtained from the solution of a subproblem i, the one that minimizes the value of the expression. Hence, the number of levels L_t will be $L_t = L_i + 1$, since one extra level is created. The size of the new level (the first level of the optimal solution to the subproblem t) is $b_{t_1} = i - t$ and the sizes of the rest of the levels are identical to the sizes of the levels of the optimal solution to the subproblem i, that is, $b_{t_{k+1}} = b_{i_k}, k < L_t$. On the other hand, if the minimal value is obtained from the bottom part of the expression (4.2), then only one level is created, that is, $L_t = 1$. The size of this level is $b_{t_1} = m - t + 1$ bits.

For the computation of the values of L_t and $b_{t_k}, \forall k \leq L_t$ for each subproblem t using the dynamic programming algorithm described above, we use three vectors of size $m + 1$. The position t of each vector contains the following information:

- $s[t]$: contains a long value representing the optimal size of the encoding of all the values greater or equal to 2^t when the t lowest bits are ignored.
- $l[t]$: stores the optimal number of levels L_t for the subproblem t, which considers only the $m - t + 1$ highest bits.
- $b[t]$: stores the size of the blocks for the first level of the optimal subdivision in blocks of the subproblem t. Once the optimal subdivisions for all the subproblems have been created, the optimal values b_k can be obtained from the values of this vector.

The $s[t]$ values give the costs in bits of the optimal solutions to the subproblems. The other two vectors, $l[t]$ and $b[t]$ help us keep track of how to construct the optimal solution.

Algorithm 4.1: **Optimize**(m, fc)

```
for $t = m \dots 0$ do
    $minSize \leftarrow maxLongValue$
    $minPos \leftarrow m$
    for $i = m \dots t+1$ do
        $currentSize \leftarrow s[i] + (fc[m+1] - fc[t]) \times (i - t + 1 + X)$
        if $minSize > currentSize$ then
            $minSize \leftarrow currentSize$
            $minPos \leftarrow i$
        end
    end
    if $minSize < (fc[m+1] - fc[t]) \times (m - t + 1)$ then
        $s[t] \leftarrow minSize$
        $l[t] \leftarrow l[minPos] + 1$
        $b[t] \leftarrow minPos - t$
    else
        $s[t] \leftarrow (fc[m+1] - fc[t]) \times (m - t + 1)$
        $l[t] \leftarrow 1$
        $b[t] \leftarrow (m - t + 1)$
    end
end
$L \leftarrow l[0]$
$t \leftarrow 0$
for $k = 1 \dots l[0]$ do
    $b_k \leftarrow b[t]$
    $t \leftarrow t + b[t]$
end
return $L, b_k$
```

Algorithm 4.1 obtains the optimal number of levels L and the b_k values, $k \leq L$, given a vector of cumulative frequencies of size $m+2$. The optimal size for each subproblem is stored in vector $s[t]$, the optimal number of levels in $l[t]$ and the parameter for the first level of those optimal subdivisions in $b[t]$. Since we cover all the bits from the most significant bit (A_m) to the least significant bit (A_0), the values stored in $l[0]$ and $b[0]$ are the optimal values for the whole encoding.

Analysis Let us denote M the maximum value of the sequence and $m+1$ the number of bits needed for the binary representation of M.

We have few subproblems: one problem for each bit of the binary representation of the maximum value of the sequence. Each subproblem computes its solution by accessing to the values stored in an auxiliary table of size $m+1$, which contains the solutions to all the previous subproblems.

Hence, as we can see in Algorithm 4.1, the algorithm is quadratic in the number of bits of the binary representation of the largest integer of the sequence. Since $m = \lfloor \log M \rfloor$, the optimization would cost just $O(\log^2 M)$ time, provided that vector *fc* has been previously computed.

The space consumption is $O(\log M)$, since the input *fc* has size $m + 2$ and the three vectors used during the optimization algorithm, that is, $s[t]$, $b[t]$ and $l[t]$ have size $m + 1$.

4.2.2.1 Limiting the number of levels

As we have said, the optimization algorithm presented above obtains the optimal number of levels and b values that minimize the space of the representation given a sequence of integers. However, the optimal number of levels can be high, degrading the time efficiency. Hence, it would be interesting to obtain the configuration of parameters that minimizes the space usage while limiting the number of levels of the representation. If we restrict the number of levels of the representation, we are limiting the access time for the worst case (the access time for the maximum value of the sequence).

Thus, the new problem consists in, given the sequence of integers to encode, their frequency distribution and an integer R, returning the optimal values for the parameters of DACs (number of levels and b value for each level) that minimize the space of the encoded sequence such that the number of levels is lower or equal to R.

This can be trivially computed by modifying the optimization algorithm described in Algorithm 4.1, including a new parameter v that restricts the number of levels to use. For each subproblem t, its optimal solution is computed as follows:

1. If $v = 1$, then just one level is created to represent all the integers involved in that subproblem.

2. If $v > 1$, we compute the solution to subproblem t as explained before using the optimal solutions for the subproblems i with $i = t + 1, \ldots, m$, which were computed with the parameter $v - 1$.

Hence, the optimal configuration restricting the number of levels to R is obtained when computing the optimal solution to subproblem $t = 0$ and $v = R$. The trivial case when $R = 1$ consists in using one level with blocks of $m + 1$ bits, that is, we use a fixed-length encoding where each integer is represented with the number of bits required by the binary representation of the maximum integer of the sequence. Being L the optimal number of levels (without restriction), we obtain the same configuration than Algorithm 4.1 for $R \geq L$. Varying the value of R, $1 \leq R \leq L$, we can obtain a space/time tradeoff where space improves as more levels can be created, and access times become faster when limiting the number of levels to a lower value.

Analysis Being R at most $m+1$, for the extreme case of building a representation with $m+1$ levels, one level for each bit of the binary representation of each integer, thus $R = O(\log M)$. Hence, the time complexity of the optimization limiting the number of levels would be $O(\log^3 M)$, as we must compute at most each one of the previous subproblem t with the values of parameter v, for $v \leq R$.

In addition, we can restrict the possible values of b_k at each level of the representation (except for the last level) to the values 1, 2, 4, and 8, that is, restricting b_k to be a power of two lower or equal to 8. This byte-aligned variation of the algorithm generates a representation of the sequence where each chunk is completely contained in a unique byte, and decompression and accesses can be implemented more efficiently in practice.

In the next chapter we show the experimental evaluation of our technique applied in different domains and we show how the optimal configuration of b values restricting the number of levels of the representation leads to an interesting space/time tradeoff.

Chapter 5

Applications and experiments

The new technique presented in the previous chapter, the Directly Addressable Codes (DACs), is practical and can be successfully used in numerous applications where direct access is required over the representation of a sequence of integers. This requirement is frequent in compressed data structures, such as suffix trees, arrays, and inverted indexes, to name just a few. We show experimentally that the technique offers a competitive alternative to other encoding schemes that require extra structures to support direct access.

More generally, we can consider two different scenarios:

- **Coding sequences of integers**

  There are several domains where integers are the symbols to encode. These integers have values of different magnitudes, usually small, but some of them have large values. We want to represent these integers in a compact way while supporting direct access to any value of the sequence.

  As we have seen in Section 3.1, there are several encoding schemes especially designed for integers, such as δ-codes, γ-codes, Rice codes or Vbyte codes. If direct access must be provided, we must include a sparse sampling over the encoded sequence. We can compare the performance of DACs in this scenario against the performance of the sparse sampling over the encoded sequences using these encoding schemes for integers.

- **Representing sequences of arbitrary symbols as sequences of integers via a statistical modeler**

  There are also multiple domains where we want to obtain a compressed representation of a sequence of arbitrary symbols. A statistical modeler can be used to obtain a model of the sequence where the frequencies of the symbol are obtained and used to assign them variable-length codewords: shorter codewords are given to more frequent symbols and longer codewords are assigned to those symbols that appear fewer times in the sequence. This strategy produces a compact representation of the sequence. In addition to the encoded sequence, a vocabulary of symbols sorted by frequency and some information about the encoding scheme must be also stored, such that the original sequence can be retrieved. Huffman encoding is one of the most used techniques when compressing sequences of symbols, since it obtains a very compact space. However, direct access is not possible over the compressed sequence obtained by Huffman encoding, so a sparse sampling is used when this functionality is required.

DACs can also be used in this scenario, by considering the sequence of positions of the symbols in the sorted vocabulary instead of the sequence of symbols. This represents the original sequence in a compact way and supports fast access to any element of the encoded sequence and, consequently, to any symbol of the original sequence, since the vocabulary of sorted symbols is also stored.

This chapter is organized as follows. In Section 5.1 we start by analyzing the space/time trade-off due to the value of parameter b. Section 5.2 shows some immediate applications of our scheme, and compares the behavior of DACs with other solutions that also support direct access. Finally, Section 5.3 describes some other scenarios where DACs have been successfully applied.

5.1 Influence of the parameter b

We first analyze the influence of the chosen value for parameter b on the space/time efficiency of our proposal. We implemented our technique with b values manually chosen for each level (in many cases the same b for all) and also with the optimal values obtained with the optimization algorithm described in Section 4.2, including the variations where we limit the number of levels. We implemented *rank* operations using the 5%-extra space data structure by González *et al.* [GGMN05] (this is space over the B_k bitmaps).

The machine used in these experiments is an isolated Intel®Pentium®-IV 3.00 GHz system (16Kb L1 + 1024Kb L2 cache), with 4 GB dual-channel DDR-400Mhz RAM. It ran Debian GNU/Linux (kernel version 2.4.27). The compiler used was gcc version 3.3.5 and `-O9` compiler optimizations were set.

To study the behavior of our representation, DACs are applied over a large sequence of integers. In this section, the sequence of integers represents a natural language text, regarding the text as a sequence of words. Therefore, the integer at position i of the sequence to encode represents the word at position i of the text.

We first extract the σ different words of the text and create a vocabulary of terms. We sort this vocabulary by frequency, and then assign an integer code to each word according to the position of the word in this sorted vocabulary: we assign the integer 1 to the most frequent symbol, the integer 2 to the next most frequent symbol and so on, assigning the integer σ to the least frequent symbol of the vocabulary. Hence, the text can be regarded as a sequence of integers if we replace each word of the text by the integer associated with it.

If we apply DACs over this sequence, we obtain a representation of the text where decoding from any position, forward and also backwards, is supported. In addition, since the integers associated with the words are assigned according to a statistical modeling, that is, we assign smaller integers to those words that appear more frequently and larger integers for words appear less frequently, we obtain a compact representation of the text.

We took the TREC-4[1] collection CR (Congressional Record 1993), of about 47 MB, composed of 10,113,143 words from a vocabulary of 117,713 different words. We represented this text using DACs with different b values for the size of the blocks for each level of the representation. More specifically, we analyzed the behavior of the seven following configurations of the technique:

- We call *opt* the approach with the b values for each level of the representation that minimize the space. These b values are computed using the optimization algorithm presented in Section 4.2. For this collection CR, the optimal number of levels (without restriction) is 7 and values of b are - starting from the first level of the representation to the last level - 4, 4, 2, 2, 2, 1, 2. We also compute the optimal configuration limiting the number of levels R, varying $1 \leq R < 7$.
- We call *opt-aligned* the approach using the variation of the optimization algorithm that restricts to R the number of levels of the representation and uses b values that are power of two. We compute the optimal configuration when the number of levels is restricted to R with $1 \leq R \leq 7$.
- $'8'$, $'7'$, $'4'$, $'2'$ stand for the alternatives where we maintain a fixed b value for all the levels of the representation, with $b = 8$, $b = 7$, $b = 4$ and $b = 2$ respectively.

Space usage

Table 5.1 shows the compression ratio obtained by each alternative. The first column contains the name of the alternative. The second and third columns shows the

[1] One goal of the Text REtrieval Conference (TREC) consists in providing the infrastructure necessary for large-scale evaluation of text retrieval methodologies. The TREC test collections are available to the retrieval research community. More information in http://trec.nist.gov/

Alternative	Number of levels	b values	Compression ratio
opt (no restrict.)	7	4, 4, 2, 2, 2, 1, 2	**28.88%**
opt $R = 6$	6	4, 4, 2, 2, 2, 3	28.90%
opt $R = 5$	5	5, 4, 3, 2, 3	29.14%
opt $R = 4$	4	6, 4, 3, 4	29.51%
opt $R = 3$	3	6, 5, 6	30.45%
opt $R = 2$	2	8, 9	33.29%
opt $R = 1$	1	17	43.91%
opt-aligned $R = 7$	7	4, 4, 2, 2, 2, 1, 2	**28.88%**
opt-aligned $R = 6$	6	4, 4, 2, 2, 2, 3	28.90%
opt-aligned $R = 5$	5	4, 4, 4, 2, 3	29.25%
opt-aligned $R = 4$	4	4, 4, 4, 5	29.60%
opt-aligned $R = 3$	3	8, 4, 5	30.80%
opt-aligned $R = 2$	2	8, 9	33.29%
opt-aligned $R = 1$	1	17	43.91%
$'8'$	3	8, 8, 8	33.45%
$'7'$	3	7, 7, 7	32.02%
$'4'$	5	4, 4, 4, 4, 4	29.67%
$'2'$	9	2, 2, 2, 2, 2, 2, 2, 2, 2	30.56%

Table 5.1: Compression ratio obtained using different configurations for our DACs.

number of levels used by the alternative and the values of b for those levels, respectively. The last column shows the compression ratio obtained, which is measured as the total space needed to represent the natural language text compared to the total size of the uncompressed text. Not only the sequence of integers is stored, but also the vocabulary of the different words that appear in the text (in uncompressed form), such that the original text can be recovered from the compacted representation.

As it was expected, the *opt* alternative without restricting the number of levels obtains the minimal space. The most space-consuming approach is obtained by the optimization algorithm if we restrict the number of levels to $R = 1$. In this case, since there are 117,713 different words in the collection CR, each word is represented in a unique level with blocks of 17 bits. The next most space-consuming approach is the $'8'$ alternative, which uses $b = 8$ for all the levels. Its worse compression ratio is due to the fact that this alternative assigns very long codewords (9 bits codewords) to highly repetitive words, so the size of the compressed text obtained is far from optimal. Moreover, this $'8'$ version obtains a compact representation with 3 levels, using 8 bits in the last one. However, there is no need of such a large b for the last level. The larger Vbyte codeword that would be assigned to the least frequent word of the vocabulary would have 17 bits. If we use $b = 8$ for all the levels, only one bit of the blocks in the last level is significant, the other 7 bits are all zeroes, wasting that space. In fact, if we used a third level with $b = 1$ the compression ratio would improve close to 0.10%, obtaining a compression of 33.35%.

Alternative $'7'$ compresses to 32.02%, close to the ETDC ratio (31.94% for this collection, as we will see in Section 9.3). Note that we use this variant of the Vbytes codification in order to obtain better space results (so we can exploit chunks with all zeroes as the highest chunks as explained in Section 4.1.1). Hence, the compressed text obtained by DACs using chunks of 8 bits (since $b = 7$) is a relocation of the bytes of the codewords of the compressed text obtained by the ETDC: the bytes are rearranged in several levels and the most significant bit of each byte is placed separately in the bit array of the corresponding level (indicating whether the code continues or not in the following level). The space usage is slightly higher for DACs than for ETDC since extra structures to support *rank* operations in efficient time are stored. However, the compact representation of the text using DACs permits an interesting additional functionality compared to the compressed text obtained by ETDC: direct access to the i-th word of the text.

Both alternatives $'8'$ and $'7'$ are outperformed in terms of space by all the configurations obtained by *opt* and *opt-aligned* approaches for $R > 1$. Even when using only 2 levels, the optimization algorithm is able to obtain a more compressed representation of the sequence. These two approaches, *opt* and *opt-aligned*, obtain the same configuration for $R = 1, 2, 6, 7$, since the b values of the optimal configuration are power of two. For $R = 3, 4, 5$, *opt* approach obtains better compression ratios than *opt-aligned*. As expected, as we restrict the number of levels, the compression ratio is degraded.

The compression obtained by alternative $'2'$ shows that maintaining a small b value for all the levels is not a good choice either. With this approach, there are very few words whose codeword ends in the first level of the representation and most of them need several levels to completely represent the codeword. Then, the bit arrays of the first levels contain mainly 1s, indicating that most of the codewords continue. These bit arrays and the extra structures for the *rank* operation consume a substantial amount of space that can be reduced by noticing that it is not practical to make a level subdivision where there are few codewords ending at the first level. Hence, it may be a preferred choice to create fewer levels, such as with alternative $'4'$, which obtains a better adjustment and improves the compression ratio.

Time efficiency

As we have already stated, the optimization algorithm presented in Section 4.2 obtains the values of b that minimize the space usage of the representation, given a sequence of integers. However, this optimization can lead to an inefficient representation in terms of time, if the number of levels generated is high. Then we can use the variation of the algorithm explained in Section 4.2.2.1 where we limit the number of levels to decrease the worst-case time.

Figure 5.1 (top) shows the average time (in seconds) needed to decompress the whole text. For each alternative we draw a point where the x-coordinate represents the compression ratio obtained by that alternative and the y-coordinate represents the decompression time. Note that decompression is performed without using rank operations as detailed in Section 4.1.

The faster alternatives, $'8'$, *opt* with $R = 1, 2$ and *opt-aligned* with $R = 1, 2, 3$, share a common property: a large value of b is used for the first level of the representation, such that the decompression of most of the codewords of the compressed text ends in that first level. This avoids the cost of jumping to the second level of the representation, which is not located contiguously in main memory, in order to find the continuation of the codeword in the next level, which slows down the decompression speed. Alternative $'4'$ gives a good compromise between time and space, while the *opt* alternative with $R = 7$ (in red color), with the best compression ratio, obtains a worse decompression time, since its representation is composed of seven levels, whereas the representation produced by $'4'$ alternative consists only of five levels. For this sequence, the alternative *opt-aligned* with $R = 7$ obtains the same compression ratio than *opt* with $R = 7$, and since the implementation takes advantage of the byte-alignments, the decompression time for this alternative is significantly better in practice. Alternative $'2'$ is the worst configurations in both time and space. The problem of using low b values in all the levels of the representation is not only of space, as previously explained for the alternative $'2'$, but most importantly in time, since a high number of levels is generated.

Figure 5.1 (bottom) shows the average time to extract a codeword at a random position of the compressed text. We measured the average time to access to (and decode) all the positions of the text in random order. Results are very similar to those obtained for the decompression time and shown in the upper figure. However, the time differences between all the alternatives have been enlarged due to the cost of computing rank operations, which were not necessary when decompressing the whole sequence.

The best times are obtained for *opt* and *opt-aligned* with $R = 1$, since no rank operations are required (they consist of just one level). Then, the alternatives using few levels are the ones that obtain the best times. The byte-alignment can also significantly improve access times. For instance, alternative $'8'$ obtains a very competitive performance since at most 3 rank operations must be performed to extract one codeword of the encoded sequence and it works with complete bytes at each level of the representation[2]. The worst times are obtained when the number of levels of the representation is high (alternative $'2'$ may require 9 rank operations to extract some of the codewords). The optimal alternative in terms of space (alternative *opt*) obtains an intermediate access time. In this case, the *opt-aligned* configuration does not always obtain better access times than the *opt* configuration using the same

[2] We omit in both figures the alternative $'7'$, as its results are worse than alternative $'8'$, as it does not take advantage of byte-alignments.

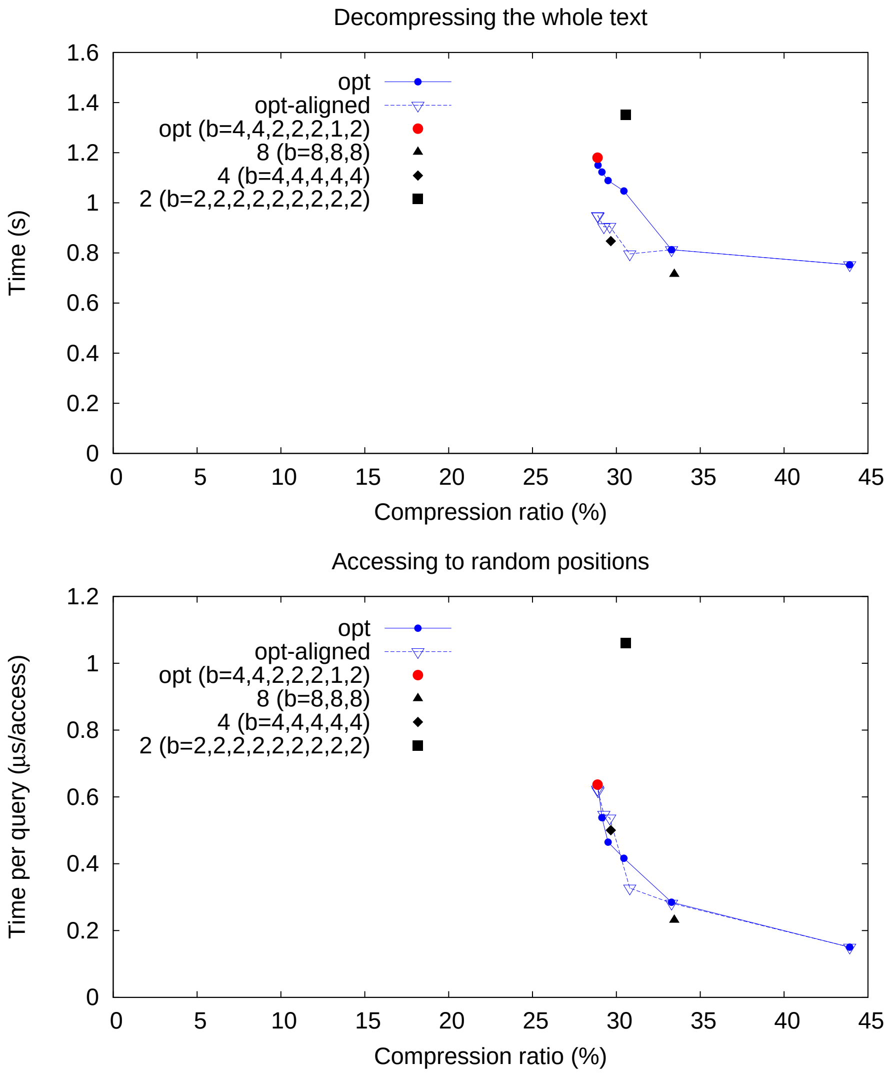


Figure 5.1: Space/time trade-off for different configurations when decompressing the whole text (top), and when accessing and decompressing random positions of the text (bottom).

number of levels. For instance, when $R = 3$, the *opt-aligned* configuration obtains worse compression ratio but better access times, since it uses a large b value for the first level than the *opt* configuration with $R = 3$ and it takes advantage of byte-alignments. However, when $R = 4$ the first level of the *opt-aligned* configuration uses blocks of 4 bits, while the first level of the *opt* configuration with $R = 4$ uses blocks of 6 bits. Hence, even when it is faster to extract a block from the *opt-aligned* representation, there are several integers that are completely represented with *opt* but require an access to the second level of the representation and a rank operation when using the *opt-aligned* approach. Hence, the *opt-aligned* approach does not always lead to a better average time performance.

As we can see in the previous figures, we obtain the best compression ratio when we use the configuration that the optimization algorithm calculates. However, the representation obtained from those values does not obtain the best results in terms of time, since there are other configurations of b values that obtain better decompression speed and access time to any position of the text at the expense of worsening the compression ratio. The speed-up is achieved when the number of levels of the representation is reduced or due to byte-alignments.

5.2 Applications

We will now compare the performance of our DACs with other alternatives that represent the sequence of integers and enable direct access over it.

We consider the two different scenarios that were previously described. In Section 5.2.1 we compare DACs with sparse sampling over δ-codes, γ-codes, Rice codes and Vbyte codes when direct access is provided to a encoded sequence of integers. We use LCP arrays as an example of sequences of integers to encode.

Section 5.2.2 and Section 5.2.3 describe scenarios where we have sequences of arbitrary symbols instead of sequences of integers. We compare the behavior of DACs in this scenario with other statistical encodings such as bit-oriented and byte-oriented Huffman encondings, which require a sparse sampling to provide direct access over the sequence. We also compare our technique with the dense sampling of Ferragina and Venturini, explained in Section 3.2.2, and with a Huffman-shaped wavelet tree (see Section 2.3.2), which compactly represents a sequence of symbols from an arbitrary alphabet and supports efficient access to any element of the sequence. Section 5.2.2 studies the behavior of all these alternatives to represent tuples of k characters of some texts so that they can be compressed to high-order empirical entropy and Section 5.2.3 compares the performance of the byte-oriented version of our technique with the byte-oriented Huffman to represent a natural language text considering words as source symbols.

Table 5.2: Description of the LCP arrays used.

dat	num. elem.	max value	avg value	most freq. value
`dblp`	104,857,601	1,084	28.22	10 (2.15%)
`dna`	104,857,601	17,772	16.75	13 (24.59%)
`proteins`	104,857,601	35,246	195.32	6 (28.75%)

For all the experiments in this section the machine used is a AMD Phenom(tm) II X4 955 Processor (4 cores) with 8 GB RAM. It ran Ubuntu GNU/Linux with kernel version 2.6.31-22-server (64 bits). We compiled with gcc version 4.4.1 and the option `-O9`.

5.2.1 LCP array representation

Consider a text $T[1,n]$ of length n, and all the suffixes of the text, that is, $T[i,n]$ with $1 \leq i \leq n$. Assume that we have all those suffixes lexicographically sorted. The *Longest Common Prefix Array* (LCP array) is an array that stores, for each suffix, how many symbols has in common with the previous suffix, that is, the length of the longest common prefix between each suffix and its predecessor. Most *LCP* values are small, but some can be much larger. Hence, a variable-length encoding scheme is a good solution to represent this sequence of integers.

Our experiments were performed on 100 MB of the XML, DNA and protein texts from Pizza&Chili corpus (http://pizzachili.dcc.uchile.cl). We denote `dblp` the LCP array obtained from the XML file, which contains bibliographic information on major computer science journals and proceedings. We denote `dna` the LCP array obtained from the DNA text, which contains gene DNA sequences consisting of uppercase letters A,G,C,T, and some other few occurrences of special characters. We denote `proteins` the LCP array obtained from the protein text, which contains protein sequences where each of the 20 amino acids is coded as one uppercase letter. Some interesting information about this dataset is shown in Table 5.2. The first column indicates the number of element of the LCP array. The second and third column show, respectively, the maximum and average integer values stored in the LCP array. The last column shows the most frequent integer value and its frequency. For instance, we can observe that the most frequent value of the LCP array in the XML file is 10, but its frequency is 2.15%, since the distribution is more uniform than the values in the LCP array of DNA or protein texts. This information will be useful to understand the behavior and different parameters of each encoding scheme used to represent these LCP arrays.

The LCP arrays were computed and represented using DACs with different configurations for parameter b:

- "DACs b=2" stands for the alternative that uses a fixed value of $b = 2$ for all the levels of the representation.
- "DACs b=4" stands for the alternative that uses a fixed value of $b = 4$ for all the levels of the representation.
- "DACs b=8" stands for the alternative that uses a fixed value of $b = 8$ for all the levels of the representation.
- "DACs opt" stands for the alternative that uses the optimal value for b at each level of the representation. These values are $b = 6, 1, 1, 1, 2$ for `dblp`, $b = 4, 1, 1, 1, 2, 2, 2, 2$ for `dna` and $b = 3, 3, 2, 2, 2, 1, 1, 2$ for `proteins` when no restriction of number of levels is applied. These values were obtained using the optimization algorithm explained in Section 4.2, which considers the frequency distribution of the values to be encoded to compute the optimal values for the number of levels and block sizes that minimize the space required by the DACs. For instance, as we can see in Table 5.2, the most frequent value for `proteins` is 6, and it appears the 28.75% of the times, hence, if we use just 3 bits for the first level, we can compactly represent all those occurrences of integer 6 without wasting any extra bit. In fact, the next most frequent values are 5 and 7, with frequencies 22.98% and 7.55%, which can also be compactly represented in that first level with 3 bits. The same considerations can be made for `dna`, whose most frequent value is 13 and it can be completely represented in a first level of 4 bits. On the other hand, values of `dblp` are more uniformly distributed, and the most frequent values are not that small. Hence, 6 bits for the first level is the optimal fit with its distribution. In addition, we built several configurations limiting the number of levels R, using values of $1 \leq R \leq 5$ for `dblp`, $1 \leq R \leq 8$ for `dna` and $1 \leq R \leq 8$ for `proteins`.
- "DACs opt-aligned" stands for the alternative using the variation of the optimization algorithm that limits the number of levels and uses b values that are power of two. Several representations are built using values of $1 \leq R \leq 6$ for `dblp`, $1 \leq R \leq 8$ for `dna` and $1 \leq R \leq 7$ for `proteins`.

We implemented *rank* operations using the 5%-extra space data structure by González *et al.* [GGMN05] (this is space over the B_k bitmaps).

We compare the space and time efficiency of our proposal with some integer encodings[3], more concretely:

- δ-codes.

[3] I would like to thank Eduardo Rodríguez for providing efficient implementations of δ-codes, γ-codes and Rice codes.

- γ-codes.
- Byte codes, that is, Vbyte codes with $b = 7$ (using bytes as chunks).
- Rice codes, using the value of parameter b that minimizes the space of the encoded sequence. This value is $b = 5$ for `dblp`, $b = 4$ for `dna` and $b = 7$ for `proteins`. These values depend on the average value for each sequence, detailed in Table 5.2. For instance, the average value for sequence `dna` is smaller than for the rest of the sequences, hence we also use a lower b value as parameter for the Rice codes.

To support direct access over the compressed representation of the LCP array we attach a sparse sampling to the encoded sequence obtained by all these integer encoding schemes, so we can compare them with the representation obtained by DACs, which support direct access without any extra structure.

We also compare our structure with the representation of the sequence of integers using the Elias−Fano representation of monotone lists as explained in Section 3.2.3. We use the implementation from the Sux4J project[4] [Vig08], compiling with java version 1.6.0_18.

We measure the space required by each technique in bits per element (bits/e), that is, we show the average number of bits required to encode each value of the LCP array. We also measure decompression and access time in seconds. Decompression time measures the seconds needed to retrieve the original LCP array in plain form. Access time is measured in microseconds per access as the average time to retrieve the elements at random positions of the LCP array.

Table 5.3 shows the space required by δ-codes, γ-codes, byte codes and Rice codes (without any sampling) to represent the three different LCP arrays, and the space occupied by the different configurations of DACs. This is the space required by each alternative to decompress the compressed representation of each LCP array and retrieve the original one in plain form. Note that we do not require the use of the samples to decompress the whole array. We also include the decompression time in seconds.

We can observe that "DACs opt" (without restriction on the number of levels used) obtains the best space among all the alternatives, except for `dblp` using Rice codes, which is also the fastest bit-oriented alternative for this LCP array. Byte codes obtain the fastest decompression times among all the alternatives, including our byte-oriented DACs, that is, "DACs b=8", since the sequential decoding procedure of the byte codes is faster than decompressing using DACs, which requires

[4] http://sux.dsi.unimi.it/

Text	`dblp`		`dna`		`proteins`	
Method	Space (bits/e)	Time (sec.)	Space (bits/e)	Time (sec.)	Space (bits/e)	Time (sec.)
δ- codes	9.5421	1.04	8.3908	1.04	7.8635	1.31
γ- codes	10.0834	1.19	7.7517	1.15	8.2899	1.40
byte codes	8.4024	**0.44**	8.0612	**0.43**	9.2683	**0.51**
Rice codes	**6.9194**	0.91	6.0493	0.89	9.5556	0.93
DACs $b = 2$	8.6992	1.44	6.5008	1.15	7.9499	1.61
DACs $b = 4$	8.9410	0.99	6.0474	0.81	7.1516	0.97
DACs $b = 8$	9.0515	0.54	9.0900	0.50	9.8896	0.58
DACs opt (no restrict.)	7.5222	1.41	**5.5434**	1.35	**6.5797**	2.01

Table 5.3: Space for encoding three different LCP arrays and decompression time under different schemes.

reading bytes at different levels of the representation that are not contiguously located in memory. However, these byte-oriented representations occupy much more space than the bit-oriented encoding schemes. Notice that byte codes cannot occupy less than 8 bits per element and “DACs b=8” cannot occupy less than 9 bits. “DACs b=4” offers an attractive space/time compromise for `dna` and `proteins`, obtaining better spaces than any integer encoding over these LCP arrays and also better times than those bit-oriented encodings (except for Rice codes over `proteins`, which obtain slightly better decompression times but significantly worse space). Its times are close twice the times obtained by byte-code encodings, but the space required is significantly lower.

The main goal of our proposal is to provided fast direct access to the encoded sequence. Hence, we tested the efficiency of DACs by accessing all the positions of each LCP array in random order. Figure 5.2 shows the space/times achieved for `dblp` (top), `dna` (center), and `proteins` (bottom) LCP arrays. The space for the integer encodings includes the space for the sparse sampling, where we varied the sample period to obtain the space/time trade-off. We also include Elias-Fano representation in this comparative.

DACs obtain the most compact space among all the alternatives when the optimal values for b are computed using the optimization algorithm, except for `dblp`. However, in this case, DACs are faster than those schemes that occupy less space, more concretely, Rice codes. In all the figures we can observe that DACs dominate the space/time trade-off.

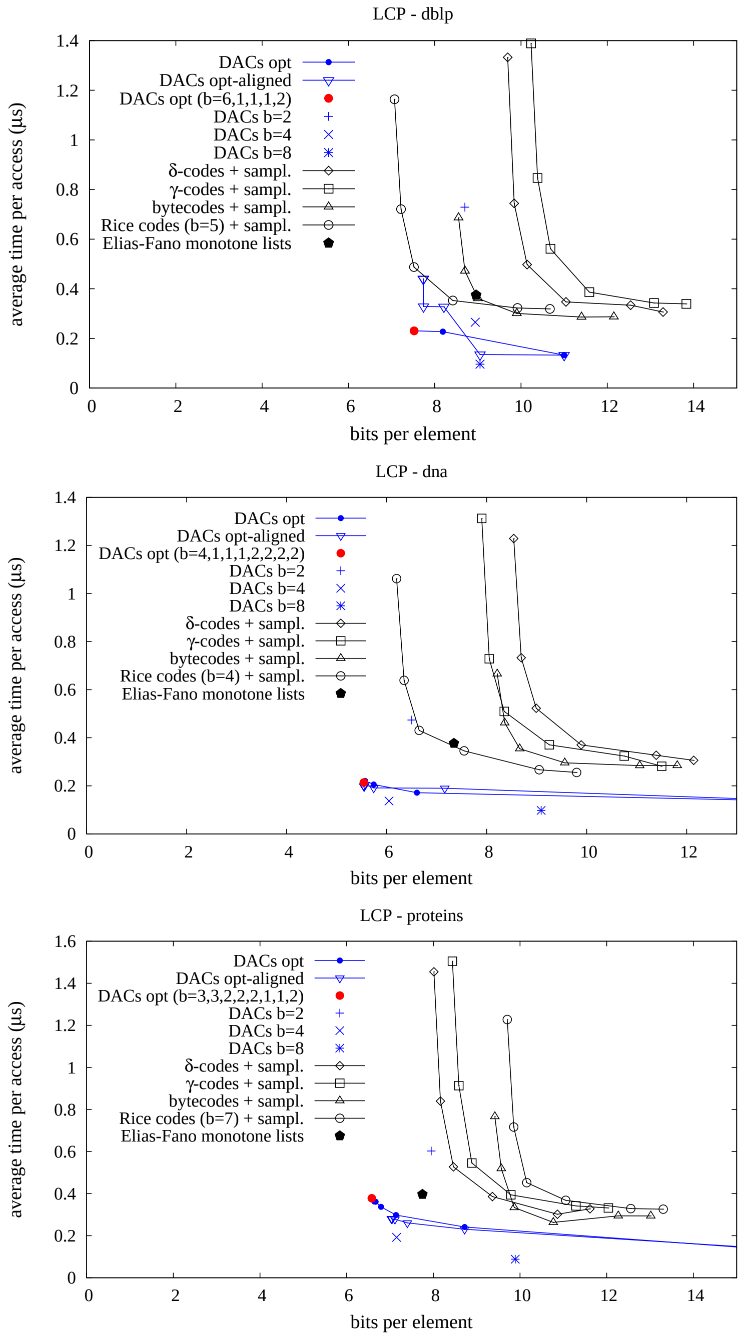


Figure 5.2: Space and average access time tradeoff for different configurations of DACs and other integer encodings when accessing to random positions of three LCP arrays.

5.2.2 High-Order Entropy-Compressed Sequences

Ferragina and Venturini [FV07] gave a simple scheme (FV) to represent a sequence of symbols $S = S_1 S_2 \dots S_n$ so that it is compressed to its high-order empirical entropy and any $O(\log n)$-bit substring of S can be decoded in constant time. This is extremely useful because it permits replacing *any* sequence by its compressed variant, and any kind of access to it under the RAM model of computation retains the original time complexity. Then, the compressed representation of the sequence permits us to answer various types of query, such as obtaining substrings or approximate queries, in efficient time without decompressing the whole compressed data.

The idea of Ferragina and Venturini is to split the sequence S of length n into *blocks* of $\frac{1}{2} \log n$ bits, and then sort the blocks by frequency. That is, they create a vocabulary with the different blocks of length $\frac{1}{2} \log n$ bits, count the number of times that each block appears in the sequence and then order those blocks in the vocabulary from higher to lower frequency. Then, each block will be represented by one integer p_i: the relative position of the block among the sorted list of blocks, that is, its position in the sorted vocabulary. The next step consists in replacing each block in the sequence by the assigned integer such that a sequence of integers is obtained. Then, the sequence is stored using a dense sampling, as explained in Section 3.2.2.

We compare Ferragina and Venturini's dense sampling proposal with our own representation using DACs, as well as a classical variant using sparse sampling with the bit-oriented and byte-oriented Huffman encodings (see Section 2.2.1). We also include a binary Huffman-shaped wavelet tree built over the sequence of symbols, which provides efficient access to any symbol of the sequence as explained in Section 2.3.2 for a balanced binary wavelet tree.

For the experiments of this section, we represent the sequence of k-tuples of a text, that is, we consider substrings composed of k characters as the source symbols of the text. We process the text obtaining the vocabulary of k-tuples that appear in the text, compute their frequency and sort them by frequency to obtain the p_i values. We obtain the representation of the text as the concatenation of all the codewords of the k-tuples of the text, the vocabulary of symbols and the codeword assignment if needed[5].

We took the first 200 MB of three different texts from Pizza&Chili corpus

[5] Our DACs and Ferragina and Venturini's encoding do not require any additional information about the codeword assignment, since this assignment does not depend on the probabilities of the symbols and a dense encoding is used (the codewords are consecutively assigned). Huffman-based encodings do require the storage of the codeword assignment as they need to reconstruct the Huffman tree to properly encode and decode. However, this additional information is minimal, since canonical Huffman is used, thus the extra space required is negligible.

Table 5.4: Size of the vocabulary composed of k-tuples for three different texts.

k	`xml`	`sources`	`english`
1	96	230	225
2	6,676	9,183	9,416
3	11,4643	208,235	77,617
4	585,599	1,114,490	382,398

(http://pizzachili.dcc.uchile.cl). We used a XML text, denoted by `xml`, containing bibliographic information on major computer science journals and proceedings[6]. We also used a text that contains source program code, denote by `sources`, formed by the concatenation of some .c, .h, .C and .java files from C and Java source code. Finally, we also used a natural language text, denoted by `english`, which contains some English text files. Table 5.4 shows the size of the vocabulary for each text when considering tuples of length k, with $k = 1, 2, 3, 4$.

We implemented the scheme FV proposed in the paper of Ferragina and Venturini [FV07], and optimized it for each scenario. Using the encoding scheme explained in Section 3.2.2, where an integer p_i is represented with $\lfloor \log p_i \rfloor$, the longest block description (corresponding to the least frequent block in the sorted vocabulary) requires a different number l of bits depending on the size of the vocabulary obtained. We use a two-level dense sampling, storing absolute pointers every c blocks and relative pointers of $\lceil \log((c-1) \cdot l) \rceil$ bits for each block inside each of those superblocks of c blocks. We adjust this setting for each text and k value to obtain the best space possible. For text `xml`, $c = 20$ for $k = 1, 2$, $c = 30$ for $k = 3$ and $c = 26$ for $k = 4$. For text `sources`, $c = 18$ for $k = 1, 2$, $c = 30$ for $k = 3$ and $c = 24$ for $k = 4$. For text `english`, $c = 20$ for $k = 1, 2$, $c = 30$ for $k = 3$ and $c = 28$ for $k = 4$.

We also implemented the classical solution to provide direct access to any block of the sequence, by encoding the different blocks with bit-oriented and byte-oriented Huffman codes and setting absolute samples every h codewords, $h = \{16, 32, 64, 128, 256\}$, so that partial decoding is needed to extract each value. This gives us a space-time tradeoff, which will be represented as curves in the figures.

In Section 2.3.2 we described how wavelet trees can represent a sequence of arbitrary symbols and compute rank, select and access operations efficiently over the sequence. Hence, we also include a Huffman-shaped wavelet tree as a solution to provide direct access to a sequence of arbitrary symbols. For the comparison, we

[6] Notice that this XML text is the same text used to obtain the LCP array denoted by `dblp` in Section 5.2.1.

create several binary Huffman-shaped wavelet trees with different sizes, varying the size for the extra structure used to compute fast binary rank and select operations. We use the implementation of Francisco Claude available at the Compact Data Structures Library (libcds)[7].

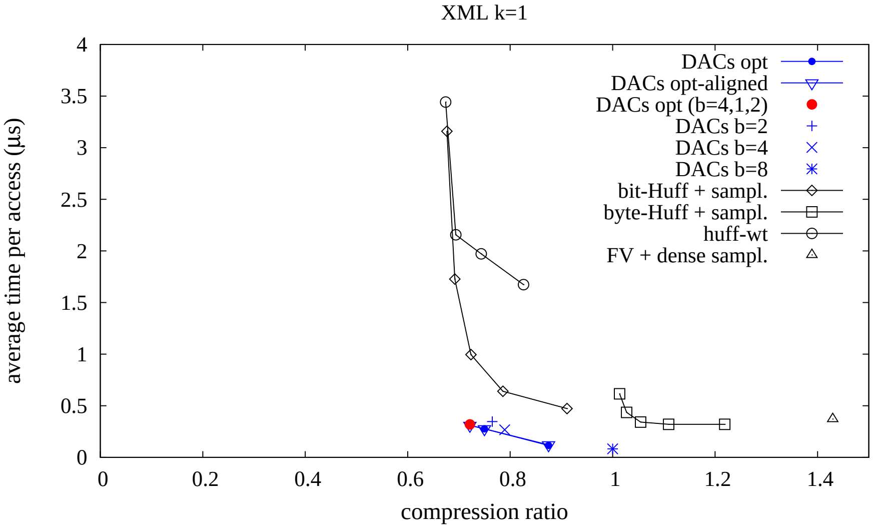
We compare those solutions with several configurations of DACs. When we use the same value for all the levels, we prefer powers of 2 for b, so that faster aligned accesses are possible. More concretely, we use $b = 2$, $b = 4$ and $b = 8$. We also use the b values obtained with the optimization algorithm, including the configurations where we restrict the number of levels of the representation and the byte-aligned approach.

We measure the space required by each alternative in terms of compression ratio and the average access time (in microseconds per accessed k-tuple) by computing the time to access all the k-tuples of the text in random order. We illustrate in the figures the space/time tradeoff of Ferragina and Venturini's dense sampling proposal ("FV + dense sampl."), bit-oriented Huffman code plus sparse sampling ("bit-Huff + sparse sampl."), byte-oriented Huffman code plus sparse sampling ("byte-Huff + sparse sampl."), the binary Huffman-shaped wavelet tree ("huff-wt") and our DACs with fixed b values for all the levels ("DACs b=2","DACs b=4","DACs b=8"), and the optimal b values that minimize the space ("DACs opt", where the optimal configuration without levels restriction is emphasized in red color in the figures, and "DACs opt-aligned" for the byte-aligned variant).

Figures 5.3 and 5.4 show the space/time tradeoff obtained by all the solutions applied over the text `xml` for $k = 1, 2$ and $k = 3, 4$ respectively. Figures 5.5 and 5.6 show the space/time tradeoff obtained by all the solutions applied over the text `sources` for $k = 1, 2$ and $k = 3, 4$ respectively. Figures 5.7 and 5.8 show the space/time tradeoff obtained by all the solutions applied over the text `english` for $k = 1, 2$ and $k = 3, 4$ respectively.

All the alternatives behave similarly over the three texts, where the differences are due to the size of the vocabulary at each scenario. When k is increased from $k = 1$ to $k = 4$, the compression obtained is generally better, since we are compressing the text at its k-order entropy, but the compression ratio is higher for $k = 4$ for some solutions (such as "huff-wt") due to the size of the vocabulary (which must be also stored). In fact, if we kept increasing k, we would obtain poorer compression ratios for all the alternatives, since the size required to store the vocabulary would be considerably larger than the reduction of size of the compressed text obtained from the k-order compression. We can also observe that the average access times are noticeably higher for large k values for some of the solutions employed. This is due to the size of the vocabulary, which increases the number of levels of the

[7] http://libcds.recoded.cl/



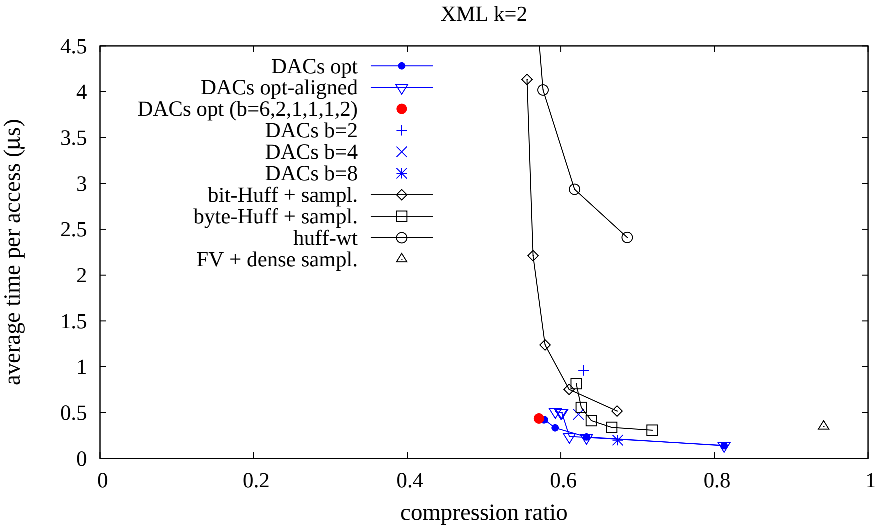
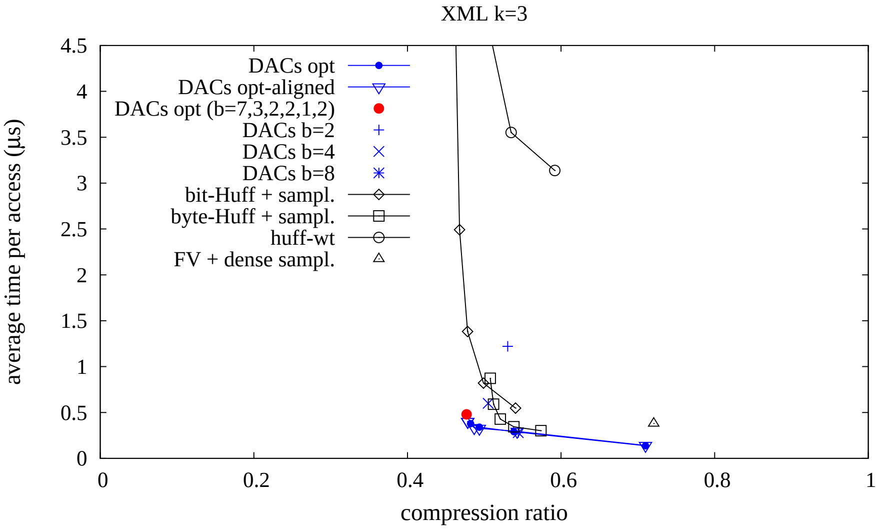


Figure 5.3: Space usage and average access time for several configurations of DACs versus several encodings that represent the sequence of k-tuples for a XML text when $k = 1$ (top) and $k = 2$ (bottom).



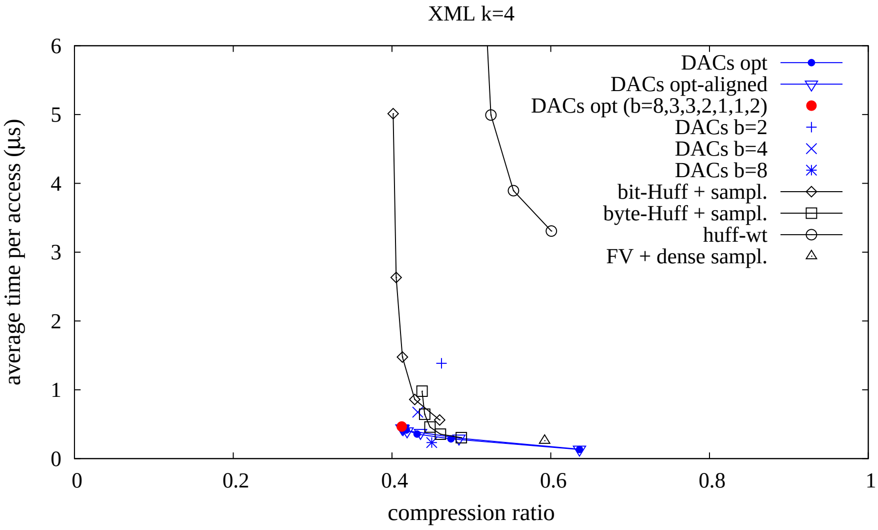
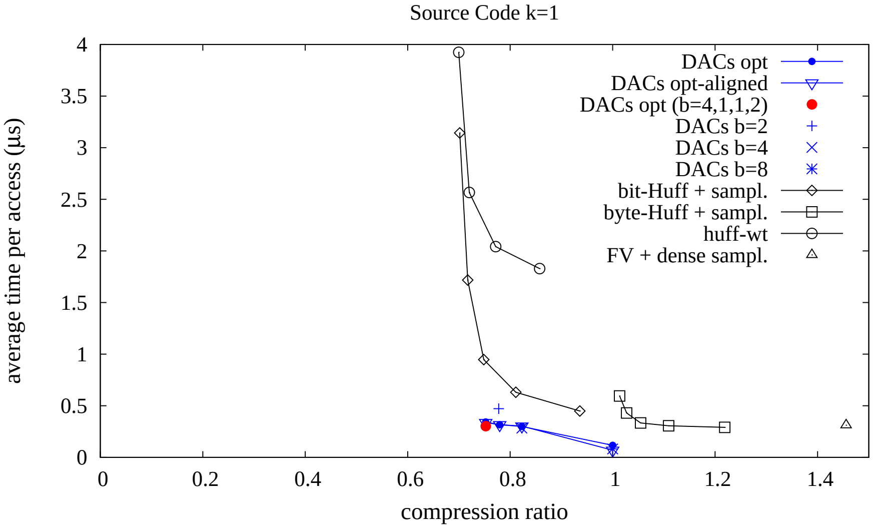


Figure 5.4: Space usage and average access time for several configurations of DACs versus several encodings that represent the sequence of k-tuples for a XML text when $k = 3$ (top) and $k = 4$ (bottom).



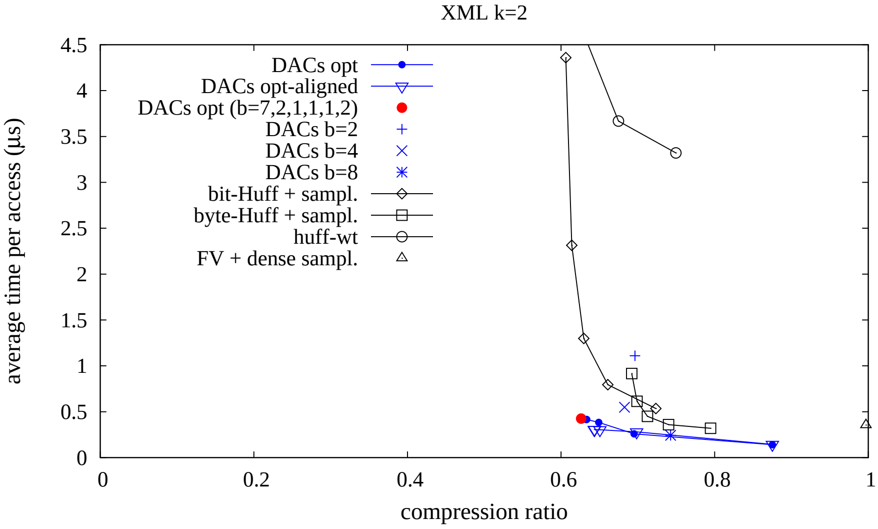
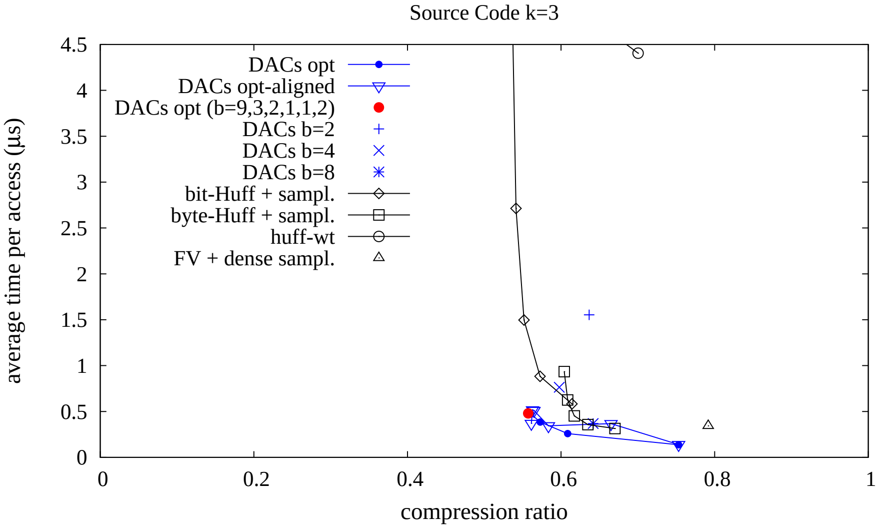


Figure 5.5: Space usage and average access time for several configurations of DACs versus several encodings that represent the sequence of k-tuples for a source code text when $k = 1$ (top) and $k = 2$ (bottom).



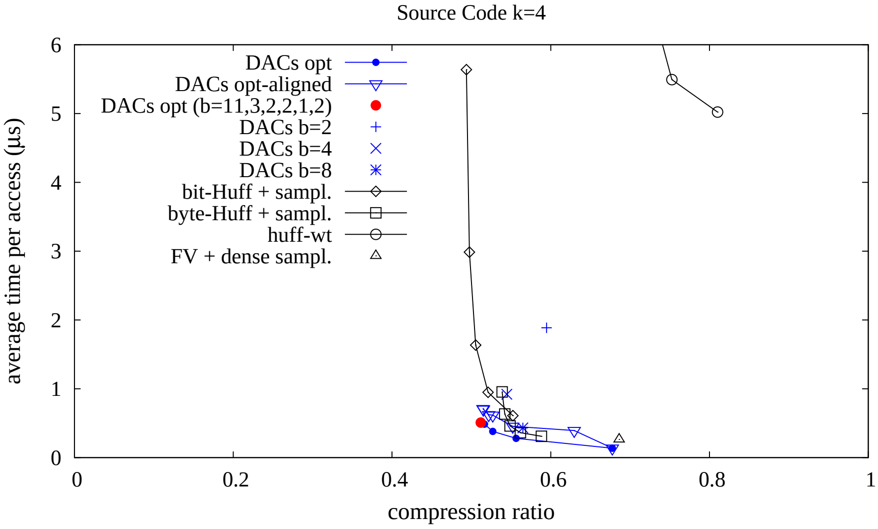
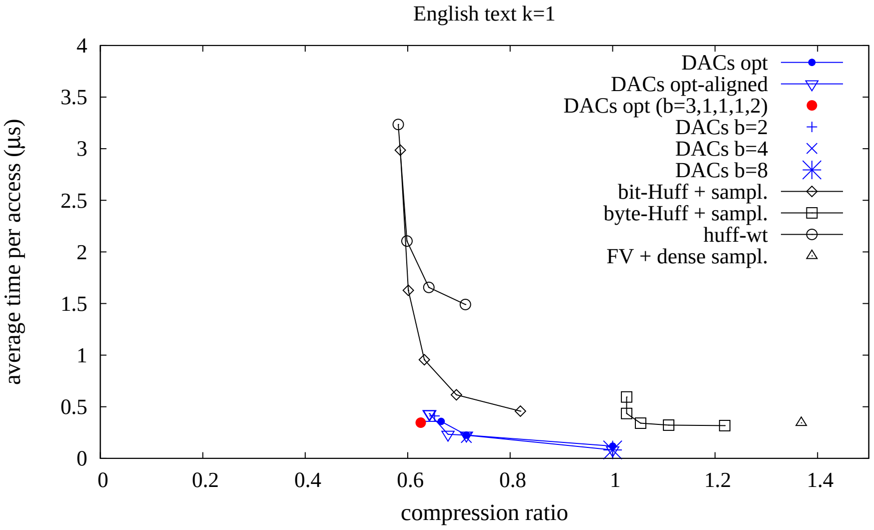


Figure 5.6: Space usage and average access time for several configurations of DACs versus several encodings that represent the sequence of k-tuples for a source code text when $k = 3$ (top) and $k = 4$ (bottom).



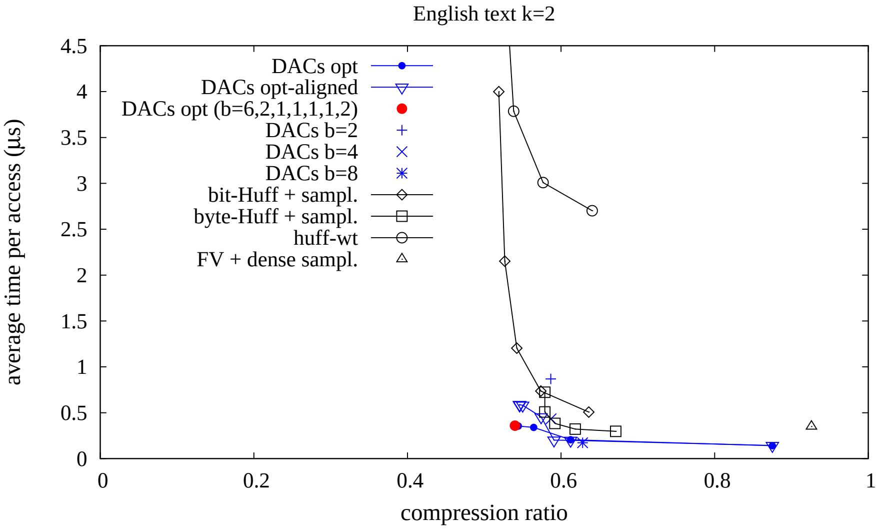
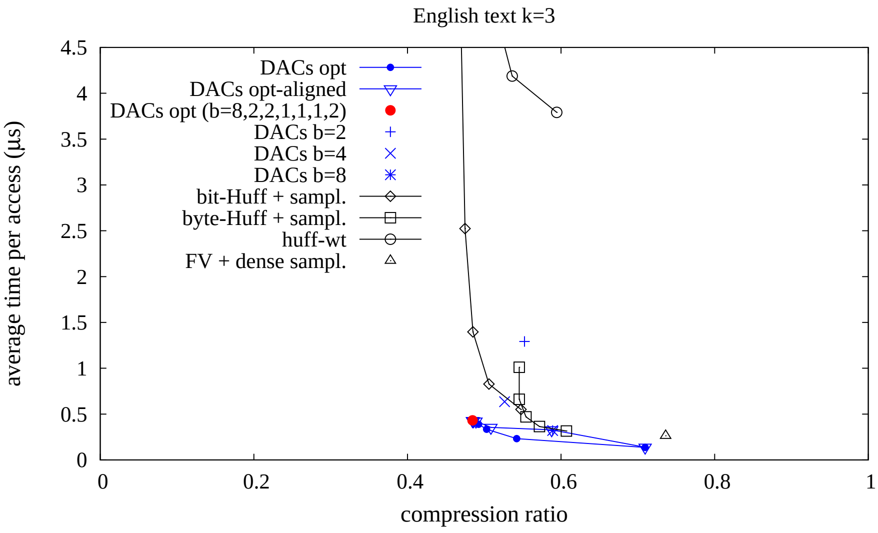


Figure 5.7: Space usage and average access time for several configurations of DACs versus several encodings that represent the sequence of k-tuples for a natural language text when $k = 1$ (top) and $k = 2$ (bottom).



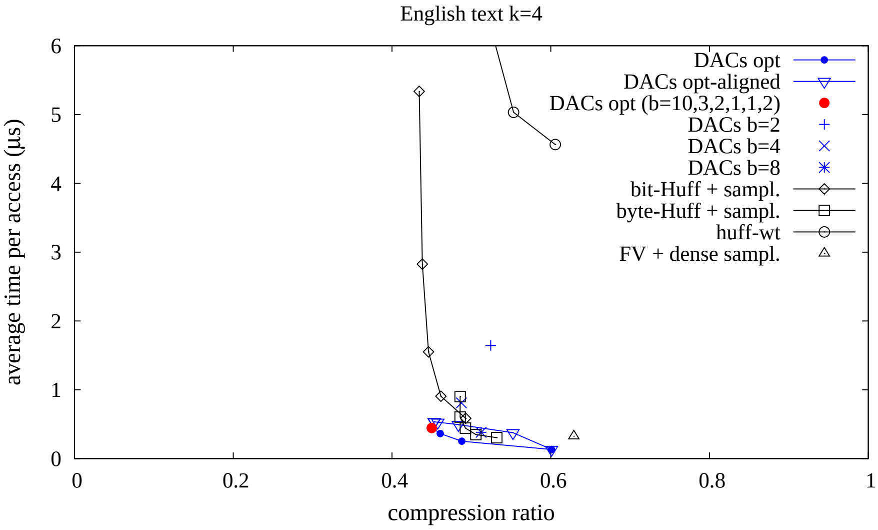


Figure 5.8: Space usage and average access time for several configurations of DACs versus several encodings that represent the sequence of k-tuples for a natural language text when $k = 3$ (top) and $k = 4$ (bottom).

representation when DACs and wavelet tree are used. Other solutions do not suffer the influence of this parameter, such as FV, where constant time is obtained due to the dense sampling.

The original FV method, implemented as such, poses much space overhead due to the dense sampling, achieving almost no compression. This, as expected, is alleviated by the bit-oriented Huffman coding with sparse sampling, but the access times increase considerably. The FV method extracts each block in constant time, while some extra decoding is always needed with the sparse sampling. Byte-oriented Huffman encoding with sparse sampling obtains, as expected, better times than bit-oriented Huffman encoding, but worsening the compression ratio. However, this byte-oriented alternative outperforms FV in space while being comparable in time. The binary Huffman-shaped wavelet tree behaves similarly to bit-oriented Huffman coding with sparse sampling for low k values, however its compression ratio and time efficiency are degraded as the size of the vocabulary grows, that is, as k increases.

The configuration of DACs with $b = 8$, which uses bytes as chunks, achieves much better space and noticeably better access times than FV for lower k values and comparable access times for $k = 3, 4$. When using the same space of a sparse sampling, on the other hand, "DACs b=8" is significantly faster. "DACs b=2" obtains competitive space and time results when $k = 1$. However, as the value of k increases, the number of levels grows, worsening the results of this alternative. "DACs b=4" behaves similarly, obtaining its best results when k is not high. DACs can improve the compression ratio obtained if the optimal b values are computed to minimize the space without restriction on the number of levels. As we can see in the figures, these optimal b values are adjusted according to the distribution of integers. For instance, we can observe how the size of the blocks at the first level increases as k grows, since the vocabulary is also increased. "DACs opt" and "DACs opt-aligned" obtain a competitive space/time tradeoff.

DACs using the b values obtained with the optimization algorithm can improve the compression ratio, however, sparse sampling can get lower spaces, as just the bare Huffman encoding, at the price of higher and higher access times. Hence, DACs become a very attractive solution if direct access must be provided to an encoded sequence, since it obtains fast times and considerably compact spaces.

5.2.3 Natural language text compression

We have seen in Section 5.1 that we can directly access to a compressed representation of a natural language text using our DACs. The faster alternative is obtained when $b = 8$, that is, when bytes are used as chunks, since it avoids bit-wise operations and takes advantage of the byte alignments.

In this section, we compare our faster alternative, denoted by "DACs b=8", with byte-oriented Huffman encoding, which is also faster than any bit-oriented encod-

Table 5.5: Description of the corpora used.

CORPUS	size (bytes)	num words	voc. size
CR	51,085,545	10,113,143	117,713
ZIFF	185,220,211	40,627,131	237,622
ALL	1,080,720,303	228,707,250	885,630

ing. The byte-oriented Huffman compressor that uses words as source symbols, instead of characters, is called *Plain Huffman*[8]. As we want to directly access to random words of the original text, we include a sparse sampling over the compressed sequence obtained by Plain Huffman. We denote this alternative "PH + sampl".

We used three corpora:

- Congressional Record 1993 (CR) from TREC-4.
- Ziff Data 1989-1990 (ZIFF) from TREC-2.
- A large corpora (ALL), with around 1GiB, created by aggregating the following text collections: AP Newswire 1988 and Ziff Data 1989-1990 (ZIFF) from TREC-2, Congressional Record 1993 (CR) and Financial Times 1991 to 1994 from TREC-4, in addition to the small Calgary corpus[9].

Table 5.5 presents the main characteristics of the corpora used. The first column indicates the name of the corpus, the second its size (in bytes). The third column indicates the number of words that compose the corpus, and finally the fourth column shows the number of different words in the text.

Table 5.6 shows the compression ratio, decompression time (in seconds) and access time (microseconds per access) for the two alternatives over all the corpora considered. "DACs b=8" uses the *rank* structure that occupies 5%-extra space over the sequence. We have adjusted the sampling parameter of the alternative "PH + sampl" to obtain the same compression ratio than "DACs b=8". The value of this parameter is shown in the table for each text: we store one sample each 24 codewords for CR corpus, one sample each 26 codewords for ZIFF corpus and one sample each 36 codewords for ALL corpus.

The decompression time includes the time, in seconds, to decompress the whole text, retrieving an exact copy of the original text. This procedure does not require

[8] Plain Huffman will be explained in Section 7.2, which is devoted to byte-oriented word-based text compression techniques. However, it can be briefly described as the word-based byte-oriented variant of the original Huffman code.

[9] http://www.data-compression.info/Corpora/CalgaryCorpus/

Table 5.6: Space and time performance for DACs and byte-oriented Huffman code (PH) when representing the sequence of words of three natural language texts.

	DACs b=8				PH + samp			
	ratio	t dec	t access		ratio	words per	t dec	t access
Text		(s)	(μs)			sample	(s)	(μs)
CR	0.3345	0.42	0.0544		0.3353	24	0.34	0.1938
ZIFF	0.3557	1.53	0.0761		0.3562	26	1.26	0.2581
ALL	0.3524	10.12	0.1088		0.3523	32	8.57	0.2838

the use of samples in the case of "PH + sampl", nor does it require the use of rank operations when "DACs b=8" is used, since all the levels of the representation can be sequentially processed and the synchronization between the bytes of the same codeword can be carried out using one pointer at each level, indicating the last byte read. Decompression is faster for PH than for DACs. Decompression just involves a sequential decoding of all the bytes of the encoded sequence in the case of PH. If DACs are used, it requires reading bytes at different levels of the representation, with are not contiguously located in memory. This procedure is not as fast as the sequential reads of PH. In addition, the compressed sequence using PH (without taking into account the sparse sampling) is shorter than the compressed sequence using DACs. Hence, PH processes a smaller number of bytes during the decompression procedure, which also speeds up the decompression time.

The access time was computed as the average time to access 10,000,000 words at random positions of the text. We can observe that "DACs b=8" obtains considerably better access times than "PH + sampl", around 3-4 times faster. It is also noticeable that for both alternatives, larger corpora obtain worse results than smaller corpora. In the case of "DACs b=8" this is due to the size of the vocabulary: since there are more different words in a larger text, there are many words that obtain larger codewords, and consequently the number of levels is bigger than for smaller corpora, which causes a higher number of rank operations when extracting those codewords. In the case of "PH + sampl", the sample period used is bigger for larger corpora, as we can see in Table 5.6, and this slows down the accesses to random words of the text.

Our proposal obtains better access time to individual words of the text, but it becomes slower when decompressing the whole text. We now analyze the time required by each alternative to access to t random consecutive positions of the text. When $t = 1$, we have already shown that "DACs b=8" is faster than "PH +samp'. When $t = n$, that is, when we are decompressing the whole text, PH is faster than DACs. Therefore, there must be a value r, with $1 \leq r \leq n$ where PH becomes faster

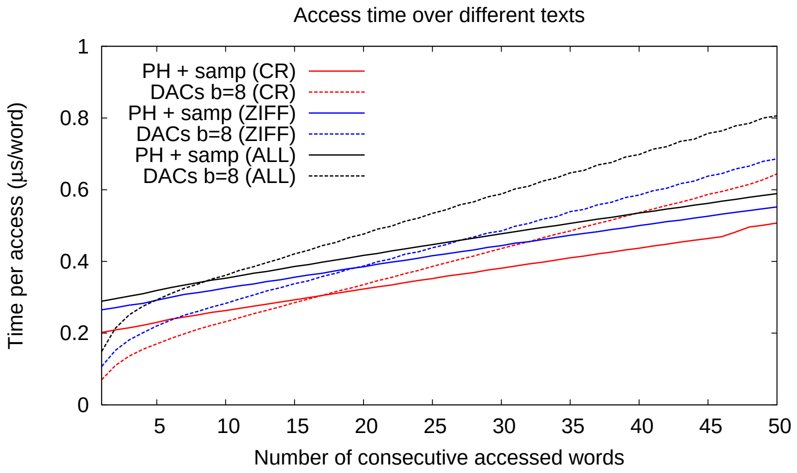


Figure 5.9: Accessing consecutive words for DACs (b=8) and PH (with sampling).

than DACs for all $t \geq r$. Figure 5.9 shows the average time to retrieve t consecutive words for the three corpora CR, ZIFF and ALL using "DACs b=8" and "PH +samp", where the sampling used is the same as in Table 5.6. We observe in the figure that "DACs b=8" outperforms "PH +samp" when the value of t is small, that is, when we access to few consecutive words of the text. As we increase t, the benefits of PH encoding, that is, the fact that we have a lower number of processed bytes that can be sequentially decoded, become noticeable, and "PH +samp" outperforms "DACs b=8" for larger t values. For instance, if we want to decompress 25 consecutive words, "PH +samp" becomes the preferred alternative. However, when accessing few consecutive words, such as five or less, "DACs b=8" obtains better time results, especially when accessing to just one word.

5.3 Other experimental results

The applicability of the technique is wide, and DACs have been used by other researchers and have been proved efficient in different domains, such as the representation of PATRICIA trees or compressed suffix trees, as we explain more in detail next.

PATRICIA tree

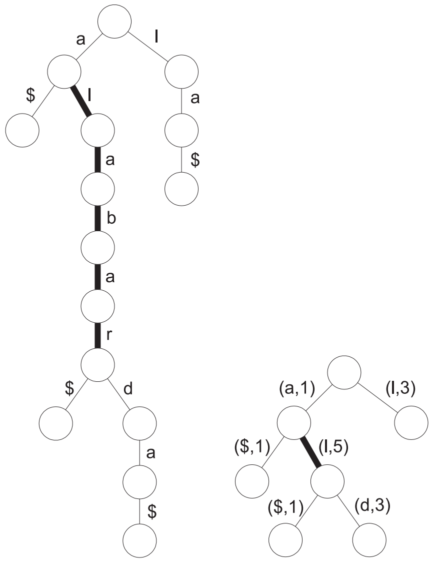
A *trie* or digital trie is a data structure that stores a set of strings over an alphabet. The height of the tree is the length of the longest string of the set. It has been used, for instance, to store large dictionaries of English words in spelling-checking programs. It can find all those strings of the set that are prefixed by a pattern in time proportional to the pattern length.

A *PATRICIA tree* [Mor68] differs from the trie data structure in that the PATRICIA tree stores only true branches. It collapses unary nodes, that is, those internal nodes in the trie that have only one descendant. This is also done by compact tries, but instead of storing the concatenation of the labels of the collapsed nodes, PATRICIA trees just store the first character of the label string and its length (we will call this length *skip*). This modification significantly reduces the size of the tree when the set of keys is sparse.

Figure 5.10 illustrates an example of these two structures for the set of strings S={'alabar', 'a', 'la', 'alabarda'}. Figure 5.10(a) shows the trie built over this set. Each node represents a distinct prefix in the set. We suppose that all strings are ended by a special symbol $, alphabetically smaller than any element of the alphabet. With this special character, the tree has exactly one leaf for each string of the set (in this case, the trie has exactly 4 leaves). As we can observe, there are long unitary paths which can be collapsed in just one edge. This is done by the PATRICIA tree in Figure 5.10(b). Just the first character of the collapsed path is stored in the tree, in addition to the skip. For instance, the path 'labar', which is marked with thick lines in Figure 5.10(a) is collapsed with just one edge in Figure 5.10(b), and represented with the pair $(l, 5)$, which consists of the first character of the string and the skip.

The average value for the skips in a PATRICIA tree is usually very low, but there are also long values in practice. Hence, these values can be efficiently represented using DACs.

An LZ77-Based Self-Index. Kreft [Kre10, KN11] proposed a new self-index oriented to repetitive texts and based on the Lempel-Ziv parsing, which parses the text into *phrases* so that each phrase, except its last letter, appears previously in the text, and compresses by replacing each phrase by a backward pointer. He uses some compact data structures to achieve the minimum possible space. Since the text is not stored, the self-index includes all the structures needed to randomly extract any substring from the text. The structures used include two tries: one sparse suffix tree that indexes all the suffixes of the text starting at the beginning of a phrase, and one PATRICIA tree that indexes all the reversed phrases, stored as a compact labeled tree. They are used to search the left and right side of the pattern sought. The theoretical proposal does not store the skips of these tries, as they can be computed from the trie and the text. However, this is a slow procedure,



(a) Trie (b) Patricia tree

Figure 5.10: Example of a trie and a PATRICIA tree for the set of strings S={'alabar', 'a', 'la', 'alabarda'}, and a long unary path that is compacted.

so Kreft considered storing the skips for one or for both tries using DACs. In the experimental evaluation of this new self-index, several variants are compared which include different structures and algorithms, obtaining a space/time tradeoff. In all those variants, skips are stored using DACs, because they give the best results.

Practical Compressed Suffix Trees

Cánovas and Navarro recently presented [CN10a] a new practical compressed suffix tree implementation, based on a theoretical proposal by Fisher *et al.* [FMN09]. According to the authors, the efficient implementation of the proposal was not trivial, so they developed different structures and solutions. Their implementations offer a relevant space/time tradeoff between the two most well-known solutions to the problem, which are inefficient in either time or space.

A *suffix tree* is a compact trie storing all the suffixes of a text T. If the children of each node are ordered lexicographically by their string label, the leaves of the suffix tree form the *suffix array* of T (suffix arrays will be explained more in detail in Section 7.3.2). A *compressed suffix tree (CST)* of a text can be represented using the compressed suffix array of the text and storing some extra information:

the tree topology and the longest common prefix (LCP) information. The practical implementation of this new solution can be divided in two challenges: the efficient representation of the *LCP* array and the efficient computation of some common queries over the *LCP* (range minimum query and previous/next smaller value query). These queries enable all the navigation over the suffix tree without the need of representing the topology of the tree.

Among the solutions proposed, Cánovas and Navarro studied the use of DACs and the optimization algorithm in order to represent the *LCP* array. The technique takes advantage of the fact that, as we have already seen in Section 5.2.1, most *LCP* values are small, and some can be much larger. Hence, a variable-length encoding scheme is a good solution to represent that sequence of integers. Moreover, the operations that are performed over the *LCP* array (which support the navigation in the suffix tree, such as finding the next sibling, the parent or children of a given node) must support direct access to any position of the array. Then, our technique fits perfectly with the needs of the *LCP* array representation in this context. They used two variants of DACs: one corresponds to a fixed b value for all the levels of the representation, while the other uses the b values computed with the optimization algorithm that minimizes the space of the representation. Both of them offer interesting results in the space/time trade-off map, giving, by far, the best time performance of all the alternatives proposed to efficiently represent the *LCP*.

The final CST implementation using DACs for the *LCP* representation occupies between 13-16 bits per symbol and carries out most operations within a few microseconds, being faster than the previous existing implementations of CST (including some that need more space), and requiring an affordable extra space.

Efficient representation of grammars

Claude and Navarro [CN09] proposed an indexed compressed text representation based on Straight-Line Programs (SLP), a restricted kind of grammar, so that a text $T[1,u]$, over alphabet $\Sigma = [1,\sigma]$, can be represented with a grammar with n rules, occupying $O(n \log n) + n \log u$ bits. This last space term is due to the storage of the lengths of the rules of the grammar. That is, if $\mathcal{F}(X)$ is the expansion of a non-terminal X into terminals, the representation of $|\mathcal{F}(X)|$, the length of the phrase $\mathcal{F}(X)$, for each one of the n rules of the grammar requires $n \log u$ bits. The proposed structure supports operations extract and find in $o(n)$ time, where operation extract returns any desired portion $T[l, l+m]$ of the text and operation find returns the positions of T where a given search pattern $P[1,m]$ occurs in T. In addition, this technique can also be used to represent a labeled binary relation. This type of compression is very promising for highly repetitive sequences, which arise in applications such as computational biology, software repositories, transaction logs, versioned documents, temporal databases, etc.

This theoretical proposal has been used to create a compressed index specialized on searching short substrings (q-grams) on highly repetitive sequences [CFMPN10], by representing the rules generated by Re-Pair [LM00], a dictionary-based compression algorithm. The practical implementation of the proposal uses DACs for the representation of the lengths of the rules, reducing considerably the $n \log u$ space term for real data. For instance, using a collection of repetitive texts, they obtain a compression ratio close to 0.32%, where the generated grammar consists of 100,762 rules that would occupy 214,119 bytes using the $n \log u$ representation. However, DACs with $b = 4$ occupied 134,151 bytes (62.65%). For another collection composed of 27 biological sequences, they achieve a compression ratio close to 11.36%, where the generated grammar consisted of 3,093,368 rules. These rules occupied 8,506,762 bytes using a $n \log u$ representation, but DACs with $b = 4$ occupied 3,863,681 bytes (45.42%).

Chapter 6

Discussion

6.1 Main contributions

In this part of the thesis we have introduced the *Directly Addressable Codes (DACs)*, a new encoding scheme for sequences of integers that enables easy and direct access to any element of the sequence. It achieves very compact spaces, bypassing the heavyweight methods, based on sampling, used in current schemes. This is an important achievement because the need of random access to variable-length codes is ubiquitous in many sorts of applications, particularly in compressed data structures, but also arises in everyday programming. Our method is simple to program and is space- and time-efficient, which makes it an attractive practical choice in many scenarios.

We first explained the proposal in Chapter 4. DACs divide each integer into $\lceil l/b \rceil$ blocks of b bits, where l is the length of the binary representation of the integer and b is a fixed parameter of the representation. Each block is stored in a chunk of $b + 1$ bits, using 1 extra bit to indicate whether the code of the integer continues in the next block or finishes in the current one. Those blocks are rearranged in several levels, with the first level of the representation gathering all the least significant blocks of all the integers, the second level with the second least significant blocks, and so on, up to the last level of the representation, which contains the most significant blocks for the largest integers. This rearrangement in levels allows fast random access, so it is possible to directly access to any integer of the sequence in an efficient way, without the need of any sampling method.

Space and time efficiency can be improved by using different b values at each level of the representation, being b the size in bits of the blocks in each level. Instead of using a fixed b value for all the representation, a variable b value for each level permits a better adjustment to the frequency distribution of the integers of the sequence to encode. Thus, the representation becomes most compact without impairing the efficiency if it does not generate a high number of levels. Hence, we propose in Section 4.2 an optimization algorithm that, given a sequence of integers and their frequency distribution, obtains the b value for each level that minimizes the space occupied by the compact representation of that sequence of integers.

In Chapter 5, we have shown experimentally that our technique competes successfully with other solutions which encode and enable direct access to the represented sequence. We analyze the behavior of DACs in two different scenarios: when the variable-length encoding is used along with a statistical modeling or when it is used to represent sequences of integers that are frequently small, but they can also be larger. In both scenarios, we want to support fast direct access to any element of the encoded sequence. In the first case we compare our proposal with other statistical encodings such as bit-oriented Huffman and byte-oriented Huffman with sparse sampling, a Huffman-shaped binary wavelet tree representing the sequence and the dense sampling solution of Ferragina and Venturini. For the second scenario, we study the space/time trade-off of our representation and compare it with the space and time achieved by other integer encodings, such as δ-codes, γ-codes, Rice codes and byte codes. In both scenarios, DACs outperform the other alternatives when providing direct access to the encoded sequence, obtaining very compact spaces.

The conceptual description of the technique and some application results were published [BLN09a].

6.1.1 Interest of the rearrangement

We have not only presented a new encoding scheme for integers. The rearrangement strategy used in our proposal can be seen as a contribution by itself that could provide synchronism to any encoded sequence of symbols obtained after using a variable-length encoding technique.

We have presented a rearrangement of the Vbytes codewords in several levels in order to obtain direct access to any codeword. In practice, we rearrange the chunks of the codewords obtained by End-Tagged Dense Codes (see Section 7.2.3 for a complete description of its encoding scheme). But more generally, we can rearrange the codewords obtained by any encoding scheme by splitting them into chunks (or bytes when using a byte-oriented encoding scheme) and relocating those chunks in several levels, adding a bit array that indicates whether the codeword continues in the next level or not.

We can also use this rearrangement over other encodings after using a statistical modeler. For example, we could consider the byte-oriented Huffman encoding for a sequence of symbols and then create the same data structure that we propose for Vbyte encoding, that is, placing the bytes of the byte-oriented Huffman code in levels. However, the use of the extra bit array to obtain the synchronization between the bytes of the same codeword makes unbeneficial the use of the Huffman encoding, since the additional bit already produces a prefix free code. Hence, instead of using a Huffman encoding in the chunks of the levels, a dense encoding that uses all the possible combinations of codewords, as DACs encoding, becomes the optimal solution. Therefore, the encoding scheme for integers we propose is the preferred representation among any rearrangement of codewords obtained with a statistical compression technique when direct access must be provided to a compressed sequence of symbols.

However, we can also use the rearrangement strategy over the codewords obtained using other non-statistical compression techniques. For instance, one can provide direct access to a sequence of symbols compressed with Re-Pair [LM00], which is a dictionary-based compression algorithm that assigns a fixed-length codeword to a variable-length sequence of symbol. Let us imagine we have a sequence of strings, such as a list of URLs, and we want fast direct access to each string. We can compress those strings with Re-Pair, such that each string will be composed of a variable-length sequence of codewords. Hence, those sequences can be split in several levels, following the idea of DACs, and direct access can be achieved by including one bitmap per level indicating if the representation of the string continues in the next level or not, instead of using sparse sampling. Hence, direct access to any encoded sequence obtained after using any compression technique can be supported following the rearrangement proposed in this thesis.

6.2 Other Applications

DACs obtain very attractive times results when direct access to a compressed sequence of symbols is required. As we have seen in Section 5.3, they have been successfully used to improve the performance of classical data structures, such as the representation of the LPC array or PATRICIA trees.

Compressed Suffix Trees [CN10a] can be efficiently implemented in practice using a compact and directly addressable representation of the LCP array. Hence, Directly Addressable Codes are suitable for this scenario because most *LCP* values are small, and some can be much larger. The direct access obtained with our technique is vital for the performance of their data structure, since it is required to navigate over the suffix tree. Two variants of the DACs are used for the representation of the LCP array. The first one corresponds to a fixed b value for all the levels of the representation, while the other uses the b values computed with the optimization algorithm that minimizes the space of the representation. Both of them offer interesting results in the space/time tradeoff map, giving, by far, the best time performance of all the alternatives proposed to efficiently represent the *LCP*. The final CST implementation using any of the two alternatives for the *LCP* representation occupies between 13-16 bits per symbol and carries out most operations within a few microseconds, being faster than the previous existing implementations of CST (including some that need more space), and requiring an affordable extra space. This *LCP* representation using DACs was also compared in the proposal of a *sampled LCP array* by Jouni Sirén [Sir10].

PATRICIA trees index a set of strings in a compact way by collapsing several unary nodes and storing just one character and the length of the collapsed path for each node. These lengths can be represented using a variable-length encoding scheme, since they are usually small values. However, direct access to these values is required, so the use of Directly Addressable Codes has been proved to be a very efficient solution. An example of this use is a recent work that implements a self-index based on LZ77 [Kre10]. This self-index includes two tries that store all the phrases obtained by the LZ77 parsing of the text, one of them in reverse order. These tries are compacted by collapsing unary nodes, and DACs are used to represent the length of the collapsed paths. The experimental results of this self-index indicates that the use of Directly Addressable Codes becomes the best alternative to represent those values.

DACs have been successfully used to reduce the space usage of an indexed representation of a *grammar*. A recent work [CN09] presented a self-index technique for straight-line programs (SLPs), a restricted kind of grammar. This proposal has been used to provide a compressed storage scheme for highly repetitive sequence collections, while providing efficient indexed search for q-grams [CFMPN10]. In this work, the length of the expansion of each rule of the grammar has been represented with DACs, reducing the space requirements significantly.

DACs are also being used in ongoing research work on suffix trees (N. Herrera, PhD thesis, personal communication), string B-trees (C. Ruano, MSc thesis, personal communication), representing in a more compact form the output of Re-Pair compressors (D. Valenzuela, MSc. thesis, personal communication), representing dictionaries, among others.

Finally, we will see how DACs can also be used in *Web graph compression*, discussed in Part III of this thesis. We propose a technique to represent Web graphs in a very compact space using a tree-shaped data structure. This data structure supports extended navigability over the compressed graph. We will see that DACs can be used to represent the leaves of the tree, obtaining better space and time results than the initial proposal without DACs. Moreover, with the use of DACs, the compression technique for Web graphs proposed in Chapter 12 becomes the most space-efficient method of the state of the art that provides forward and backward navigation over the graph. It obtains the smallest space compared to other techniques, without considerably degrading navigational times.

Part II

Reorganizing Compressed Text

Chapter 7

Introduction

In the Part I of this thesis we presented a new strategy to represent sequences of integers using variable-length codes. Directly Addressable Codes (DACs) enable efficient direct access to any position of the encoded sequence by rearranging the codeword chunks into different levels. If we use this encoding to represent a natural language text, as in Section 5.2.3, we obtain a compressed representation of the text where direct access to any word of the text is efficiently supported. However, finding words or phrases on the compressed representation of the text, such as counting or locating their occurrences, cannot be efficiently performed using DACs. This is due to the fact that all the second chunks of the codewords are mixed together in the second level, all the third chunks are located in the third level and so on, and so sequential scans of the complete levels must be performed.

In this part of the thesis we propose a new data structure that represents a natural language text in compressed way, which is inspired in the DACs strategy and considerably improves the efficiency of searches in the text. As in the DACs, we will also rearrange the codewords into several levels to obtain direct access. Therefore, we can start decompressing from any position of the compressed text, and display any portion of the text. Culpepper and Moffat [CM06] already proposed in 2006 a separation between the fist byte of a codeword and the rest of the bytes in order to gain efficiency in sequential pattern matching algorithms over compressed texts. A code splitting strategy was already used to improve string matching algorithms [RTT02]. The data structure we propose goes one step beyond. We separate the chunks of the codewords into distinct branches depending on the preceding chunks, forming a tree-shaped data structure. In this way, in addition to the relocation of the chunks in several levels, as with the DACs, we follow a multi-ary wavelet tree strategy to improve searches in the compressed text. Implicit indexing properties are achieved by this separation of the levels into different tree nodes. The goal is to have a single path from the root node of the tree to a leaf representing each complete codeword, such that we will be able to search for any word in the compressed text in time independent of the text length. This rearrangement can be applied to the compressed text obtained by any word-based, byte-oriented prefix-free encoding technique.

Hence, in the next chapters of this thesis we will show that by just performing a simple rearrangement of the codeword bytes of the compressed text (more precisely, reorganizing the bytes into a wavelet-tree-like shape) and using little additional space, searching capabilities are greatly improved without a significant impact in compression and decompression times. With this approach, all the codes achieve synchronism and can be searched fast and accessed at arbitrary points. Moreover, this new data structure can be regarded as an *implicitly self-indexed* representation of the text, which can be searched for words in time independent of the text length. That is, we achieve not only fast sequential search time, but indexed search time, for almost no extra space cost.

Section 7.1 starts with a revision of the state of the art in text compression, explaining how compression methods that consider words as source symbols obtain better compression properties than those using characters. Section 7.2 describes several word-based compression techniques of interest; some of them will be used in the experimental evaluation of the new data structure proposed. Section 7.3 studies the indexing problem, describing more in detail the most used solutions, that is, inverted indexes and suffix arrays, and introducing the newer concept of *self-indexes*, that is, indexes that operate in space proportional to that of the compressed text. Finally, Section 7.4 briefly summarizes the pursued goal in the next chapters, which consists in a new data structure that represents a natural language text in a compressed and self-indexed form, such that an interesting search performance is obtained.

7.1 Natural Language Text Compression

Current Text Databases contain hundreds of gigabytes, and there are terabytes of documents in the Web. Although the capacity of new devices to store data grows fast and the associated costs decrease, the size of text collections increases faster. Moreover, CPU speed grows much faster than that of secondary memory devices and networks, so storing data in compressed form reduces not only space, but also the I/O time and the network bandwidth needed to transmit it. Compression techniques have become attractive methods that can be used in Text Databases to save disk space, but more importantly, to save processing, transmission and disk transfer time.

Compressing the text as much as possible is important. However, if the compression scheme does not allow us to search directly the compressed text, then the retrieval over such compressed documents will be less efficient due to the necessity of decompressing them before the search. Even if the search is done via an index, some text scanning is needed in the search process [MW94, NMN$^+$00]. Therefore, even in these cases, it is important that the compressed text supports searches. In addition, it is desirable that the compression technique supports *direct access* to the compressed text, what enables decompressing random parts of the compressed text without having to process it from the beginning. Summarizing, compression techniques are well-suited for *Text Retrieval* systems iff:

i) they achieve good compression ratio,

ii) they maintain good search capabilities without decompressing the text, and

iii) they permit direct access to the compressed text.

Traditionally, classical compressors used characters as the symbols to be compressed, that is, they regarded the text as a sequence of characters. Classical Huffman [Huf52] computes the frequencies of the characters of the text and assigns shorter codes formed by variable length sequences of bits to more frequent characters. Then each character of the text is replaced by its codeword. Unfortunately, the compression achieved when applying classical Huffman to English natural language text is poor (around 65%).

Other techniques, such as Ziv and Lempel algorithms [ZL77, ZL78], replace text substrings by pointers to previous occurrences. They are commonly used due to their compression and especially decompression speeds, but their compression ratio is still not that good (around 35-40%) on natural language text.

Some techniques obtain better compression ratios by using a k-order model of the text, such as PPM (Prediction by Partial Matching) compressors [BCW84], which couple such modeling with an arithmetic coder [Abr63]. Compression ratio is very good, around 19-26%, but they are very slow at compression and decompression and require much memory. Similar results are obtained by Seward's bzip2[1], which can use less memory than PPM and obtain attractive ratios (around 24-29%), while being much faster at both compression and decompression.

There are some techniques that obtain very good compression ratios, by making several passes over the source text, such that compression is improved after each new pass. These so-called *offline compressors* are very time and/or space demanding procedures, and they are not always suitable. However, they are fast and memory-efficient at decompression. One well-known example of this approach is Re-Pair [LM00], which successively replaces the most frequent pair of adjacent source symbols by a new symbol until all the pairs occur only once, obtaining high compression on natural language (around 20-31%)

[1] `http://www.bzip.org`

Classic compression techniques, like Ziv and Lempel [ZL77, ZL78] or classic Huffman [Huf52], permit to search directly on the compressed text [NT00]. Empirical results showed that searching the compressed text can take half the time of decompressing that text and then searching it, but it is slower than searching the uncompressed text. Therefore, searching over the compressed text obtained by these techniques is useful if the text has to be kept compressed. However, storing the plain version of the text becomes the preferred choice, instead of using these classic compression techniques, when efficient searches are pursued and there are not serious space restriction.

An important change in the history of text compression was produced by the revolutionary idea of compressing natural language text using words as the source symbols, instead of using characters [BSTW86]. Compression techniques following this *word-based model* obtain better compression ratios and search performance, since they permit searching the compressed text much faster than the original text [TM97, MNZBY98] and achieve compression ratios around 25%-35%.

There are two empirical laws, Heaps' and Zipf's, which describe some of the properties of natural language texts and explain why using words instead of characters improves the compression achieved. *Heaps' law* gives an approximation to how a vocabulary grows as the size of a text collection increases, whereas *Zipf's law* gives an estimation of the word frequency distribution for a natural language text. Therefore, they provide interesting information about the number of distinct source symbols in the text, and help to estimate the frequency of those symbols.

- Heaps' law [Hea78] establishes that the relationship between the number of words in a natural language text (n) and the number of different words (V) in that text (that is, words in the vocabulary) is given by the expression $V \approx \alpha n^{\beta}$, where α and β are free parameters empirically determined. In English text corpora, their typical values are $10 \leq \alpha \leq 100$ and $0.4 \leq \beta \leq 0.6$. For natural language text corpora, Heaps's law also predicts the vocabulary size (V) from the size of the text in bytes (i), $V = K \times i^{\beta}$.

- Zipf's Law [Zip49] gives a good estimation for the word frequency distribution in natural language texts [BCW90]. Simplifying the formula, the frequency of a word is $f = k/r^{\theta}$, where θ is a constant that depends on the analyzed text ($1 < \theta < 2$) and r is the rank of the word in the vocabulary and k is a constant. Hence, the frequency of a word is inversely proportional to its rank in the vocabulary.

Following Zipf's Law, words present a more biased distribution of frequencies than characters [BYRN99]. Thus the text (regarded as a sequence of words) is highly compressible with a zero-order model. By using words one captures k-th order statistics for a reasonable value of k, while ensuring that the model is not too large (as the vocabulary grows sublinearly with the size of the text collection, according to Heaps' law). With Huffman coding, compression ratios approach 25%. In addition, word-based compression techniques are especially interesting for IR Systems, since words are the basic elements on which most IR systems are built. The vocabulary of source symbols of the compressor is the same vocabulary used by the IR system. This permits a natural integration between IR and word-based compression methods.

Using words as source symbols instead of characters improves compression. Using sequences of bytes instead of bits as target symbols improves time performance. Different word-based compression methods follow this approach, such as Plain Huffman [MNZBY00] or Restricted Prefix Byte Codes [CM05]. The compression achieved is not as good as for binary Huffman code, since the use of bytes instead of bits degrades ratios to around 30%. However, decompression speed is significatively improved.

Still other encoding methods, such as Tagged Huffman codes [MNZBY00], End-Tagged Dense Codes, and (s, c)-Dense Codes [BFNP07], worsen the compression ratios a bit more in exchange for being *self-synchronized*. By using self-synchronized codes, codeword boundaries can be distinguished starting from anywhere in the encoded sequence, which enables random access to the compressed text, that is, permitting the decompression to start at any position of the compressed text. In addition, they also support very fast Boyer-Moore-like direct search [BM77, Hor80] of the compressed text. These algorithms skip some bytes during the search, such that it is not necessary to check every byte of the text against the pattern. They use a search window corresponding to the search pattern that is moved along the text. It is firstly aligned with the leftmost part of the text, and then the pattern is compared right-to-left against the text in the window until they match (or until a difference between the pattern and the text in the window appears). In each step, the longest possible safe-shift to the right of the window is performed. Due to the property of self-synchronization, since the boundaries of the codewords can be easily known, no false matchings can happen when searching directly over compressed text obtained by these techniques. Tagged Huffman obtains compression rations around 35% of the original text and End-Tagged Dense Codes improve Tagged Huffman by around 2.5 percentual points. Finally, (s, c)-Dense Codes obtains better compression than these two techniques, achieving compression ratios very close to Plain Huffman (only $+0.2$ percentual points).

The next section is devoted to briefly explain these word-based byte-oriented compression methods.

7.2 Word-based Bytewise Encoders

We now describe some byte-oriented encoding methods that are frequently used when compressing natural language text. We only cover those techniques that will be used in the next chapters of this thesis; many others exist, e.g. [MNZBY00, BFNP07]. The main advantage of these techniques is that decompression and searching are faster with byte-oriented codes than with bit-oriented codes because no bit manipulations are necessary. This fact permits that searching can be up to eight times faster for certain queries [MNZBY00].

7.2.1 Plain Huffman

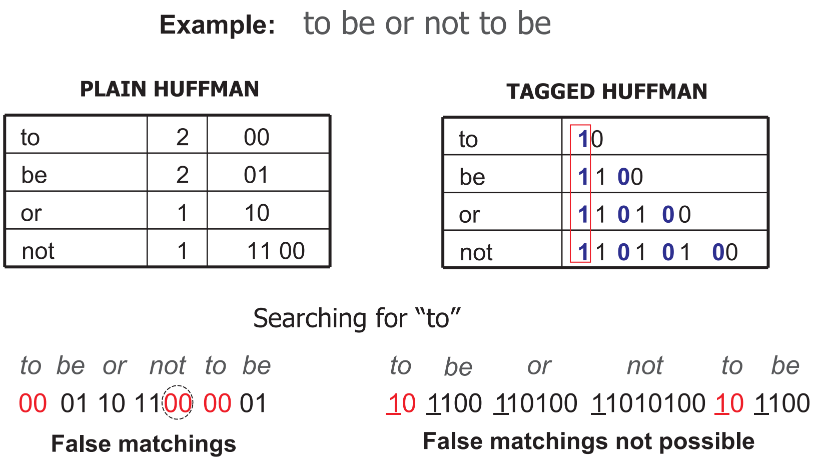
The original bit-oriented Huffman coding achieves compression rations around 25% when it is applied over natural language text and words are used as symbols instead of characters [Mof89].

The basic word-based byte-oriented variant of the original Huffman code is called *Plain Huffman (PH)* [MNZBY00]. Plain Huffman does not modify the basic Huffman code except for the use of bytes as the symbols of the target alphabet. This change worsens the compression ratios to 30%, instead of the 25% achieved by bit-oriented Huffman code. In exchange, as we have previously mentioned, decompression and searching are much faster.

If Plain Huffman has been used to compressed a text, we cannot search for a pattern in the compressed text by simply compressing the pattern and then using a classical string matching algorithm that jumps over the compressed text. This does not work because the pattern could occur in the text and yet not correspond to our codeword. Concatenations of parts of two codewords may match with the codeword of another vocabulary word (see Figure 7.1 for an example). Therefore, searching for a word in a text compressed with the Plain Huffman scheme requires a sequential search over the compressed text, reading one byte at a time. It first performs a preprocessing phase that searches for and marks in the vocabulary those words that match the search pattern. Then, a top-down traversal of the Huffman tree is performed, returning all those words associated to leaves that have been marked during the preprocessing step.

7.2.2 Tagged Huffman

Tagged Huffman (TH) [MNZBY00] is a variation of Plain Huffman that allows for an improved search algorithm. This technique is like the previous one, differing only in that the first bit of each byte is reserved to flag the first byte of a codeword. Then, a Huffman code is assigned using the remaining 7 bits of each byte, in order to obtain a prefix code.



to	2	00
be	2	01
or	1	10
not	1	11 00

to	10
be	11 00
or	11 01 00
not	11 01 01 00

Figure 7.1: Example of false matchings in Plain Huffman but not in Tagged Huffman codes. Note that we use special "bytes" of two bits for shortness.

Since the first bit of each byte signals the beginning of a codeword, no false matches can happen in Tagged Huffman Code. Therefore, Boyer-Moore-type searching is possible over Tagged Huffman Code. We can observe a comparison between Plain Huffman and Tagged Huffman in Figure 7.1, where false matches occur if Plain Huffman is used but not with Tagged Huffman.

Another important advantage of using flag bits is that they *synchronize* the codewords. Tagged Huffman permits *direct access* to the compressed text and therefore *random decompression*. That is, it is possible to access a compressed text, and start decompressing it without the necessity of processing it from the beginning. Those encoding schemes that support these characteristics are called *self-synchronizing* codes. In the case of Tagged Huffman, it is feasible to quickly find the beginning of the current codeword (synchronization) by just looking for a byte whose flag bit is 1.

Tagged Huffman Code obtains all these benefits at the expense of worsening its compression ratio: full bytes are used, but only 7 bits are devoted to coding. The loss of compression ratio is approximately 3.5 percentage points. As a compensation, Tagged Huffman searches compressed text much faster than Plain Huffman because Boyer-Moore type searching algorithms can be used over Tagged Huffman.

7.2.3 End-Tagged Dense Code

End-Tagged Dense Code (ETDC) [BINP03, BFNP07] is also a word-based byte-oriented compression technique, where the first bit of each byte is reserved to flag whether the byte is the last one of its codeword. Since the flag bit signals the last byte of the codeword instead of the first one, this is enough to ensure that the code is a prefix code regardless of the content of the other 7 bits of each byte, so there is no need at all to use Huffman coding in order to guarantee a prefix code. Therefore, all possible combinations are used over the remaining 7 bits of each byte, producing a *dense* encoding. ETDC is easier to build and faster than TH in both compression and decompression.

As for TH, the tag bit in ETDC permits Boyer-Moore-type searching by simply compressing the pattern and then running the string matching algorithm. However, since ETDC is not a *suffix free* code (a codeword can be the suffix of another codeword), each time a matching of the whole pattern occurs in the text, it is mandatory to check whether the byte preceding the first matched byte is the last byte of a codeword or a part of the current codeword, which is longer than the pattern. It is also possible to start decompression at any point of the compressed text, because the flag bit gives ETDC the self-synchronization property: one can easily determine the codeword boundaries.

In general, ETDC can be defined over symbols of b bits, although in this thesis we focus on the byte-oriented version where $b = 8$. Given source symbols with decreasing probabilities $\{p_i\}_{0 \leq i < V}$, with V being the size of the vocabulary, the corresponding codeword using the ETDC is formed by a sequence of symbols of b bits, all of them representing digits in base 2^{b-1} (that is, from 0 to $2^{b-1}-1$), except the last one which has a value between 2^{b-1} and $2^b - 1$, and the assignment is done sequentially.

The code assigned to a word depends on the rank of that word in the sorted vocabulary (which is sorted in decreasing order of frequencies), not on its actual frequency. As a result, only the sorted vocabulary must be stored with the compressed text for the decompressor to rebuild the model. Therefore, the vocabulary will be slightly smaller than in the case of Huffman codes, where some information about the shape of the Huffman tree must be stored (even for canonical Huffman trees).

As it can be seen in Table 7.1, the computation of codes is extremely simple: after sorting the source symbols by decreasing frequency, a sequential assignment of codewords is performed. In addition, simple *encode* and *decode* procedures can be efficiently implemented, since the codeword corresponding to the symbol in position i is obtained as the number x written in base 2^{b-1}, where $x = i - \frac{2^{(b-1)k} - 2^{b-1}}{2^{b-1}-1}$ and $k = \left\lfloor \frac{\log_2\left(2^{b-1} + (2^{b-1}-1)i\right)}{b-1} \right\rfloor$, and adding 2^{b-1} to the last digit.

As we can observe from the code assignment for ETDC, the most significant bit of each byte is used to signal the end of a codeword. Hence, there are 128 different byte values, called *continuers*, that can be used as the first bytes of a codeword (from 0 to 127). Likewise, there are 128 different byte values (from 128 to 255) that

Word rank	codeword assigned	# Bytes	# words
0	**1**0000000	1	2^7
1	**1**0000001	1	
2	**1**0000010	1	
...	...	...	
$2^7 - 1 = 127$	**1**1111111	1	
$2^7 = 128$	**0**0000000 **1**0000000	2	$2^7 2^7$
129	**0**0000000 **1**0000001	2	
130	**0**0000000 **1**0000010	2	
...	...	...	
255	**0**0000000 **1**1111111	2	
256	**0**0000001 **1**0000000	2	
257	**0**0000001 **1**0000001	2	
...	...	...	
$2^7 2^7 + 2^7 - 1 = 16511$	**0**1111111 **1**1111111	2	
$2^7 2^7 + 2^7 = 16512$	**0**0000000 **0**0000000 **1**0000000	3	
16513	**0**0000000 **0**0000000 **1**0000001	3	
...	...	...	

Table 7.1: Code assignment in the byte-oriented End-Tagged Dense Code.

act as the last byte of a codeword (called *stoppers*).

An improved variant[2], (s, c)*-Dense Code (SCDC)* [BFNE03, BFNP07], obtains better compression ratios than ETDC, reaching less than 0.3 percentage points over PH compression ratio, by noticing that a different number of continuers and stoppers might compress better depending on the distribution of frequencies of the words. Hence, two parameters are used, s and c for the number of stoppers and continuers respectively, such that byte values between 0 and $c - 1$ are used as continuers and byte values between c and $c + s - 1 = 2^b - 1$ are used as stoppers. Compression can be optimized by computing the optimal values of c and s for a given corpus.

7.2.4 Restricted Prefix Byte Codes

In *Restricted Prefix Byte Codes (RPBC)* [CM05] the first byte of each codeword completely specifies its length. The encoding scheme is determined by a 4-tuple (v_1, v_2, v_3, v_4) satisfying $v_1 + v_2 + v_3 + v_4 \leq R$. The code has v_1 one-byte codewords, $v_2 R$ two-byte codewords, $v_3 R^2$ three-byte codewords and $v_4 R^3$ four-byte ones. They require $v_1 + v_2 R + v_3 R^2 + v_4 R^3 \geq V$ where R is the radix, typically 256, and V the size of the vocabulary. This method improves the compression ratio of ETDC as it

[2] Implementations of both compression techniques, ETDC and SCDC, can be download from the public site http://vios.dc.fi.udc.es/codes/download.html.

can use 256 different byte values as second, third or fourth bytes of the codeword instead of just 128 as ETDC does. It maintains the efficiency with simple encode and decode procedures (it is also a dense code) but it loses the self-synchronization property. If we seek to a random position in the text, it is not possible to determine the beginning of the current codeword. It is possible to adapt Boyer-Moore searching over text compressed with this technique, but it is slower than searching over text compressed with ETDC.

7.3 Indexing

As we have already mentioned, searching is a very demanded feature when dealing with Text Databases. A search can be solved in two different ways: in a sequential or in an indexed way.

On one hand, a *sequential* search does not require any extra structure or preprocessing of the text, but the whole text must be scanned. Searching over compressed text can be more efficiently performed than over plain text. However, it is still a sequential process and time complexity is proportional to the size of the compressed text.

On the other hand, an *indexed* search requires an extra structure built over the text, that is, an *index*, such that the occurrences of the searched pattern can be located without examining the whole text. Indexed searches are generally used for large texts, where a sequential scan of the text becomes prohibitive, provided there is enough space for constructing and maintaining an index.

Inverted indexes [BYRN99, WMB99] and suffix arrays [MM93] are the best known examples of classical indexes. We now describe these techniques in some detail. We finish this section by introducing the recent revolutionary concept of *self-index*, which is an index that contains a implicit representation of the text, such that it can efficiently search and reproduce any portion of the original text without explicitly storing it.

7.3.1 Inverted Index

An *inverted index* is a data structure built over a Text Database that permits to efficiently locate all the positions where a *search term* appears. It keeps a vocabulary of terms and maps each term (usually a word) to the part of the document where it occurs: it stores a *list of occurrences* that keeps the *positions* where the term appears.

The size of the index can vary depending on the *granularity* used [WMB99], which determines the accuracy to which the index identifies the location of a term. Hence, the length of the list of occurrences may vary. A coarse-grained index (e.g., if the index only tells the block where a term appears) stores a much smaller list of occurrences than indexes whose granularity is fine (e.g., if the index tells the exact positions for each term). Using coarse granularity increases the possibilities of maintaining the whole index in main memory, improving some searches [WMB99, BYRN99]. However, if the exact location of an occurrence is required, a sequential scan of the block must be performed.

Depending on the granularity of the index built over a document collection we can distinguish:

- *Word-addressing index* (word level): it stores the documents identifiers and offsets inside those documents for all the occurrences of all the terms in the vocabulary. Therefore, it is the most space demanding index.

- *Document index* (document level): it only stores the identifiers of the documents where a term occurs. Hence, the exact locations of the terms must be sequentially searched for inside those documents.

- *Block addressing index* (block level) [NMN$^+$00]: it stores lists of occurrences that point to *blocks*, and a block can hold several documents or portions of a single long document. Hence, all searches require inspecting all the text in those pointed blocks in order to know where the search pattern appears. There is a space-time trade-off regarding the block size. Block addressing indexes take special advantage of compression. Since a compressed document requires less space, more documents can be held in the same block. This reduces considerably the size of the inverted index.

Searching for phrase patterns (finding a sequence of words in the text) involves obtaining the list for all the words that compose the pattern, and then intersect those lists. There are several algorithms to efficiently intersect inverted lists, such as a *merge*-type algorithm or a *set-versus-set* algorithm (based on searching for the elements of the smallest list over the longest one, typically using either binary or exponential search). For word-addressing indexes, it is necessary to check if the positions stored in the lists correspond to contiguous positions in the document collection. In the case of document or block addressing indexes, the index can be used to select those candidate documents/blocks where both words appear, and then a sequential search must be performed to check if the words appear together in the text.

7.3.1.1 Compressed inverted indexes

Compression has been used along with *inverted indexes* with good results [NMN$^+$00, ZMR98]. Using compression along with block addressing indexes usually improves their performance. The index size is reduced, since the compressed text size is smaller and thus the number of documents that can be held in a block increases.

Moreover, if the text is compressed with a technique that allows direct searching for words in the compressed text, searching inside candidate blocks becomes much faster.

On the other hand, compression techniques can also be used to compress the inverted indexes themselves, as suggested in [NMN$^+$00, SWYZ02], achieving very good results. Efficient representations of inverted indexes typically rely on integer compression techniques. An inverted list can be stored as an ascending sequence of integers. Therefore, using the differences between consecutive integers can reduce the space required to represent the list if they are represented with a variable-length encoding [WMB99], for example γ-codes, δ-codes or Rice codes, explained in Section 3.1. More recent proposals [CM07] use byte-aligned codes, which lose little compression and are faster at decoding. There are also hybrid representations [MC07] where the inverted lists of the most frequent words are represented with bitmaps (the i^{th} bit is set if that word occurs in block i), and the remaining lists are represented with differential values. Intersection of compressed inverted lists can be performed using a merge-type algorithm along with the decoding of such lists. Set-versus-set can also be used avoiding the decompression of the whole lists [CM07, ST07, MC07]. This approach requires the storage of sampled values to permit the direct access to the compressed list.

Hence, applying compression to inverted indexes reduces the overall storage and processing overhead associated with large text collections.

7.3.2 Suffix arrays

The *suffix array* was proposed by Manber and Myers [MM93]. It is a basic full-text index, which supports searching for any substring of the text using binary searches.

Let $T[1,n]$ be a text consisting in a sequence of symbols from an alphabet Σ of size σ. The suffix array $SA[1,n]$ of T is a permutation of $[1,n]$ of all the suffixes $T[i,n]$, with $1 \leq i \leq n$. The permutation is defined by means of the lexicographic ordering $\prec$ such that $T[SA[i],n] \prec T[SA[i+1],n]$ for all $1 \leq i < n$, that is, the suffix array contains the starting position of all the suffixes of the sequence T in lexicographic order.

Figure 7.2 gives an example of the suffix array SA built from the text "*cava o cabo na cova*"[3]. A special character \$ which is lower than the rest of the characters from Σ has been included as the last symbol of the sequence to signal the end of the text. The white space is written as an underscore for clarity, and it is lexicographically smaller than the characters "*a*"-"*z*". Array SA is obtained by storing the original positions in the text of the lexicographically sorted suffixes. Below each position of the suffix array SA we have written the text suffix pointed by that position.

[3] The text of the example is written in Galician and means "The corporal digs inside the cave"

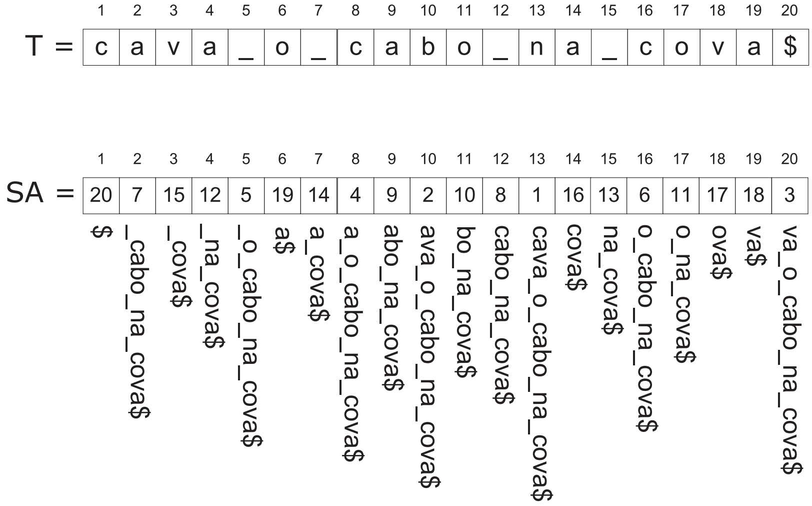


Figure 7.2: Suffix array for the text "*cava_o_cabo_na_cova*$".

The search for a pattern $P[1,m]$ of length m can be performed using a binary search on the suffix array SA. Since the suffixes are sorted following the lexicographic order, all the suffixes that begin with a given substring (the pattern) are grouped together in the suffix array. Then, a binary search is computed in order to obtain the interval $SA[sp,ep]$, being sp the pointer to the first occurrence of the pattern in lexicographic order and ep the last pointer to that zone of occurrences of the pattern. This procedure can be performed in time $O(m \log n)$. Each step of the binary search needs to access the text from the position indicated by the suffix array in order to compare that text with the pattern. Hence, the string $T[SA[i], SA[i]+m-1]$ for some i is compared with $P[1,m]$ to locate its occurrences. Therefore, several accesses to the original text are required during the search.

The main problem of this indexing structure is that it requires a high amount of space. It consists of an array of pointers to positions of the text, and the length of this array is equal to the length of the text. Therefore, it occupies four times the space of the text[4] and it also requires the explicit storage of the text (which is accessed during the searches). There exist several compressed representations of the suffix array [NM07, FGNV09], which exploit in different ways the regularities that appear on the suffix arrays of compressible texts.

We will describe in the next section Sadakane's Compressed Suffix Array (CSA)

[4] Assuming that $n \leq 2^{32}$ and that we implement the suffix array in practice using integers of 32 bits, a 32-bits pointer is needed per each 8-bits character of the text, thus the array of pointers occupies four times the space of the text.

[Sad03], which is a self-indexed structure, that is, in addition to locating any substring of the text, it replaces text, since it contains enough information to efficiently reproduce any text substring.

7.3.3 Self-indexes

Inverted indexes and suffix arrays build the index using space additional to that required by the text. A *self-index* [NM07] is an index that operates in space proportional to the compressed text, replaces it, and provides fast search functionality. It can locate occurrences of patterns in the text, and in addition it contains enough information to extract any text substring in an efficient way.

Classical self-indexes are built over the string of characters that compose the text and permit to search for any substring of the text. They are called *full-text* self-indexes. Some examples of those self-indexed structures are the Compressed Suffix Array [Sad03], the Succinct Suffix Array [MN05], the FM-index [FM05], the Alphabet-Friendly FM-index (AFFM) [FMMN07], or the LZ-index [Nav04]. Most of these full-text self-indexes are based on the Burrows-Wheeler transform (BWT)[5] [BW94] or on the suffix array (see [NM07] for a complete survey).

Those indexes work for any type of text, achieve compression ratios of 40-60%, and can extract any text substring and locate the occurrence positions of a pattern string in a time that depends on the pattern length and the output size, but it is not proportional to the text size (that is, the search process is not sequential). Most can also count the number of occurrences of a pattern string much faster than by locating them.

Compressed Suffix Array

We describe Sadakane's *Compressed Suffix Array (CSA)* [Sad03] since it is one of the best known self-indexes of the literature. The Compressed Suffix Array is a self-index based on the suffix arrays (Section 7.3.2).

The CSA uses function Ψ [GV00] which indicates the position in the suffix array that points to the following suffix of the text, that is, $\Psi(i)$ tells where in SA is the pointer to $T[SA[i]+1]$. Given a suffix array $SA[1,n]$, function $\Psi : [1,n] \rightarrow [1,n]$ is defined so that, for all $1 \leq i \leq n, SA[\Psi(i)] = SA[i]+1$. Since $SA[1] = n$, we fix $SA[\Psi(1)] = 1$ so that Ψ is a permutation.

Figure 7.3 shows function Ψ for the example of the suffix array built over the text "*cava o cabo na cova*". For instance, we can know which position of SA points to the suffix following the suffix "*bo_na_cova*$", which is located at position 11 of SA. Since $SA[\Psi(11)] = SA[11]+1$, we obtain that $SA[17] = SA[11]+1$, that is,

[5] Burrows and Wheeler [BW94] proposed a transformation (BWT) that consists in a reversible permutation of the text characters, generating a more compressible string.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T =	c	a	v	a	_	o	_	c	a	b	o	_	n	a	_	c	o	v	a	$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
SA =	20	7	15	12	5	19	14	4	9	2	10	8	1	16	13	6	11	17	18	3

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Ψ =	13	12	14	15	16	1	3	5	11	20	17	9	10	18	7	2	4	19	6	8

Figure 7.3: Ψ function for the text "*cava_o_cabo_na_cova$*".

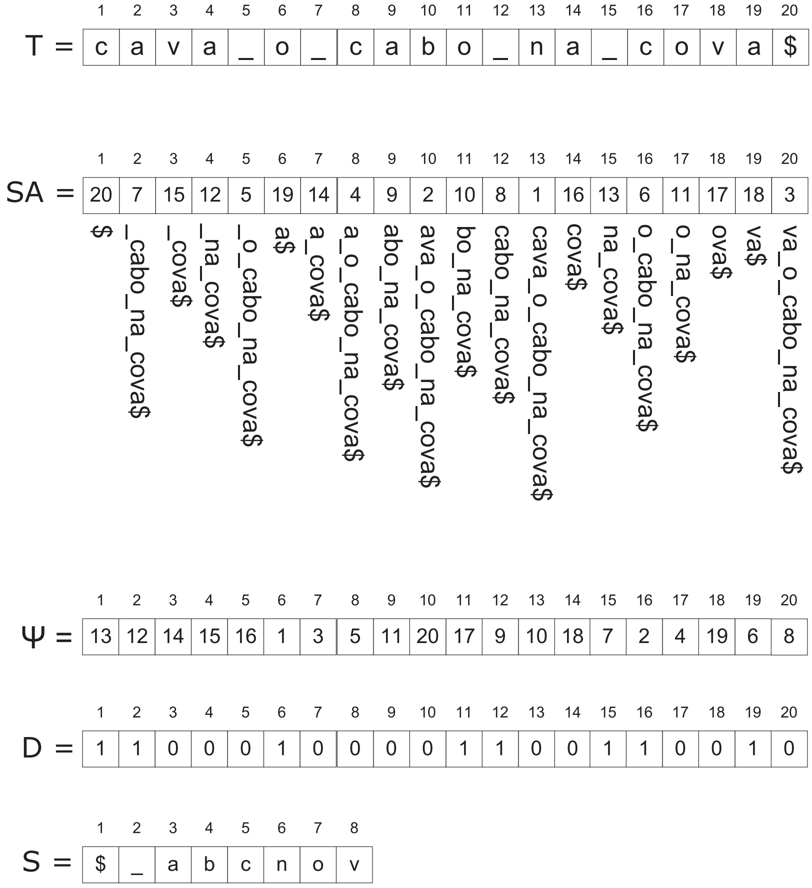
the suffix following "*bo_na_cova$*" is pointed by the 17th element of SA (indeed, it points to "*o_na_cova$*").

Sadakane's CSA represents the suffix array SA and the text T using function Ψ and one extra structure. This additional structure is composed of a bit array D and a sequence S. With this structure it will be possible to know the first character of the suffix pointed by a given position of the suffix array. Sequence S contains the different symbols that appear in the text in lexicographical order. D consists in a bit vector that indicates those positions in the suffix array that point to a suffix that starts with a different character than the suffix pointed by the previous position of the suffix array, that is, $D[i] = 1$ iff $i = 1$ or $T[SA[i]] \neq T[SA[i-1]]$. Therefore, the first character c of the suffix pointed by entry i of the suffix array is $c = S[rank(D, i)]$, where, as discussed in Section 2.3.1, $rank(D, i)$ is computed in constant time using $o(n)$ bits on top of D.

Figure 7.4 illustrates the structures used by Sadakane's CSA for the example "*cava o cabo na cova*". As we have already said, arrays T and SA are replaced by Ψ, D and S, therefore they are shown in the figure only for clarity. For instance, position 7 in the suffix array points to suffix "*a_cova$*", which starts with the same character (a) as the precedent suffix "*a$*". Therefore, $D[7] = 0$. However, position 15 points to suffix "*na_cova$*" which starts with a different character than the suffix pointed by position 14 ("*cova$*"); thus $D[15] = 1$. S is the sorted sequence of the symbols that appear in the text, that is, $S = \{\$, \_, a, b, c, n, o, v\}$.

Now we explain how to use the structures that compose Sadakane's CSA, that is, how the suffix pointed by a given position of the suffix array can be obtained using Ψ, D and S, without having neither SA nor T.

Given a position i of the suffix array, we can know the first character of the suffix using D and S as explained before, that is, $c = S[rank(D, i)]$. To extract



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T =	c	a	v	a	_	o	_	c	a	b	o	_	n	a	_	c	o	v	a	$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
SA =	20	7	15	12	5	19	14	4	9	2	10	8	1	16	13	6	11	17	18	3
	$	_cabo_na_cova$	_cova$	_na_cova$	_o_cabo_na_cova$	a$	a_cova$	a_o_cabo_na_cova$	abo_na_cova$	ava_o_cabo_na_cova$	bo_na_cova$	cabo_na_cova$	cava_o_cabo_na_cova$	cova$	na_cova$	o_cabo_na_cova$	o_na_cova$	ova$	va$	va_o_cabo_na_cova$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Ψ =	13	12	14	15	16	1	3	5	11	20	17	9	10	18	7	2	4	19	6	8

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
D =	1	1	0	0	0	1	0	0	0	0	1	1	0	0	1	1	0	0	1	0

	1	2	3	4	5	6	7	8
S =	$	_	a	b	c	n	o	v

Figure 7.4: Sadakane's CSA for the text "*cava_o_cabo_na_cova*$". Arrays T and SA are shown only for clarity, they are not actually stored.

the next character of the suffix we simply have to extract the first character of the suffix in position $i' \leftarrow \Psi(i)$ using the same process again to obtain $T[SA[i']]$, and so on. In this way, we take advantage of the identity $T[SA[i] + 1] = T[SA[\Psi(i)]]$. Therefore, binary searches performed over the suffix array during searches can be done without the use of the original arrays T and SA.

Notice that using Ψ, D and S we can extract the text given a position of the suffix array, but we do not have any information about the positions of the suffixes in the original text. Hence, if we want to locate the occurrences of a pattern in the text, or we want to extract some portion of the text starting at a given position we still need the suffix array SA. This can be solved by sampling the text at regular intervals and storing the suffix array positions pointing to those sampled text positions.

To obtain a compressed structure, Ψ must be represented in a compact way. Sadakane shows that Ψ is formed by σ increasing subsequences, which can be represented by gap encoding its differential values. Absolute Ψ values at regular intervals are retained to permit fast random access to Ψ (yielding constant time in theory).

Word-based self-indexes

Full-text self-indexes applied over natural language text obtain compression ratios around 60% of the original text size. It would be interesting if these structures could achieve compression close to 30% to 40%, that is, similar to the compression achieved by compressed inverted indexes over compressed text. This reduction can be achieved at the expense of losing some of the searching flexibility of the full-text self-indexes. Recent research has focused on applying a compressed self-index (as developed for general strings) over the sequence of *words* of a natural language text, that is, regarding the words as the basic symbols, such that the benefits of the word-based model are translated to these self-indexes structures. However, the resulting indexes can only search for words and word phrases, just like inverted indexes, but not for any substring of the text, as full-text self-indexes can.

We cannot directly apply traditional full-text self-indexes to natural language words, since it involves some interesting challenges. For instance, the alphabet is now composed of all the words of the text, not just the characters, hence it becomes very large. In addition, some flexible searching might be desired. For example, inverted indexes often permit to find phrases regardless of whether the words are separated by a space, two spaces, a tab, a newline, etc. Moreover, it is customary to apply some filtering on the text words to be searched [BYRN99], such as stemming. It is also usual to disregard stopwords (articles, prepositions, etc.) in the searches. Hence, some preprocessing of the text is required to obtain a sequence of stemmed, significant words, instead of the original sequence of words. However, the original sequence of words must be also represented, so that the self-indexed structure can reproduce the original text.

Some word-based self-indexes have been developed, such as the word-based Compressed Suffix Array (WCSA) and the word-based Succinct Suffix Array (WSSA) [BFN$^+$08, BCF$^+$11]. These self-indexes achieve compression ratios of 35-40% and provide indexed word-based searches, close to many natural language text compressors that do not provide any indexing. Some configurations can even achieve compression ratios around 30% at the expense of worsening search times. They are much faster at counting the number of occurrences of a pattern than a block-addressing inverted index, and they also obtain better relative search performance for locating patterns when the indexes achieve good compression ratios. Their behavior is particulary notorious when searching for phrase patterns, especially for those occurring few times.

7.4 Our goal

As we have already mentioned, it has been demonstrated the benefits of compressing natural language texts using word-based statistical semistatic compression. Not only it achieves extremely competitive compression ratios, but also direct search on the compressed text can be carried out faster than on the original text. Indexing based on inverted lists benefits from compression as well.

Such compression methods assign a variable-length codeword to each different text word. Some coding methods (Plain Huffman and Restricted Prefix Byte Codes) do not clearly mark codeword boundaries, and hence cannot be accessed at random positions nor searched with the fastest text search algorithms. Other coding methods (Tagged Huffman, End-Tagged Dense Code, or (s,c)-Dense Code) do mark codeword boundaries, achieving a self-synchronization property that enables fast searches and random access, in exchange for some loss in compression effectiveness.

In this part of the thesis we propose a new data structure that rearranges the codeword bytes of the compressed text obtained by a byte-oriented word-based encoding scheme, following a wavelet-tree-like strategy. We show that this simple variation obtains a compressed text that is *always* self-synchronized, despite building on encodings which are not. That is, if this new data structure is used, the compressed text can be accessed at any point, even if Plain Huffman coding is used, for example. This encourages using the most space-efficient bytewise encodings with no penalty.

Our aim is not only to obtain direct access to the compressed text, but to also improve search performance. The proposed reorganization provides some *implicit indexing* properties, obtaining a word-based self-indexed structure. That is, with very little extra space, we will be able to search in time that is not proportional to the text length (as sequential search methods) but logarithmic on it (as typical indexed techniques). Indeed, we compare our proposal with *explicit* inverted indexes, and show that it can compete when using the same amount of space.

Chapter 8

Our proposal: Byte-Oriented Codes Wavelet Tree

In this chapter we explain our proposal, Byte-Oriented Codes Wavelet Tree (BOC-WT) in detail. This new data structure aims to represent any natural language text in compact space and self-indexed form, such that not only random access to any position of the text can be achieved, but also supports efficient counting, locating and extracting snippets when searching for a pattern in the text.

The chapter starts by presenting a conceptual description of our proposal in Section 8.1, including some examples to clarify the technique. Later, we detail the different algorithms to compress, decompress and search over the structure in Section 8.2.

8.1 Conceptual description

Our structure, Byte-Oriented Codes Wavelet Tree (BOC-WT), has been designed to be applied to a word-based, byte-oriented semistatic prefix-free compression technique (as all those mentioned in Section 7.2), but it could also be applied to other word-based, byte-oriented prefix-free encoding techniques. Basically, the idea is to rearrange the different bytes of each codeword, placing them in different nodes of a tree that we call wavelet tree for its similarity with the wavelet trees [GGV03] explained in Section 2.3.2. That is, instead of representing the compressed text as a concatenated sequence of codewords (composed of one or more bytes), each one replacing the original word at that position in the text, BOC-WT represents the compressed text as a tree where the different bytes of each codeword are placed at different nodes. Hence, our structure is byte-oriented and the resulting tree is neither balanced nor binary.

The root of the BOC-WT consists of an array of bytes containing the first byte of all the codewords, following the same order as the words in the original text. That is, at position i in the root we place the first byte of the codeword that encodes the i^{th} word in the source text.

The root has as many children as different bytes can be the first byte of a codeword with more than one byte. For instance, in ETDC the root has always 128 children and in RPBC it will typically have $256 - v_1$. The node x in the second level (taking the root as the first level) is also an array of bytes which stores the second byte of those codewords whose first byte is x. Hence each node handles a subset of the text words, in the same order they have in the original text. That is, the byte at position i in node x is the second byte of the i^{th} text codeword that starts with byte x. The same arrangement is used to create the lower levels of the tree. That is, node x has as many children as different second bytes exist in codewords with more than 2 bytes having x as their first byte.

Formally, let us represent the text words[1] as $\langle w_1, w_2 \ldots w_n \rangle$. Lets call cw_i the codeword assigned to word w_i. Notice that two codewords cw_i and cw_j can be the same if the i^{th} and j^{th} words in the text coincide. The bytes of codeword cw_i are denoted as $\langle cw_i^1 \cdots\ cw_i^m \rangle$ where m is the size of codeword cw_i. The root node of the tree is formed by the following sequence of bytes $\langle cw_1^1, cw_2^1, cw_3^1 \cdots\ cw_n^1 \rangle$, that is, the first byte of each codeword $cw_i, 1 \leq i \leq n$ for each of the n words of the text. Hence, the root has as many bytes as words has the text.

As explained, the root has a child for each byte value that can be the first in a codeword. Assume there are r words in the source text encoded by codewords (longer than 1 byte) starting with the byte x: $cw_{i_1} \cdots\ cw_{i_r}$. Therefore, node x will store the sequence $\langle cw_{i_1}^2, cw_{i_2}^2, cw_{i_3}^2 \cdots\ cw_{i_r}^2 \rangle$. Some of those will be the last byte of their codeword, yet others would correspond to codewords with more than two bytes. Therefore, node x would have in turn children as explained before. Assume node xy is a child of node x. It stores the byte sequence $\langle cw_{j_1}^3, cw_{j_2}^3, cw_{j_3}^3 \cdots\ cw_{j_k}^3 \rangle$ of all the third bytes of codewords $cw_{j_1} \cdots\ cw_{j_k}$ starting with xy, in their original text order.

As we can observe from the conceptual description of our proposal, the BOC-WT data structure is not a balanced wavelet tree. Its philosophy differs from the original wavelet tree: the binary wavelet tree divides the vocabulary in two at each level, inducing a binary representation of each element of the vocabulary due to that subdivision, whereas the BOC-WT data structure obtains its shape from the codeword assignment determined by the compression method used[2]. Depending on

[1] We speak of words to simplify the discussion. In practice both words and separators are encoded as atomic entities in word-based compression.

[2] Notice that the Huffman-shaped wavelet tree described in Section 2.3.2 and used in the experimental evaluation of Part I is not a balanced wavelet tree either.

the frequency distribution of the words in the text, some codewords can be longer than others, such that BOC-WT data structure becomes a non-balanced tree. The height of this tree will be equal to the number of bytes of the longest codeword, but usually some of its branches are shorter than this value.

This new data structure follows, somehow, the strategy of the Directly Addressable Codes proposed in the previous part of the thesis. That is, it rearranges the bytes of the codewords in several levels in order to synchronize the codewords, such that it permits direct access to any symbol of the compressed text. In addition to this feature, fast searches over the compressed text are also desirable. Hence, the data structure proposed, BOC-WT, not only rearranges the bytes in levels, but it also separates the bytes of each level in several nodes, obtaining a tree. With this modification, each word of the vocabulary of text is associated with one leaf of the tree and efficient algorithms for searching can be designed such that they just require a simple traversal over the tree.

Example Figure 8.1 shows an example of the proposed data structure, BOC-WT, built from the text `‘LONG TIME AGO IN A GALAXY FAR FAR AWAY’`, where the alphabet is $\Sigma = \{\mathtt{A}, \mathtt{AGO}\ \mathtt{AWAY}, \mathtt{FAR}, \mathtt{GALAXY}, \mathtt{IN}, \mathtt{LONG}, \mathtt{TIME}\}$. After obtaining the codewords for all the words in the text, using any prefix-free byte-oriented encoding, we reorganize the bytes of the codewords in the wavelet tree following the explained arrangement. The first byte of each codeword is in the root node. The next bytes are contained in the corresponding child nodes. For example, the second byte of the codeword assigned to word `‘AWAY’` is the third byte of node $B2$, because it is the third word in the root node whose codeword has b_2 as first byte (the previous bytes of node $B2$ correspond to words `‘TIME’` and `‘IN’`, since the first byte of their codewords is also b_2). In an analogous way, the third byte of its codeword is in node $B2B4$ as its two first codeword bytes are b_2 and b_4. Note that only the shaded byte sequences are stored in the nodes; the text is shown only for clarity.

Direct access to the sequence of words is supported in a similar way as for Directly Addressable Codes due to the rearrangement in levels, that is, using a top-down traversal from the root node until the leaf node and computing the corresponding position at the next level of the tree with *rank* operations. However, due to the byte-oriented tree shape of the data structure, the procedure becomes a little more complex. Let us illustrate the algorithm to extract the original word located at a certain position with an example.

Assume we want to know which is the 6^{th} word of the text. Starting at the root node in Figure 8.1, we read the byte at position 6 of the root node: $Root[6] = b_4$. According to the encoding scheme we can know that, for this code, the codeword is

TEXT: "LONG TIME AGO IN A GALAXY FAR FAR AWAY"

SYMBOL	**FREQ**	**CODE**
FAR	2	b_1
IN	1	b_2 b_5
A	1	b_3 b_1
LONG	1	b_3 b_5
AGO	1	b_4 b_3
TIME	1	b_2 b_1
AWAY	1	b_2 b_4 b_3
GALAXY	1	b_4 b_5 b_2

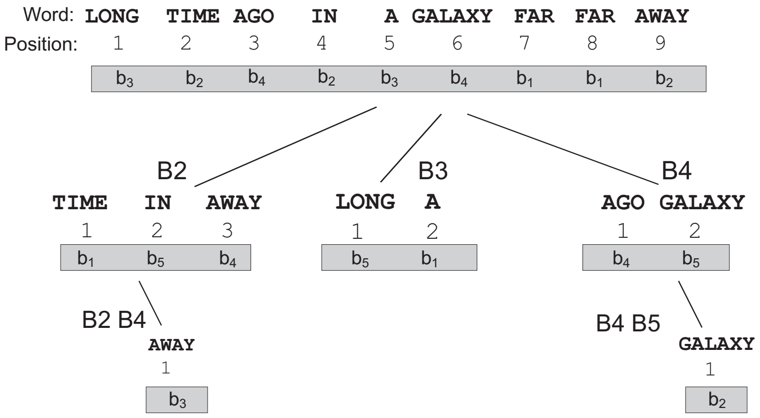

Word: LONG TIME AGO IN A GALAXY FAR FAR AWAY
Position: 1 2 3 4 5 6 7 8 9
b3 b2 b4 b2 b3 b4 b1 b1 b2
B2
TIME IN AWAY
1 2 3
b1 b5 b4
B3
LONG A
1 2
b5 b1
B4
AGO GALAXY
1 2
b4 b5
B2 B4
AWAY
1
b3
B4 B5
GALAXY
1
b2

Figure 8.1: Example of BOC-WT data structure for a short text.

not complete yet, so we move to the second level of the tree in order to continue obtaining the rest of the bytes of the codeword. The second byte must be contained in node $B4$, which is the child node of the root where the second bytes of all codewords starting by byte b_4 are stored. Using a byte *rank* operation $rank_{b_4}(Root, 6) = 2$, we obtain that byte b_4 at position 6 is the second b_4 byte of the root node. This means that the second byte of the codeword starting in the byte at position 6 in the root node will be the 2^{nd} byte of node $B4$. Then, we access to the 2^{nd} position of this node of the second level of the tree, obtaining that $B4[2] = b_5$, therefore b_5 is the second byte of the codeword we are looking for. Again the encoding scheme indicates that the codeword is still not complete, and $rank_{b_5}(B4, 2) = 1$ tells us that the next byte of the codeword will be in the node $B4B5$ at position 1. One level down, we obtain $B4B5[1] = b_2$, and now the obtained sequence $b_4b_5b_2$ is a complete codeword according to the encoding scheme. It corresponds to `‘GALAXY’`, which therefore is the 6^{th} word in the source text.

This process can be used to recover any word at any position of the text. For the complete extraction of a codeword, at most one access to each level of the tree and a rank operation over one of its nodes are needed. Notice that this mechanism gives direct access and random decompression capabilities to encoding methods that do not mark boundaries in the codewords. Independently of the encoding scheme used, with BOC-WT data structure those boundaries become automatically defined since each byte in the root corresponds to a new codeword. Hence, each position of the text can be directly accessed such that random decompression is supported. As we have previously anticipated, this direct access property is obtained due to the rearrangement of the bytes of the codewords in levels, such as the Directly Addressable Codes of Part I of the thesis.

Searching The new data structure BOC-WT does not only provide synchronism to any encoding scheme, supporting direct access and random decompression of the text, but it also improves searches thanks to its tree shape. We illustrate the searching procedure with an example.

If we want to search for the first occurrence of `‘AWAY’` in the example of Figure 8.1, we start by determining its codeword, which is $b_2b_4b_3$ according to the assignment of codewords of the example. Therefore the search will start at the node $B2B4$, which holds all the codewords starting with b_2b_4. In this leaf node we want to find out where the first byte b_3 occurs, as b_3 is the last byte of the codeword sought. Operation $select_{b_3}(B2B4, 1) = 1$ tells us that the first b_3 is at position 1 of node $B2B4$, hence the first occurrence of word `‘AWAY’` is the first of all words with codewords starting with b_2b_4, thus in the parent node $B2$ the first occurrence of byte b_4 will be the one encoding the first occurrence of the word `‘AWAY’` in the text. Again, to know where the first byte b_4 is in node $B2$, we perform $select_{b_4}(B2, 1) = 3$. Therefore, word `‘AWAY’` is the third word of the text whose first byte is b_2. Thus, the 3^{rd} byte b_2 of the root node will be the one corresponding to the first byte of our codeword. To know where that 3^{rd} byte b_2 is in the root node, we compute $select_{b_2}(Root, 3) = 9$. Finally the result is that the word `'AWAY'` appears for the first time as the 9^{th} word of the text.

Notice that it would be easy to obtain a snippet of an arbitrary number of words around this occurrence, just by using the explained decompression mechanism to extract all the words surrounding that position. This entails a significant contribution, since it allows backwards and forward decompression from any position of the text regardless of the compression technique used.

Space requirements The sum of the space needed for the byte sequences stored at all nodes of the tree is exactly the same as the size of the compressed text obtained by the compression technique used to build the BOC-WT data structure. Just a rearrangement has taken place. Yet, a minimum of extra space is necessary in order to maintain the tree shape information with a few pointers. Actually, the shape of the tree is determined by the compression technique, so it is not necessary to store those pointers, but only the length of the sequence at each node[3]. In addition, some extra space can be used to support fast rank and select operations over the byte sequences (see Section 2.3.2).

Due to all the properties previously mentioned, this new data structure can be seen as a self-indexed structure: it occupies a space proportional to the compressed text and it can efficiently solve operations such as counting and locating patterns in the text, and also displaying any portion of the text.

8.2 Algorithms

In the previous section we have conceptually presented the new data structure BOC-WT and shown how it is navigated using a small example. In this section, we detail the general algorithms for constructing the tree, accessing to any position of the text and extracting the word located in that position, and searching for patterns in the text represented by the data structure.

8.2.1 Construction of BOC-WT

The construction algorithm makes two passes on the source text. In the first pass we obtain the vocabulary and the model (frequencies), and then assign codewords

[3] Notice that we use a canonical PH, so it is not necessary to store pointers to maintain the shape of the tree and determine the i-th child of a given node in constant time. In the same way, the wavelet trees built using ETDC or RPBC can be navigated without the need of extra pointers due to the dense assignment of codewords, which causes that all the nodes with children are contiguously located in the wavelet tree. If an arbitrary code was used, the use of pointers or bitmaps may be required to determine which node is the i-th child of a given node.

using any prefix-free encoding scheme. In the second pass the source text is processed again and each word is translated into its codeword. Instead of storing those codewords sequentially, as a classical compressor, the codeword bytes are spread along the different nodes in the wavelet tree. The node where a byte of a codeword is stored depends on the previous bytes of that codeword, as explained.

It is possible to precalculate how many nodes will form the tree and the sizes of each node before the second pass starts, as it is determined by the encoding scheme and the frequencies of the words of the vocabulary. Then, the nodes can be allocated according to these sizes and filled with the codeword bytes as the second pass takes place. We maintain an array of markers that point to the current writing position at each node, so that they can be filled sequentially following the order of the words in the text.

Finally, we obtain the BOC-WT representation as the concatenation of the sequences of all the nodes in the wavelet tree, and we add a header with the assignment between the words of the text and their codewords, determined by the compression technique employed. In addition, BOC-WT data structures includes the length of the sequence for all the nodes of the tree and some extra information, if needed, of the shape of the tree. This information depends on the compression method used; if ETDC is the chosen technique, then there is no extra information to maintain, whereas if we reorganize the compressed text of PH, then some extra bytes representing the canonical Huffman tree are needed.

Algorithm 8.1 shows the pseudocode of this procedure, where the input is the source text that we want to represent and the output is the BOC-WT data structure generated.

8.2.2 Random extraction

Operation *display* is vital for a self-indexed structure. Since a plain representation of the original text is not stored, this procedure allows one to decompress portions of the text, starting at any position of the compressed text, or even recover the whole original text.

We first explain how a single word is extracted using the BOC-WT data structure, and in the next section we generalize the algorithm such that longer sequences of the text can be displayed.

To extract a random text word j, we access the j-th byte of the root node sequence to obtain the first byte of its codeword. If the codeword has just one byte, we finish at this point. If the read byte b_i is not the last one of a codeword, we have to go down in the tree to obtain the rest of the bytes. As explained, the next byte of the codeword is stored in the child node Bi, the one corresponding to words with b_i as first byte. All the codewords starting with that byte b_i store their second byte in Bi, so we count the number of occurrences of byte b_i in the root node before

Algorithm 8.1: Construction algorithm of BOC-WT

```
Input: $T$, source text
Output: BOC-WT representing $T$
$voc \leftarrow first\text{-}pass(t)$
$sort(voc)$
$totalNodes \leftarrow calculateNumberNodes()$
forall $node \in totalNodes$ do
    $length[node] \leftarrow calculateSeqLength(node)$
    $wt[node] \leftarrow allocate(length[node])$
    $marker[node] \leftarrow 1$
end
forall $word \in T$ do
    $cw \leftarrow code(word)$
    $currentnode \leftarrow rootnode$
    for $i \leftarrow 1$ to $|cw|$ do
        $j \leftarrow marker[currentnode]$
        $wt[currentnode][j] \leftarrow cw^i$
        $marker[currentnode] \leftarrow j + 1$
        $currentnode \leftarrow child(currentnode, cw^i)$
    end
end
return concatenation of node sequences, vocabulary, and length of node
sequences plus some extra information for the compression technique if
needed
```

Algorithm 8.2: *Display x*

```
Input: $x$, position in the text
Output: $w$, word at position $x$ in the text
$currentnode \leftarrow rootnode$
$c \leftarrow wt[currentnode][x]$
$cw \leftarrow [c]$
while $cw$ is not completed do
    $x \leftarrow rank_c(currentnode, x)$
    $currentnode \leftarrow child(currentnode, c)$
    $c \leftarrow wt[currentnode][x]$
    $cw \leftarrow cw||c$
end
$w \leftarrow decode(cw)$
return $w$
```

position j by using a rank operation, $rank_{b_i}(root, j) = k$. Thus k is the position in the child node Bi of the second byte of the codeword. We repeat this procedure as many times as the length of the codeword, as we show in Algorithm 8.2

We can also decompress backwards or forwards a given position. For instance, if we need to return a snippet, we must obtain the previous or next words around the occurrence of a word, then we can follow the same algorithm starting with the previous or next entries of the position of that occurrence at the root node.

The complexity of this algorithm is $(\ell - 1)$ times the complexity of the rank operation, where ℓ is the length of the codeword. Therefore, its performance depends on the implementation of the rank operation.

8.2.3 Full text retrieval

BOC-WT represents the text in a compact space, therefore, we must be able to recover the original text from its data structures. After loading the vocabulary and the whole structure of the BOC-WT, a full recovery of the text consists in decoding sequentially each entry of the root.

Instead of extracting each word individually, which would require $(\ell - 1)$ rank operations for each word (ℓ being the length of its codeword), we follow a faster procedure that avoids all those rank operations. Since all the nodes of the tree will be processed sequentially, we can gain efficiency if we maintain pointers to the current first unprocessed entry of each node, similarly to the markers used at construction time. Once we obtain the child node where the codeword of the current word continues, we can avoid unnecessary rank operations because the next byte of the codeword will be the next byte to be processed in the corresponding node. Except for this improvement, the procedure is the same as the one explained in the previous subsection and its pseudocode is described in Algorithm 8.3.

Algorithm 8.3: *Full text retrieval x*

```
Output: $T$, original text represented by the BOC-WT data structure
forall $node \in totalNodes$ do
    $marker[node] \leftarrow 1$
end
$T \leftarrow \varepsilon$
for $pos = 1 \ldots length[rootnode]$ do
    $currentnode \leftarrow rootnode$
    $c \leftarrow wt[currentnode][pos]$
    $cw \leftarrow [c]$
    while $cw$ is not completed do
        $currentnode \leftarrow child(currentnode, c)$
        $x \leftarrow marker[currentnode]$
        $c \leftarrow wt[currentnode][x]$
        $marker[currentnode] \leftarrow marker[currentnode] + 1$
        $cw \leftarrow cw || c$
    end
    $T \leftarrow T ||\ decode(cw)$
end
return $T$
```

For the example in Figure 8.1, we will proceed as follows. We first initialize the *marker* array to 1 for all the nodes of the tree, since we have not started the full decompression yet. Then, we extract the word at position 1. We read byte b_3 at position 1 of the root node. Since the codeword is not complete according to the encoding scheme, we must read a second byte in the second level of the tree. Instead of performing a rank operation to compute the position of that second byte at node $B3$, we check the value of $marker[B3]$, which contains 1. Hence, the second byte of the codeword is at position 1 of node $B3$, that is, b_5. Since we obtain the last byte of a complete codeword, we have finished the decompression of that word, thus the first decompressed word is the one with codeword b_3b_5, which is word `‘LONG’`. We update the value of $marker[B3]$ to 2. Then, we continue with the word at position 2 of the root node. We read byte b_2 and therefore we go to node $B2$. Since $marker[B2] = 1$, the second byte of the codeword is at position 1 of $B2$. We obtain the last byte b_1 of a complete codeword (b_2b_1), hence the word is `‘TIME’`. We update the value of $marker[B2]$ to 2. The next word has its first byte at position 3 of the root node, that is, its first byte is b_4. By proceeding in an analogous way as described before, we obtain word `‘AGO’`. The word at position 4 of the text contains its first byte at position 4 of the root node, that is, b_2. Now, instead of using a rank operation, we can know that the second byte of this codeword is at position 2 of node $B2$ because $marker[B2] = 2$. Hence, we save unnecessary rank operations using these markers and the whole text can be efficiently extracted.

Algorithm 8.4: *Count* operation

```
Input: $w$, a word
Output: $n$, number of occurrences of $w$
$cw \leftarrow code(w)$
Let $cw = cw'||c$, being $c$ the last byte
$currentnode \leftarrow node\ corresponding\ to\ code\ cw'$
$n \leftarrow rank_c(currentnode, length[currentnode])$
return $n$
```

Starting the decompression at a random position

It is also possible to display a portion of the text, starting from a random position different from the first position of the text. The algorithm is the same as the one described in this section, which retrieves the whole original text, except for the initialization of the markers. If we do not start the decompression of the text from the beginning, we cannot initialize the markers with the value 1 for each node, they must be initialized with their corresponding values, that are at first unknown. Hence, we start the algorithm with all the markers uninitialized. During the top-down traversal of the tree performed to obtain the codeword of each word, the marker of a node might not contain the value of the next byte to be read. Thus, if the marker is uninitialized, a rank operation is performed to establish that value. If the marker is already initialized, the rank operation is avoided and the value contained in the marker is used. At most t rank operations are performed, being t the total number of nodes of BOC-WT data structure.

8.2.4 Searching

As we have already mentioned, BOC-WT data structure provides some implicit self-indexed properties to the compressed text. Hence, it will allow us to perform some searching operations in a more efficient way than over the plain compressed text.

Counting individual words

If we want to *count* the occurrences of a given word, we can just compute how many times the last byte of the codeword assigned to that word appears in the corresponding leaf node. That leaf node is the one identified by all the bytes of the codeword except the last one.

For instance, if we want to count how many times the word `‘TIME’` appears in the text of the example in Figure 8.1, we first notice that its codeword is b_2b_1. Then, we just count the number of times its last byte b_1 appears at node $B2$ (since the first byte of its codeword is b_2). In an analogous way, to count the occurrences of word `‘GALAXY’`, we obtain its codeword, that is, $b_4b_5b_2$ and count the number

Algorithm 8.5: *Locate j^{th} occurrence of word w* operation

```
Input: $w$, word
Input: $j$, integer
Output: position of the $j$-th occurrence of $w$
$cw \leftarrow code(w)$
Let $cw = cw'||c$, being $c$ the last byte
$currentnode \leftarrow$ node corresponding to code $cw'$
for $i \leftarrow |cw|$ to 1 do
    $j \leftarrow select_{cw^i}(currentnode, j)$
    $currentnode \leftarrow parent(currentnode)$
end
return $j$
```

of times that its last byte b_2 appears at node $B4B5$ (since the first bytes of its codeword are b_4b_5). The pseudocode is presented in Algorithm 8.4.

The main advantage of this procedure consists in the fact that we count the number of times that a byte appears inside a node, instead of the whole text. Generally, the size of these nodes are not large, and the time cost can also be alleviated by the use of structures that support efficient rank operations. Hence, the procedure becomes faster than searching the plain compressed text.

Locating individual words

As explained in the example of Section 8.1, to *locate* all the occurrences of a given word, we start by looking for the last byte of the corresponding codeword cw in the associated leaf node using operation *select*. If the last symbol of the codeword, $cw^{|cw|}$, occurs at position j in the leaf node, then the previous byte $cw^{|cw|-1}$ of that codeword will be the j^{th} one occurring in the parent node. We proceed in the same way up in the tree until reaching the position x of the first byte cw^1 in the root node. Thus x is the position of the first occurrence of the word searched for.

To find all the occurrences of a word we proceed in the same way, yet we can use pointers to the already found positions in the nodes to speed up the select operations (this might be relevant depending on the *select* algorithm used). The basic procedure is shown in Algorithm 8.5.

Counting and locating phrase patterns

It is also possible to search for a *phrase pattern*, that is, a pattern which is composed of several words. We locate all the occurrences of the least frequent word in the root node, and then check if all the first bytes of each codeword of the pattern match with the previous and next entries of the root node. If all the first bytes of the codewords of the pattern match, we verify their complete codewords around the

Algorithm 8.6: *List intersection*

```
Input: $w_1$, word
Input: $w_2$, word
Output: positions of the occurrence of the pattern $w_1 w_2$
$x_1 \leftarrow fullselect(w_1, 1)$
$x_2 \leftarrow fullselect(w_2, 1)$
while $max\{x_1, x_2\} \leq n$ do
    if $x_1 + 1 = x_2$ then report occurrence
    if $x_2 + 1 <= x_2$ then
        $x_1 \leftarrow fullselect(w_1, fullrank(w_1, x_2 + 1) + 1)$
    if $x_1 + 1 > x_2$ then
        $x_2 \leftarrow fullselect(w_2, fullrank(w_2, x_1 + 1) + 1)$
end
return $j$
```

candidate occurrence found by performing the corresponding top-down traversal over the tree until either a byte fails to match the search pattern or we find the complete phrase pattern.

This algorithm describes both the procedure for counting and locating the occurrence of phrase patterns, so both operations are equally time-costly.

In addition to this *native method* for searching phrase-patterns over the BOC-WT, it is interesting to remark that BOC-WT also supports **list intersection** algorithms to search phrases over the compressed text. As we have explained in Section 7.3.1, inverted indexes search for phrase patterns by obtaining the lists associated to the words that compose the pattern, and then intersect those lists. The efficiency of the list intersection is crucial for search engines, and it continues to be an open research problem where new list intersection algorithms are constantly being proposed. These algorithms can be applied over BOC-WT by noticing that we can generate the lists associated to each word on the fly.

As an example, the pseudocode of a merge-type algorithm implemented over BOC-WT is shown in Algorithm 8.6. We denote by $fullselect$ the bottom-up traversal of the BOC-WT that locates the i-th occurrence of a word by performing select operations[4]; and we denote by $fullrank$ the top-down traversal that computes the position at the leaf level of the node associated to a word that corresponds to a given position at the root node, by performing consecutive rank operations.

Notice that the native method we first explained can be considered as a set-versus-set-type algorithm since it searches for the elements of the smallest list over the longest one. However, the algorithm presented has been especially adapted to take advantage of BOC-WT data structures. For instance, it will not be necessary

[4] This $fullselect$ operation is equivalent to the *locate* operation previously described

to make complete top-down traversals over the tree to check an occurrence in the longest list if we detect a false matching at upper levels of the tree. In the next chapter we will experimentally show that our native method outperforms the merge-type list intersection algorithm when searching for phrases over a real text.

Chapter 9

Experimental evaluation

This chapter presents the experimental performance of the new method proposed, BOC-WT, described in the previous chapter.

As already explained, BOC-WT can be built over different word-based byte-oriented compression methods. The new proposed structure rearranges the bytes of the codewords that conform the compressed text in a tree-shaped data structure. In this chapter, we experiment with three well-known word-based compression techniques with different characteristics (Plain Huffman, End-Tagged Dense Code and Restricted Prefix Byte Codes, all of them explained in Section 7.2), and show the searching capabilities achieved by the new structure BOC-WT built over these compression methods on several corpora. We show that BOC-WT versions are much more efficient than their classical counterparts (the sequential version of the compressed text) when searching functionality is required over the compressed text, due to the self-indexing properties that BOC-WT provides.

We also compare our BOC-WT data structure with explicit inverted indexes, when using the same amount of space. More concretely, we use block-addressing compressed inverted indexes, since they are the best choice, as far as we know [NMN$^+$00, ZMR98], when little space is used. Our results demonstrate that using BOC-WT is more convenient than trying to use very space-efficient inverted indexes. In addition to this comparison, we compare the performance of BOC-WT with some self-indexes of the literature.

The chapter is organized as follows: Section 9.1 describes the collections and machines used in the experiments whereas Section 9.2 explains some important implementation details, such as the structures used to compute rank operations over byte arrays. Next sections present the comparison of the technique with the original compression methods (Section 9.3 and Section 9.4) and also the experimental comparison between our proposal and other indexing structures, that is, inverted indexes (Section 9.5) and other self-indexes (Section 9.6).

Table 9.1: Description of the corpora used.

CORPUS	size (bytes)	num. words	voc. size
CR	51,085,545	10,113,143	117,713
ZIFF	185,220,211	40,627,131	237,622
ALL	1,080,720,303	228,707,250	885,630

9.1 Experimental framework

We used a large corpus (ALL), with around 1GiB, created by aggregating the following text collections: AP Newswire 1988 and Ziff Data 1989-1990 (ZIFF) from TREC-2, Congressional Record 1993 (CR) and Financial Times 1991 to 1994 from TREC-4, in addition to the small Calgary corpus[1]. We also used CR and ZIFF corpus individually to have smaller corpora to experiment with. Table 9.1 presents the main characteristics of the corpora used. The first column indicates the name of the corpus, the second its size (in bytes), the third the number of words that compose the corpus, and the fourth the number of different words in the text.

We used the spaceless word model [MNZBY98] to create the vocabulary and model the separators. A separator is the text between two contiguous words, and it must be coded too. In the spaceless word model, if a word is followed by a space, we just encode the word, otherwise both the word and the separator are encoded. Hence, the vocabulary is formed by all the different words and all the different separators, excluding the single white space.

Two different machines have been used for the experiments. In Sections 9.3 and 9.4 we used an isolated Intel®Pentium®-IV 3.00 GHz system (16Kb L1 + 1024Kb L2 cache), with 4 GB dual-channel DDR-400Mhz RAM. It ran Debian GNU/Linux (kernel version 2.4.27). The compiler used was gcc version 3.3.5 and `-O9` compiler optimizations were set. In Sections 9.5 and 9.6 we used an isolated Intel®Xeon®-E5520@2.26GHz with 72GiB-DDR3@800MHz RAM. It ran Ubuntu 9.10 (kernel 2.6.31-19-server), using gcc version 4.4.1 with `-O9 -m32` options. Time results refer to CPU user time.

In Sections 9.5 and 9.6 we analyze the search performance of our technique over ALL and CR corpora. When the experiments are run over ALL corpus, we use 8 sets of 100 test patterns, all of them randomly chosen from the text (discarding stopwords). Four sets were composed of single-word patterns with different frequency ranges: W_a, W_b, W_c and W_d with words occurring respectively

[1] http://www.data-compression.info/Corpora/CalgaryCorpus/

$[1, 100]$,$[101, 1000]$,$[1001, 10000]$, and $[10001, \infty]$ times. The overall number of occurrences for such sets are $5, 679$; $30, 664$; $258, 098$; and $2, 273, 565$ respectively. We also used four sets, P_2, P_4, P_6, and P_8, which consist of phrase-patterns composed of 2, 4, 6, and 8 words respectively. The number of occurrences of such sets are $201, 956$; $4, 415$; 144; and 169 respectively. When running the experiments over CR corpus, two set of patterns are used: one with 100 individual words randomly chosen from the vocabulary, and another set with 1,000 different phrase patterns composed of 4 words. The overall number of occurrences for such sets are $257, 745$ and $330, 441$ respectively.

9.2 Implementation details

We explored different alternatives to implement rank and select operations over byte sequences, due to their importance on the efficiency of the final structure.

A baseline solution is to carry out those operations by brute force, that is, by sequentially counting all the occurrences of the byte we are interested in, from the beginning of the node sequence. This simple option does not require any extra structure. Interestingly enough, it already allows operations count and locate to be carried out more efficiently than in classically compressed files. In both cases we do sequential searches, but with BOC-WT these searches are done over a reduced portion of the file. Likewise, it is possible to access the text at random, even using non-synchronized codes such as PH and RPBC, faster than scanning the file from the beginning.

However, it is possible to drastically improve the performance of rank and select operations at a very moderate extra space cost, by adapting well-known theoretical techniques [FMMN07]. Given a sequence of bytes $B[1, n]$, we use a two-level directory structure, dividing the sequence into sb superblocks and each superblock into b blocks of size $n/(sb * b)$. The first level stores the number of occurrences of each byte from the beginning of the sequence to the start of each superblock. The second level stores the number of occurrences of each byte up to the start of each block from the beginning of the superblock it belongs to. The second-level values cannot be larger than $sb * b$, and hence can be represented with fewer bits.

With this approach, $rank_{b_i}(B, j)$ is obtained by counting the number of occurrences of b_i from the beginning of the last block before j up to the position j, and adding to that the values stored in the corresponding block and superblock for byte b_i. Instead of $O(n)$, this structure answers $rank$ in time $O(n/(sb * b))$.

To compute $select_{b_i}(B, j)$ we binary search for the first value x such that $rank_{b_i}(B, x) = j$. We first binary search the values stored in the superblocks, then those in the blocks inside the right superblock, and finally complete the search with a sequential scanning in the right block. The time is $O(\log sb + \log b + n/(sb * b))$.[2]

[2] We also tried exponential search instead of binary search to improve locate operation but

Table 9.2: Sizes of the byte sequences of the leftmost nodes at levels 0, 1 and 2 of the BOC-WT data structure built using PH and ETDC.

	BOC-WT over PH	BOC-WT over ETDC
Level 0	228,707,250	228,707,250
Level 1	834,670	22,245,391
Level 2	65,536	77,799

An interesting property is that this structure is parameterizable. That is, there is a space/time trade-off associated to parameters sb and b. The shorter the blocks, the faster the sequential counting of occurrences of byte b_i. In addition, we can speed up select operations by storing the result obtained for the last query. Since it is very frequent to perform several select operations for the same byte value, for instance when finding all the occurrences of a word, this stored value can be used when the previous occurrence of the byte value is located in the same block than the sought one. Hence, instead of searching from the first position of the block, we can start the sequential search from the position of the previous occurrence.

With this solution we obtain better overall performance in practice than using other alternatives to compute rank and select over arbitrary sequences, such as those explained in Section 2.3.2. As a proof of concept, we ran some experiments over 6 byte sequences extracted from two BOC-WT data structures built over a real text. We took the ALL corpus, described in Section 9.1, and built the BOC-WT data structure using Plain Huffman (PH) and End-Tagged Dense Code (ETDC) encodings. In order to study the isolated behavior of the rank/select structure over the byte sequences at the nodes of each BOC-WT, we extracted one byte sequence from the first three levels of their trees (more concretely, we extracted the byte sequence from the leftmost node at each level). Table 9.2 shows the sizes (in bytes) of those byte sequences. Figure 9.1 shows the frequency distribution of all the byte values on those byte sequences. The byte sequence at level 0 for the BOC-WT over PH occupies the same as for the BOC-WT over ETDC, since it stores one byte per word of the text. However, the size and frequency distribution of the byte sequences at level 1 are significantly different since they depend on the encoding scheme used by PH and ETDC respectively. These properties determine the space and time performance of the rank/select structures used.

Figures 9.2 to 9.5 illustrate the behavior of our rank/select structure, denoted by "Ours", against the behavior of the binary wavelet trees used over arbitrary sequences, as explained in Section 2.3.2. We compared two different approaches: a balanced binary wavelet tree using Raman *et al.* solution (see Section 2.3.1) for the rank/select operation over the binary sequences at each level, denoted by "binary

results were not significantly better in practice.

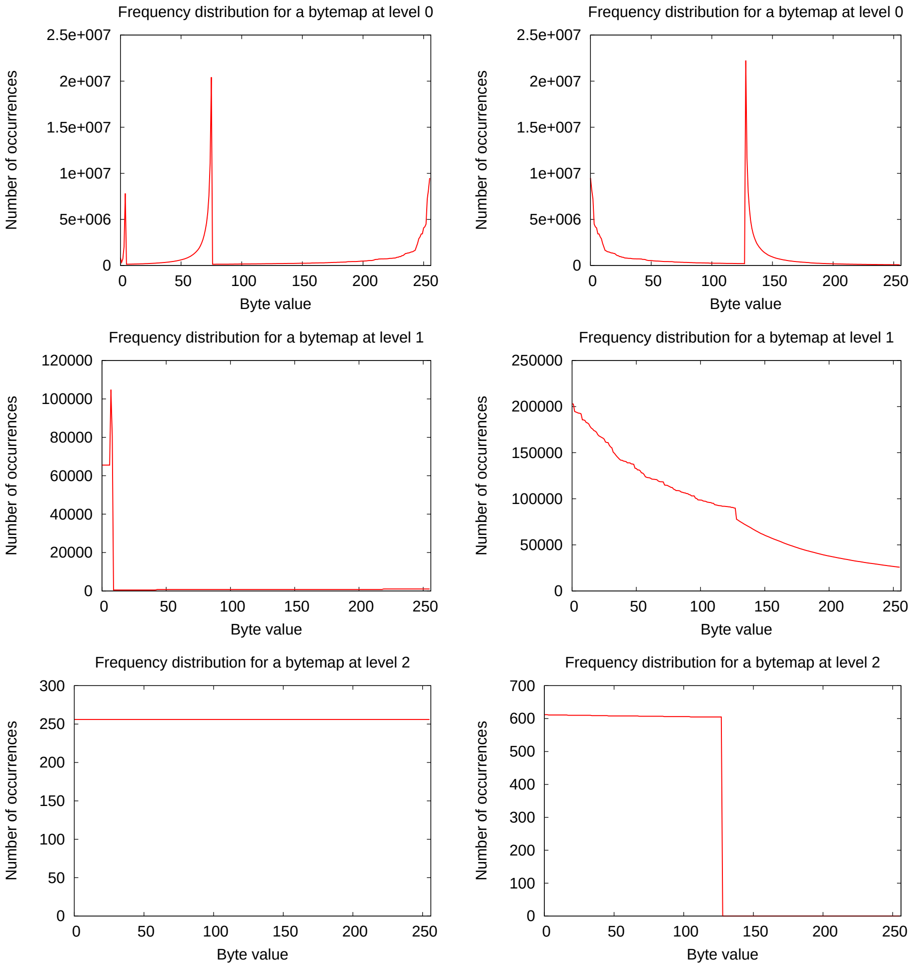
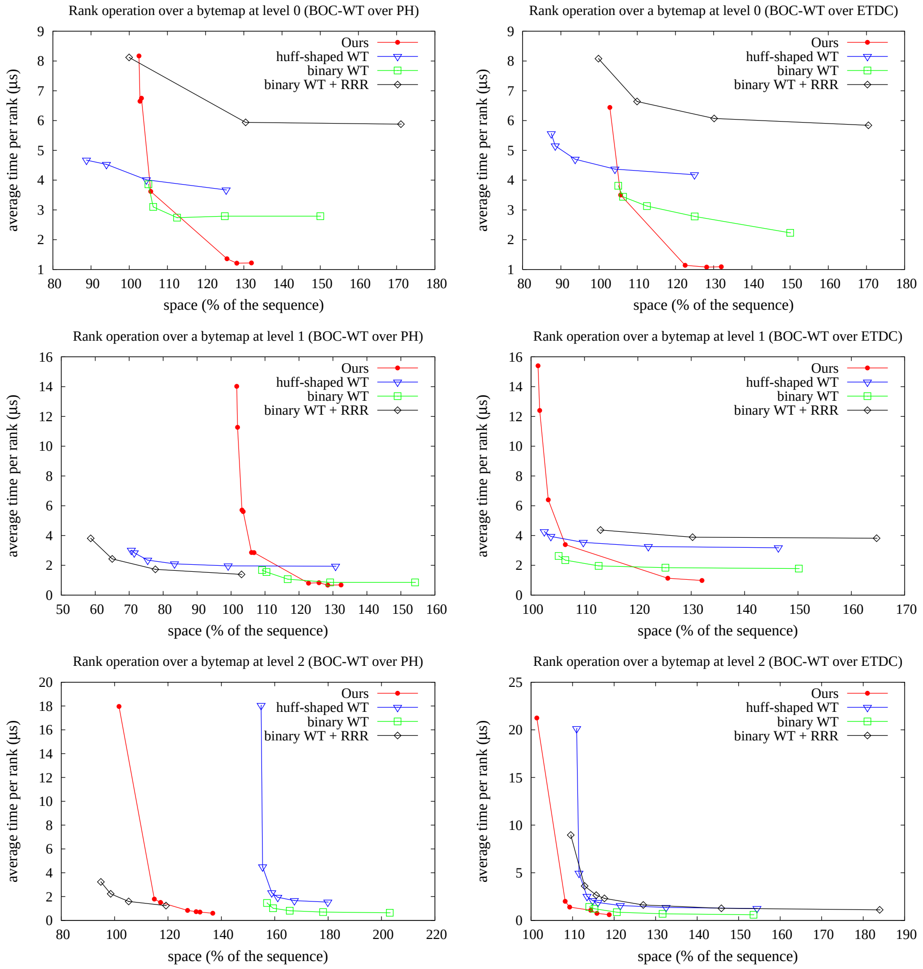


Figure 9.1: Frequency distribution of the byte values in the bytemaps at levels 0 (top), 1 (center) and 2 (bottom) from the BOC-WT built over the ALL corpus using PH (left) and ETDC (right) encodings.


Rank operation over a bytemap at level 0 (BOC-WT over PH)

Rank operation over a bytemap at level 0 (BOC-WT over ETDC)

Rank operation over a bytemap at level 1 (BOC-WT over PH)

Rank operation over a bytemap at level 1 (BOC-WT over ETDC)

Rank operation over a bytemap at level 2 (BOC-WT over PH)

Rank operation over a bytemap at level 2 (BOC-WT over ETDC)

Figure 9.2: Space/time tradeoff for *rank* operation over a byte sequence at level 0 (top), level 1 (center) and level 2 (bottom) of the BOC-WT built over the ALL corpus using PH (left) and ETDC (right) encodings.

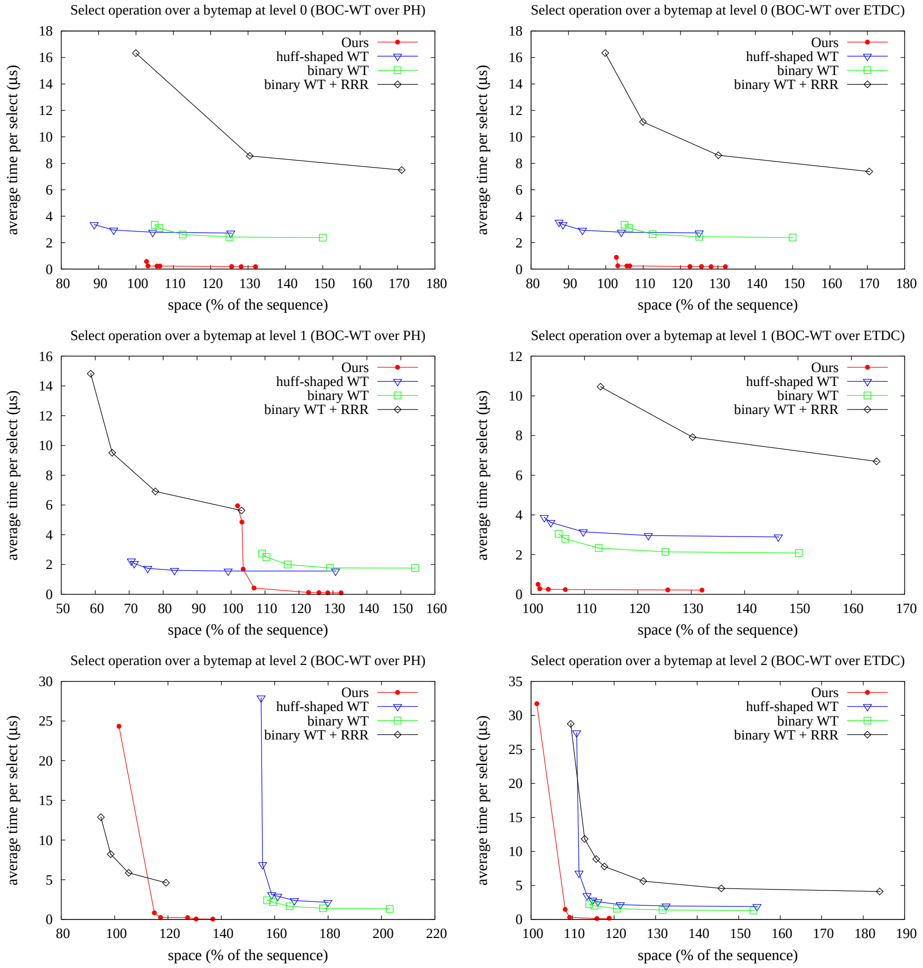


Figure 9.3: Space/time tradeoff for consecutive *select* operations over a byte sequence at level 0 (top), level 1 (center) and level 2 (bottom) of the BOC-WT built over the ALL corpus using PH (left) and ETDC (right) encodings.

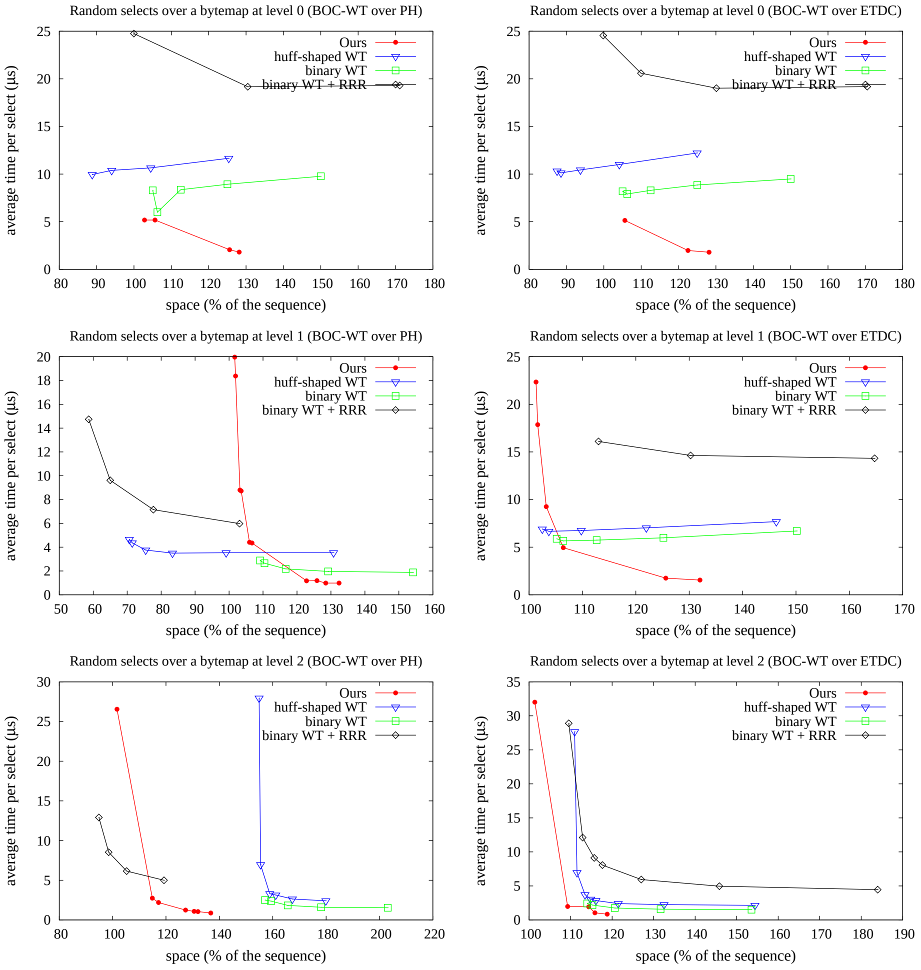


Figure 9.4: Space/time tradeoff for random *select* operations over a byte sequence at level 0 (top), level 1 (center) and level 2 (bottom) of the BOC-WT built over the ALL corpus using PH (left) and ETDC (right) encodings.

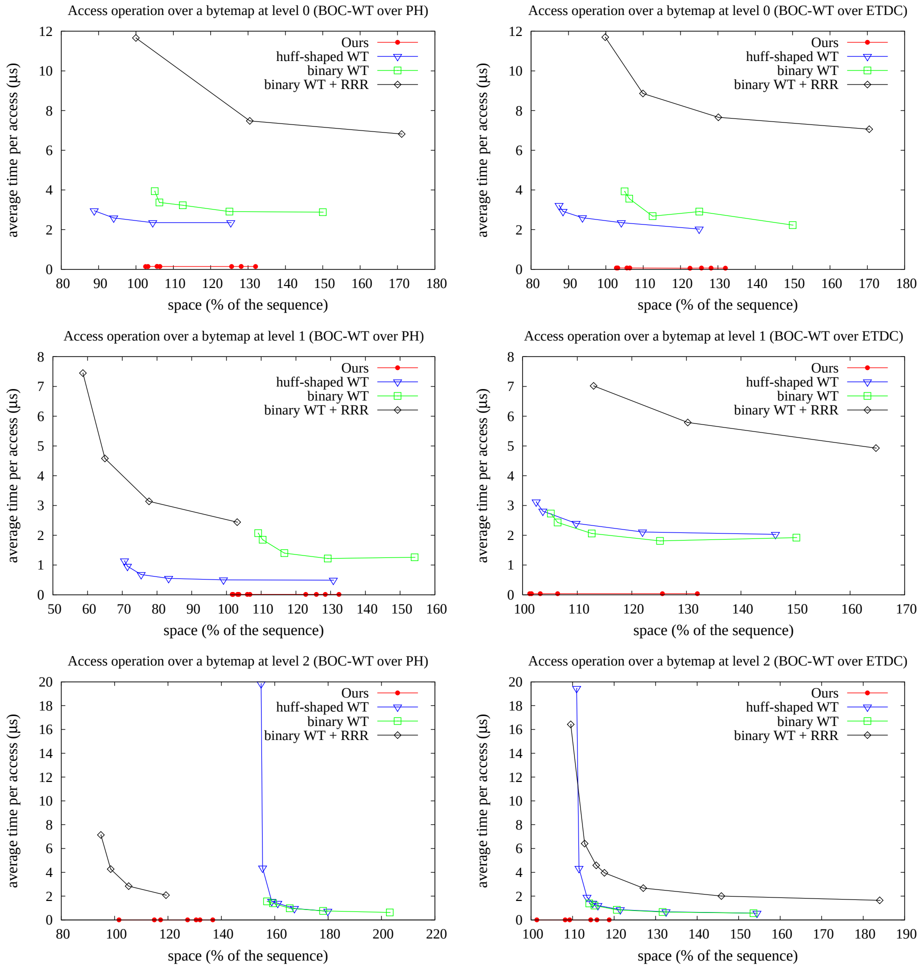


Figure 9.5: Space/time tradeoff for *access* operation over a byte sequence at level 0 (top), level 1 (center) and level 2 (bottom) of the BOC-WT built over the ALL corpus using PH (left) and ETDC (right) encodings.

WT + RRR", a Huffman-shaped wavelet tree, denoted by "huff-shaped WT", and a balanced binary wavelet tree with no bitmap compression, denoted by "binary WT". We do not include Golynski's structure in the comparative since it is not competitive for alphabets of 256 elements [CN08]. We used several configuration of parameters to obtain a space/time tradeoff. The space usage is shown as the space required by the whole representation of the byte sequence and extra structures in main memory as a percentage of the size of the original byte sequence. We computed the average time to perform *rank*, *select* and *access* operations over the byte sequences. More concretely, the *rank* time was measured by computing *rank* operations of random byte values over all the positions of the sequence in random order; *select* time was measured by computing all the occurrences for all the byte values in the sequence; we also performed *randomselect* operations, that is, searching random occurrences of random byte values, and computed the average time; and *access* time was computed by obtaining the byte value at all the positions of the sequence in random order. Times are shown in μs per operation.

We analyze the behavior of *select* operation for two scenarios (searching for consecutive occurrences and searching for random occurrences), since both of them are present when operating over BOC-WT. Note that when we want to locate a word or a phrase, we search consecutively for all the occurrences of the last byte of the codeword in a leaf of the wavelet tree, and for a subset of the occurrences of the rest of the codeword bytes in upper levels of the tree. In these cases, the select operation can take advantage of storing the result of the last select operation in the same bytemap. However, if the algorithm for list intersection described in Section 8.2.4 is used, we will search for different byte values in the same bytemap and for random occurrences of those byte values, hence we must also analyze the behavior of random select operations.

We can observe that the Huffman-shaped wavelet tree obtains a higher compression, occupying less than the original structure, when the frequency distribution of the byte sequence is skewed, as with the byte sequences at level 0 or the byte sequence at level 1 when using PH. However, it obtains the worst space when representing a uniformly distributed sequence, as for the byte sequence at level 2. The balanced binary wavelet tree with RRR behaves oppositely, since it obtains its best performance when the sequence is uniformly distributed. The balanced binary wavelet tree without RRR does not achieve compression. Our structure cannot achieve compression either, since it is built as an auxiliary directory on top of the byte sequence, which is represented in plain form, but we can always adjust the parameters to obtain the desired extra space. This becomes an attractive property, since the space usage of the rank/select structure is not dependent of original byte sequence and its frequency distribution, which may be unknown before building the BOC-WT structure and different among the nodes of the same wavelet tree.

We can also observe from the figures that our representation obtains the best time performance when computing *select* operations for consecutive occurrences and *access* operations. Our representation also obtains good results when computing *rank* or random *select* operations when the space usage greater than 10% of the size of the byte sequence, but the efficiency of these operations degrades when the space is decreased. However, *rank* or random *select* operations are not as frequent as consecutive *selects* and especially as *access* operations. Every bottom-up or top-down traversal over the BOC-WT data structures requires an *access* operation at each level of the tree, hence, the efficiency of this operation is vital in the overall efficiency of the BOC-WT data structure. Finding the occurrence of patterns requires several *select* operation, hence, it is also important its efficiency to the searching performance of the BOC-WT. *Rank* operations are mostly used when decompressing a snippet of the text. However, they are only required once per node accessed in the tree during the top-down traversals for one snippet extraction, since the use of pointers avoids *rank* operations. Moreover, *rank* operations are also used to count the number of occurrences of a word in the text, which is computed over the leaf nodes of the wavelet tree, generally short and uniformly distributed.

In addition, notice that the byte sequence at level 0 contains the 60% of the total number of bytes of the whole BOC-WT data structure. More than 50% of the bytes of this sequence constitute 1 byte-codewords. These 1 byte codewords represent very frequent words, generally stopwords, that are rarely searched. Hence, no *select* operations are performed for those byte values. *Rank* operations are not computed either, since those 1 byte codewords are not continued in lower levels. However, these bytes are commonly accessed when decompressing snippets around the occurrences of nearby significant words. Hence, since the efficiency of the *select* (for consecutive occurrences) and especially *access* operation prevails, we consider that the data structure explained in this section becomes the preferred alternative to represent the byte sequences of the BOC-WT data structure and compute *rank*, *select* and *access* operations in an efficient way.

9.3 Evaluating the compression properties

We measure how the reorganization of codeword bytes induced by our proposal affects the main compression parameters, such as compression ratio and compression and decompression times.

We build the proposed BOC-WT data structure over the compressed texts obtained using three well-known compression techniques explained in Section 7.2. We call WPH, WTDC, and WRPBC the data structures constructed over Plain Huffman (PH), End-Tagged Dense Code (ETDC), and Restricted Prefix Byte Codes (RPBC), respectively.

Table 9.3 shows that compression ratio is essentially not affected. There is a very slight loss of compression (close to 0.01%), due to the storage of the tree shape.

Tables 9.4 and 9.5 show the compression and decompression time obtained using

Table 9.3: Compression ratio (in %) of BOC-WT built using PH, ETDC and RPBC versus their classical counterparts for three different natural language texts.

	PH	ETDC	RPBC	WPH	WTDC	WRPBC
CR	31.06	31.94	31.06	31.06	31.95	31.07
ZIFF	32.88	33.77	32.88	32.88	33.77	32.89
ALL	32.83	33.66	32.85	32.83	33.66	32.85

Table 9.4: Compression time (s).

	PH	ETDC	RPBC	WPH	WTDC	WRPBC
CR	2.886	2.870	2.905	3.025	2.954	2.985
ZIFF	11.033	10.968	11.020	11.469	11.197	11.387
ALL	71.317	71.452	71.614	74.631	73.392	74.811

BOC-WT data structure. The absolute differences in times are similar at decompression and decompression: BOC-WT worsens the time by around 0.1 seconds for CR corpus, 0.4 seconds for ZIFF corpus and 3.5 seconds for ALL corpus. This is due to the fact that with BOC-WT strategy, compression and decompression operate with data that is not sequentially stored in main memory. For each word of the text, a top-down traversal is carried out on the tree, so the benefits of cache and spatial locality are reduced. This is more noticeable at decompression than at compression, since in the latter the overhead of parsing the source text blurs those time differences. Hence, compression time is almost the same (2%-4% worse) as for the sequential compression techniques, hence almost the same time is required to build the BOC-WT data structure from the text than just to compress it. However, there are larger relative differences in decompression time (20%-25% slower).

9.4 Searching and displaying

We show now the efficiency achieved by the BOC-WT technique for pattern searching and random decompression and compare them with the times achieved on a classical encoding of Plain Huffman, End-Tagged Dense Code and Restricted Prefix Byte Code.

We evaluate the performance of the main search operations. We measure user time required to:

- *count* all the occurrences of a pattern (in milliseconds)

Table 9.5: Decompression time (s).

	PH	ETDC	RPBC	WPH	WTDC	WRPBC
CR	0.574	0.582	0.583	0.692	0.697	0.702
ZIFF	2.309	2.254	2.289	2.661	2.692	2.840
ALL	14.191	13.943	14.131	16.978	17.484	17.576

Table 9.6: Load time (in seconds) and internal memory usage for queries (% of corpus size) for the ALL corpus. Load time including on-the-fly creation of rank/select structures for WPH+, WTDC+ and WRPBC+ is shown in parenthesis.

	Load time	Memory usage
	(s)	(%)
PH	0.37	35.13
ETDC	0.39	35.95
RPBC	0.36	35.14
WPH	0.38	35.13
WTDC	0.35	35.96
WRPBC	0.37	35.14
WPH+	0.38 (1.91)	36.11
WTDC+	0.39 (1.93)	36.95
WRPBC+	0.35 (1.91)	36.09

- locate the position of the *first* occurrence (in milliseconds)
- *locate all* the occurrences of a pattern (in seconds)
- retrieve all the *snippets* of 10 words centered around the occurrences of a pattern (in seconds).

We run our experiments over the largest corpus, ALL, and show the average time to search for 100 distinct words randomly chosen from the vocabulary (we remove stopwords, since it makes no sense to search for them). We present the results obtained by the compression methods PH, ETDC, and RPBC; by the BOC-WT data structure implemented without blocks and superblocks (WPH, WTDC, and WRPBC); and also by BOC-WT using blocks of 21,000 bytes and superblocks of 10 blocks with a waste of 1% of extra space to speed up rank and select operations (we denote these alternatives WPH+, WTDC+, and WRPBC+).

Table 9.7: Search performance for the ALL corpus.

	Count (ms)	First (ms)	Locate (s)	Snippets (s)
PH	2605.600	48.861	2.648	7.955
ETDC	1027.400	22.933	0.940	1.144
RPBC	1996.300	41.660	2.009	7.283
WPH	238.500	17.173	0.754	72.068
WTDC	221.900	17.882	0.762	77.845
WRPBC	238.700	17.143	0.773	75.435
WPH+	0.015	0.017	0.123	5.339
WTDC+	0.015	0.014	0.129	6.130
WRPBC+	0.015	0.018	0.125	5.036

Table 9.6 shows the loading time, so that the compressed text becomes ready for querying, and the internal memory usage needed for each method in order to solve those queries. All the alternatives compared maintain the vocabulary of words using a hash table with identical parameters and data structures. As we can see, the loading step requires the same time for the compressors PH, ETDC and RPBC and the BOC-WT data structures built over them without rank support, that is, WPH, WTDC and WRPBC. This is due to the fact that BOC-WT without *rank* structure needs to load the same number of bytes as the compressed text obtained by the respective compressor. In the case of using *rank* structures, it takes more than 1 second to create on the fly the two-level directory of blocks and superblocks (these times are shown in parenthesis for WPH+, WTDC+ and WRPBC+ in Table 9.6). This time is not as important as search times, because this loading is paid only once per session. However, it is prohibitive for one-shot queries. In that case, it is convenient to create the rank structure in disk during construction step, and store it in disk. Hence, as we can observe from the table, it takes practically the same time to load the BOC-WT data structures with rank support than without rank support if the directory of blocks and superblocks is already computed and stored in disk.

Even without using the extra space for the blocks and superblock structures, the use of BOC-WT data structure improves all searching capabilities except for extracting all the snippets, as shown in Table 9.7. This is because snippets require decompressing several codewords around each occurrence, and random decompression is very slow for BOC-WT if one has no extra support for the rank operations used to track down random codewords.

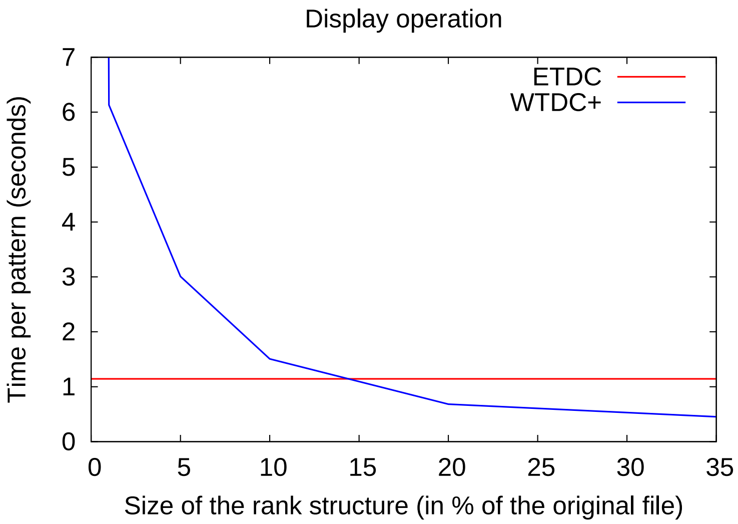


Figure 9.6: Influence of the size of the structure of blocks and superblocks on the performance of the *display* operation, comparing WTDC+ using several sizes of rank structure versus ETDC compressed text.

By just spending 1% extra space in block and superblock data structures to obtain *rank* values faster, all the operations are drastically improved, including the extraction of all the snippets. Only the self-synchronized ETDC is still faster than its corresponding BOC-WT (WTDC+) for extracting snippets. This is because extracting a snippet around a word in a non self-synchronized code implies extra operations to permit the decompression of the previous words, while ETDC, being a self-synchronized code, can easily move backwards in the compressed text.

By raising the extra space allocated to blocks and superblocks, WTDC+ finally takes over ETDC in extracting snippets as well. Figure 9.6 illustrates this behavior. We measure the time to perform the display operation, that is, to locate and extract the snippets around the occurrences of the same 100 randomly chosen patterns used in the experiments of Table 9.7. We create several WTDC+, varying the size of the structure of blocks and superblocks. More concretely, we use always 10 blocks per superblock and vary the number of bytes inside a block with the following values: 20,000; 10,000; 5,000; 2,000; 1,000; 800 and 500, obtaining rank structures occupying 1%, 5%, 10%, 20%, 35%, 40% and 50% respectively. When more extra space is employed for the rank directory, the rank operation becomes more efficient. Therefore, since the extraction operation involves several rank operations, the overall performance of the extraction procedure improves. For this set of patterns, which

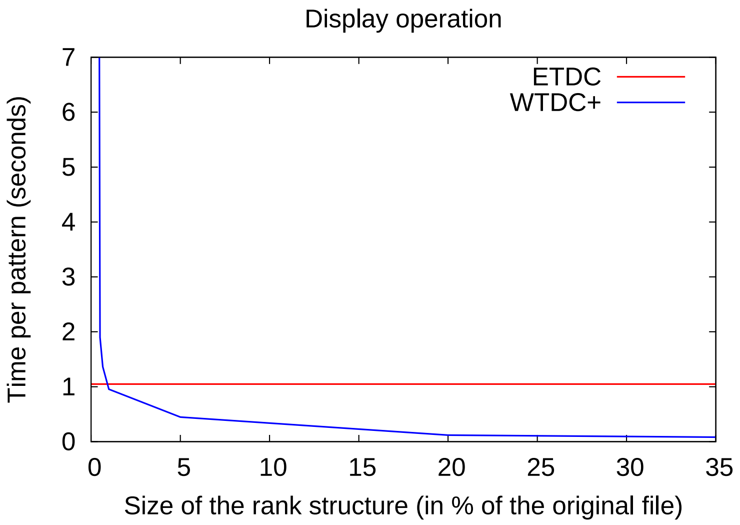


Figure 9.7: Influence of the size of the structure of blocks and superblocks on the performance of the *display* operation, comparing WTDC+ using several sizes of rank structure versus ETDC compressed text, when the words sought are not very frequent.

contains several very frequent words, WTDC+ requires almost 15% of extra space to outperform ETDC times. This is due to the fact that decompression is slower for WTDC+, hence, if the words appear many times in the text, we must decompress many snippets around all those occurrences, worsening the total time.

If we search for less frequent patterns, locating the occurrences is performed much more efficiently by WTDC+, compensating the additional time required to decompress the snippet, and WTDC+ becomes the fastest alternative to display snippets around the occurrences of less frequent words, even without using much extra space for the rank and select structure. Figure 9.7 shows the results for 100 words with frequency up to 50,000. We can observe that WTDC+ obtains better times than ETDC even when very little extra space is used.

It is important to remark that our proposal improves all searching capabilities when a compression technique is not self-synchronized, that is, compared to PH and RPBC. In addition, we observe that WPH and WPH+, which are the alternatives that obtain the best compression ratio, are also the ones that present the best average search efficiency.

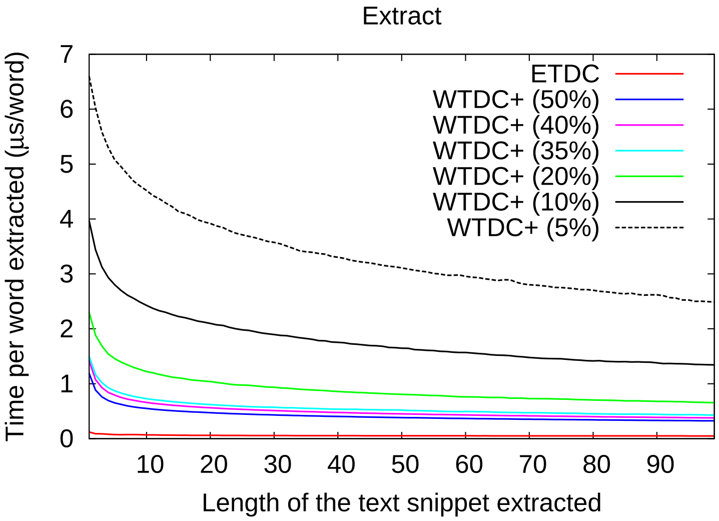


Figure 9.8: Influence of the snippet length on the performance of the *extract* operation for the BOC-WT strategy, comparing WTDC+ using several sizes of rank structure versus ETDC compressed text.

9.4.1 Influence of the snippet length on extract operation

As seen in the previous section, decompression times for BOC-WT are slower than for the original compression technique. Hence, BOC-WT is not as efficient for displaying the snippets around the occurrences of the words searched as for counting or locating those occurrences. As we can see in Table 9.7, WTDC+ is slower than ETDC for the extraction of the snippets.

We will show now how the performance of the *extract* operation improves as the length of the snippet increases. The longer the snippets are, the faster the extraction operation (measured in time per extracted word) becomes. The explanation is trivial if we take into account how the decompression starting at a random position is carried out. Remember that we use one pointer per node to avoid rank operations over the tree. These pointers are uninitialized at first, requiring one rank operation to set their value for a node if needed. Once its value is established, no more rank operations are performed to access a position of that same node. Hence, the first words that are extracted generally require several rank operations to initialize those pointers, since those first words usually have their bytes spread among different nodes of the tree. However, as the following words are being decompressed, rank operations are avoided when the bytes of their codewords share the same node than bytes of codewords previously extracted, as the pointers of those nodes have been already initialized.

Figure 9.8 shows experimentally the described behavior. We compare the extraction time (in μs/word) for BOC-WT over ETDC (WTDC+) considering several snippet lengths. We also illustrate the behavior of WTDC+ with several sizes for the rank extra structure, and compare it with the extraction time over the compressed text with ETDC. We have measured the average time to extract different portions of the ALL corpus, starting at 1,000,000 randomly chosen positions of the compressed text. The lengths of the portions extracted varied from 1 to 100 words. We can observe in the figure that ETDC requires practically constant time to extract any word, independently of the size of the snippet. However, the average time for WTDC is higher if the snippet size is small, since almost every word extracted requires a rank operation at each level of the tree. As more words are extracted, some of these rank operations are avoided and the average time decreases. In addition, we can see in the figure that this time depends on the size of the structure that supports rank operations. We obtain better time results if we spend more extra space to speed up this bytewise operation.

9.4.2 Locating phrase patterns versus list intersection

We now analyze the behavior of the native algorithm for locating the occurrences of phrase patterns, and compare it with the implementation of the merge-type list intersection method in the BOC-WT.

We run our experiments over the largest corpus, ALL, and show the average time to search two different sets of phrase-patterns composed of 2 words. The first set S_1 contains 100 distinct 2-words phrases randomly chosen from the text, where the most frequent word of each phrase appears less than 100,000 times in the text. The second test S_2 contains 100 distinct phrases composed of two words that were randomly chosen from the vocabulary among all the words of frequency f, with $1,000 \leq f \leq 50,000$. These artificially generated phrases of the second set of patterns S_2 do not actually exist in the text. We present the results obtained for both techniques by BOC-WT built over PH (WPH+) using blocks of 20,000 bytes and superblocks of 10 blocks with a waste of 1% of extra space to speed up rank and select operations.

We can observe in Table 9.8 that the best results are obtained by the native algorithm for searching phrases in the BOC-WT. Remember that this algorithm consists in searching for the occurrences of the least frequent word and then check the surrounding positions to know whether there is an occurrence of the phrase or not. This can be very efficiently checked by just comparing the first bytes of the codeword in the first level of the BOC-WT, which permits the fast detection of false matchings. If the first bytes match, then we check the bytes at the second level. Only if all the bytes at each level of the tree coincide, we reach the leaf level of the

Table 9.8: Time results (in ms/pattern) to locate a 2-words phrase for two different algorithms using two sets of patterns S_1 and S_2.

	S_1	S_2
Native phrase searching algorithm	86.07	28.89
Merge-like list intersection algorithm	411.30	100.15

BOC-WT and check if there is an occurrence of the phrase-pattern. On the other hand, the list intersection algorithm performs complete top-down traversals of the BOC-WT, which may be unnecessary.

Notice that the merge-type algorithm of list intersection may be faster than the native method if we search for a phrase composed of two words, where each word appears more frequently in one portion of the text. Thus, we will avoid checking all the occurrences of the least frequent word, as the algorithm may skip several occurrences of that word that appear in one portion of the document by jumping to another portion of the document. However, this scenario is not as probable as searching for phase-patterns composed of words where both appear in the same portion of the document.

9.5 BOC-WT versus inverted indexes

As explained, the reorganization carried out by the BOC-WT data structure brings some (implicit) indexed search capabilities into the compressed file. In this section we compare the search performance of WPH+ with two block-addressing compressed inverted indexes [NMN$^+$00], working completely in main memory[3].

The inverted indexes used are block-grained: they assume that the indexed text is partitioned into blocks of size b, and for each term they keep a list of occurrences that stores all the block-ids in which that term occurs. To reduce the size of the index, the lists of occurrences were compacted using rice codes for the shortest lists and bitmaps for the longest ones. We follow a compression lists strategy [MC07] where the list L of a given word is stored as a bitmap if $|L| > u/8$, being u the number of blocks. No sampling is used. As the posting lists are compressed with variable-length codes, intersection of lists is performed using a *merge*-type algorithm along with the decoding of such lists.

The first compressed inverted index, *II-scdc*, is built over text compressed with (s,c)-Dense Code (*SCDC*), whereas the second index, *II-huff*, is built over text compressed with Huffman. We use SCDC for one of the inverted indexes due to

[3] I would like to thank Antonio Fariña for the indications about the most adequate inverted indexes to compare our proposal to, for the detailed explanations about their implementations and performance, and for the source code provided.

its efficiency at decompression and searches, while achieving a good compression ratio (33.02% for the ALL corpus). For the other inverted index we use *Huffword*, which consists in the well-known bit-oriented Huffman coupled with a word-based modeler [WMB99]. It obtains better compression ratios than *SCDC* (29.22% for ALL corpus), but it is much slower at decompression and searching.

For each of the two alternatives, II-scdc and II-huff, we built several indexes where we varied the block size, which brings us an interesting space/time tradeoff. If the block is large we obtain a smaller index, since it generates smaller inverted lists. However, searches often require the scanning of whole large blocks, and so the index becomes slower. Using small blocks leads to large indexes, but they are faster since the inverted index can discard many blocks during searches and the sequential search inside those blocks is shorter.

To illustrate the behavior of BOC-WT, we compute searching times for the alternative built over PH (WPH+), since it obtains better space and time results. For the experiments of this section and the following ones, the vocabulary is not stored using a hash table, as in the previous sections. We store the vocabulary alphabetically sorted, so that we can obtain the codeword assigned to a word with a binary search over this structure. This solution becomes lighter than using a hash table, and the BOC-WT data structure built over the compressed text of the ALL corpus using PH requires just 33.32% of the original text to solve any query (without any rank and select extra structure). Notice that in Section 9.4, WPH required 35.13% as a hash table was used to maintain the vocabulary. Our method cannot use less than that memory (33.32%) to represent the ALL corpus in a indexed way, whereas the inverted index using Huffman can.

We built several configurations for WPH+ using different sizes for the rank and select structure, so that we can show the space/time tradeoff obtained by the representation. We compare WPH+ with the two inverted indexes, II-scdc and II-huff, over the corpus ALL, using the set of patterns W_a, W_b, W_c, W_d, P_2, P_4, P_6, and P_8 described in Section 9.1. We measure the main memory size occupied by the indexes, and the following search operations:

- *locate*: we measure the time to locate all the occurrences of a pattern inside corpus ALL.

- *display*: we measure the time to display a snippet around all the occurrences of a pattern, which includes the time to locate and extract snippets containing 20 words, starting at an offset 10 words before the occurrence.

Results for both *locate* and *display* operations refer to average time per occurrence (in msec/occurrence). We do not measure *counting* time since it could be solved trivially for word patterns by including the number of occurrences for each word along with the vocabulary (worsening compression ratio by around 0.75 percentage points). BOC-WT counting times for phrase patters are similar to locating them; hence, those counting times can be extracted from the figures for *locate* operation.

Locate time Figures 9.9 and 9.10 show the performance of locating individual words. Figure 9.9 illustrates the behavior of WPH+ and both inverted indexes for scenarios W_a (top) and W_b (bottom), whereas Figure 9.10 shows times results for scenarios W_c (top) and W_d (bottom). We can observe that WPH+ obtains the best results for all the scenarios when little space is used to index the compressed text. This is due to the fact that WPH+ directly jumps to the next occurrence whereas inverted indexes have to scan the text. When little memory is used, the inverted indexes obtain poor results, since a sequential scanning must be performed over large blocks.

WPH+ is slower when locating less frequent words, since it must perform a bottom-up traversal of the tree from the lower level of the tree, and thus several select operations must be computed. For this scenario W_a, inverted indexes overcome WPH+ when the index occupies more than 39%. This scenario is particularly advantageous for II-scdc inverted index: we are searching for less frequent words, which have long codewords assigned, over short blocks of SCDC compressed text. SCDC enables Boyer-Moore-type searching that skips bytes during the search, and since the codewords sought are long, the Boyer-Moore algorithm can skip more bytes. For scenarios W_b, W_c and W_d WPH+ obtains better times than inverted indexes, even when the space used is high.

Figures 9.11 and 9.12 show the performance of locating phrase patterns. Figure 9.11 illustrates the behavior of WPH+ and both inverted indexes for scenarios P_2 (top) and P_4 (bottom), whereas Figure 9.12 shows times results for scenarios P_6 (top) and P_8 (bottom). From the experimental results we can observe that WPH+ can efficiently locate short phrase patterns, of length 2, but its efficiency decreases as the length of the pattern increases. Notice that we are using the average time of locating the patterns measured in millisecond per occurrence. Since long phrase patterns are less frequent than short ones, this average time is greater for long phrase patterns. In addition, when the phrases are long, it is necessary to perform l top-down traversals over the tree, l being the length of the phrase. Even if some more false matchings are detected in the root level, those extra rank operations worsen the average locating time. Inverted indexes become a better choice to search for long phrase patterns for compression ratios above 37%, as it occurred when searching for less frequent patterns: when searching for long phrases, we can skip a bigger number of bytes during the sequential scanning of the blocks. However, WPH+ is always the preferred solution when little space is used.

Extract-snippet time The results here are similar to those for *locate*. As long as we set the indexes to use less space, WPH+ becomes the preferred choice. Time differences in locate times were larger, whereas those for snippet extraction tend

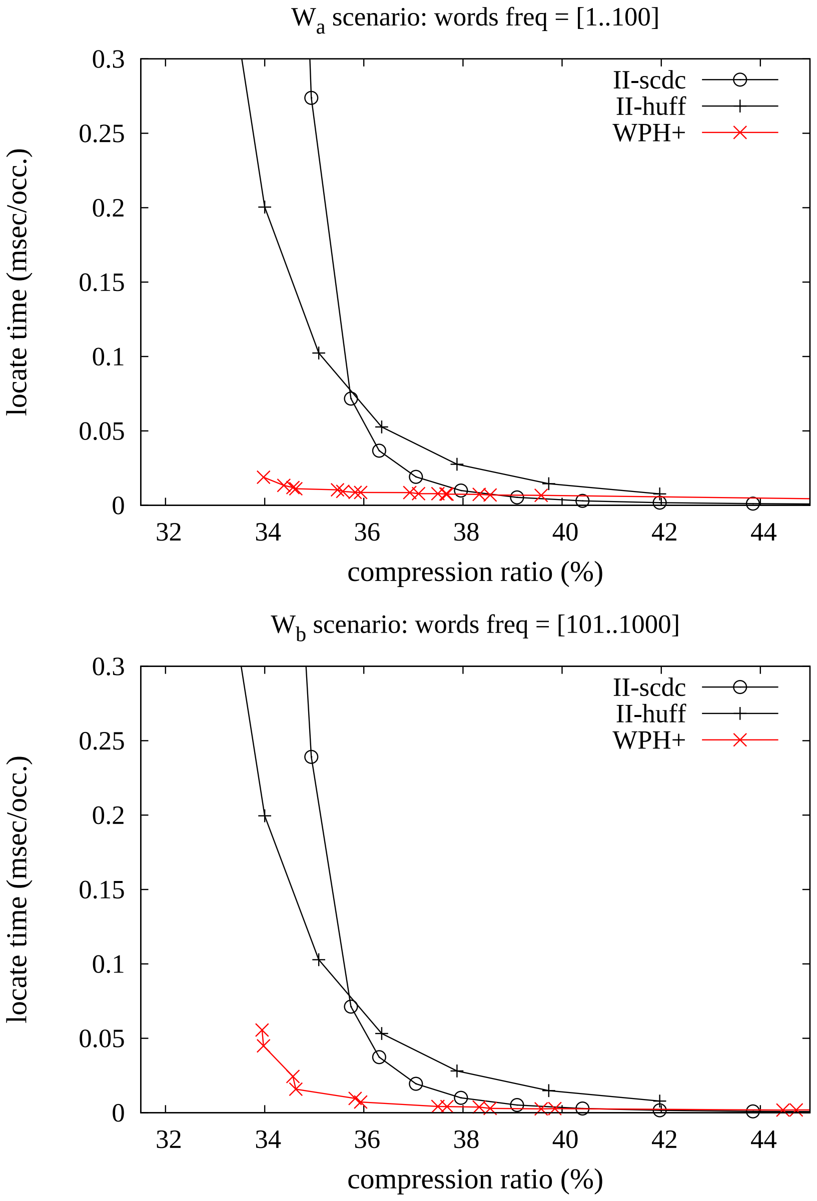
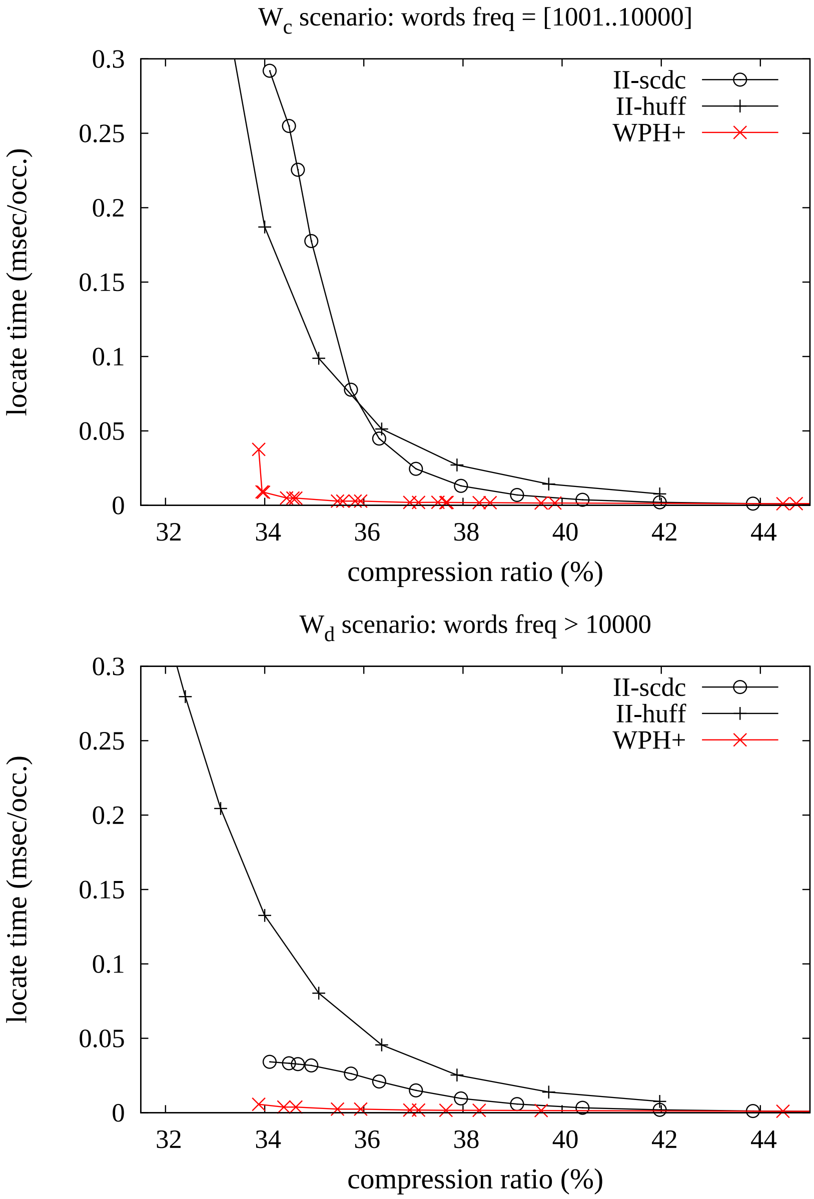


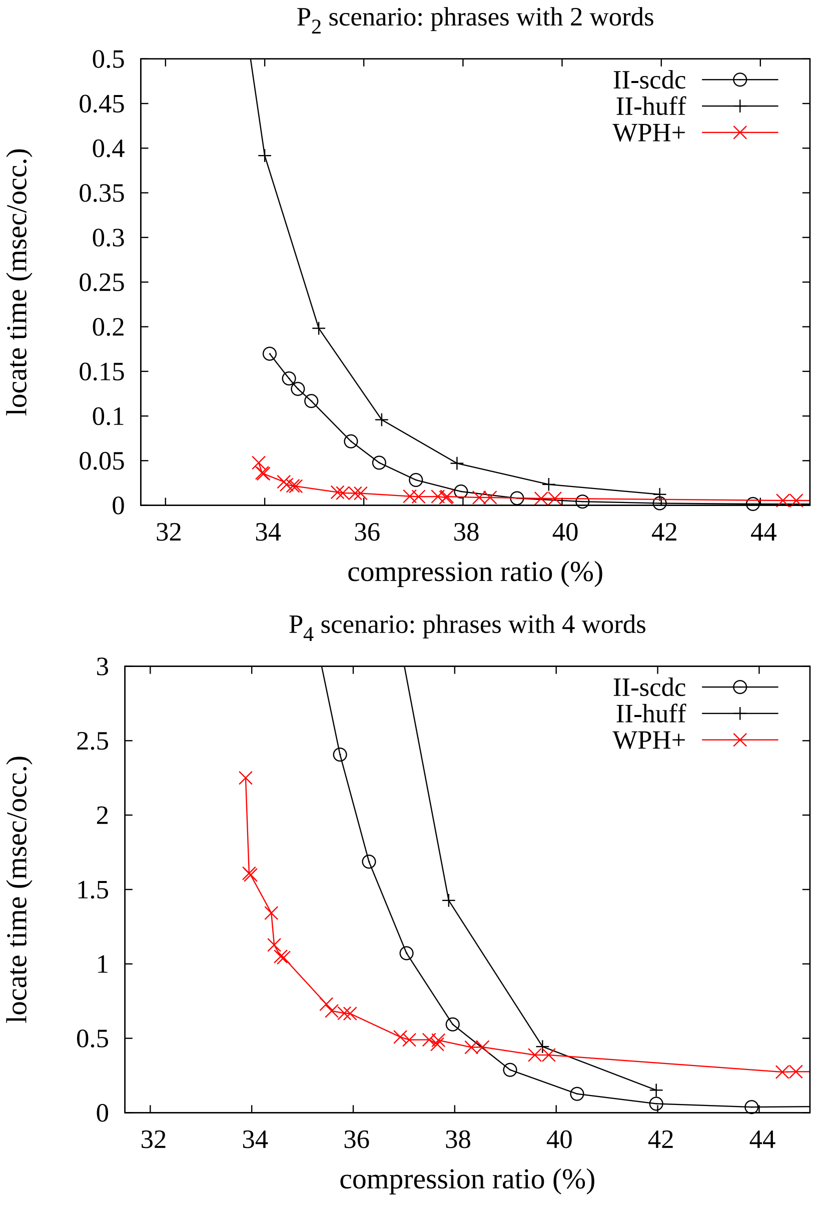
Figure 9.9: Time/space trade-off for *locating* less frequent words with BOC-WT strategy over PH against inverted indexes.



W_c scenario: words freq = [1001..10000]

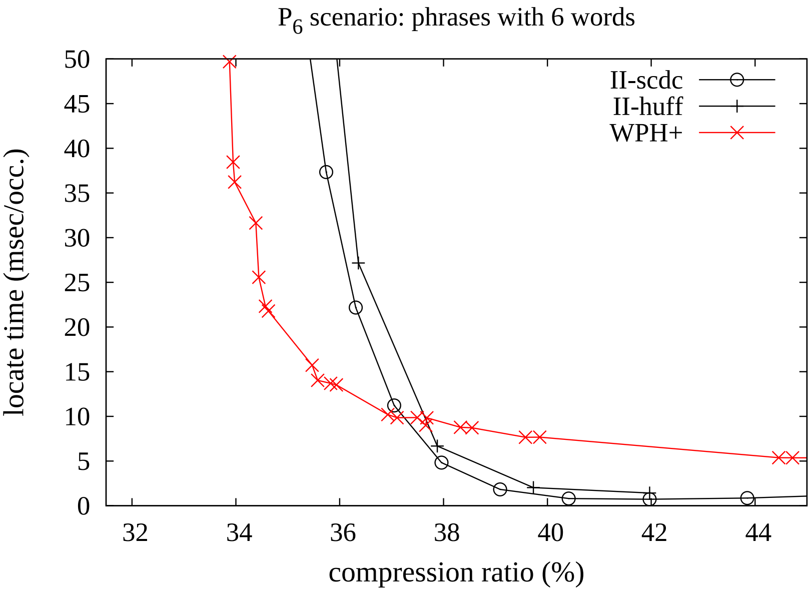
W_d scenario: words freq > 10000

Figure 9.10: Time/space trade-off for *locating* more frequent words with BOC-WT strategy over PH against inverted indexes.


P$_2$ scenario: phrases with 2 words

P$_4$ scenario: phrases with 4 words

Figure 9.11: Time/space trade-off for *locating* short phrase patterns with BOC-WT strategy over PH against inverted indexes.



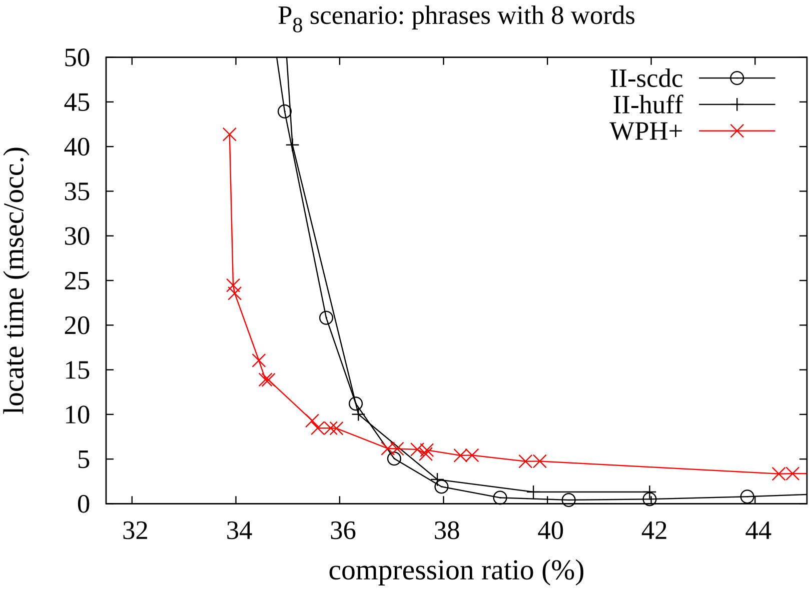


Figure 9.12: Time/space trade-off for *locating* long phrase patterns with BOC-WT strategy over PH against inverted indexes.

to reduce since decompression time is faster for inverted indexes than for WPH+, especially for the inverted index built over SCDC. Remember that the display operation consists in first locating the occurrences of the pattern, where BOC-WT obtains better times than inverted indexes, and then extracting the snippet around each occurrence, which is more efficiently performed by the inverted indexes, since they just have to sequentially decompress the text.

Figures 9.13 to 9.16 show the performance of displaying snippets around the occurrences of several patterns. Figure 9.13 illustrates the behavior of the WPH+ and both inverted indexes for scenarios W_a (top) and W_b (bottom), whereas Figure 9.14 shows times results for scenarios W_c (top) and W_d (bottom). Figure 9.15 illustrates the behavior of WPH+ and both inverted indexes for scenarios P_2 (top) and P_4 (bottom), whereas Figure 9.16 shows times results for scenarios P_6 (top) and P_8 (bottom). Again, WPH+ obtains the best time when the indexes do not occupy much memory, whereas the inverted indexes obtain better results for ratios above 37%.

We remark that our good results essentially owe to the fact that we are not sequentially scanning any significant portion of the file, whereas a block addressing inverted index must sequentially scan (sometimes a significant number of) blocks. As more space is given to both structures, both improve in time but the inverted indexes eventually take over WPH+ (this occurs when both use around 37% of the text's space). If sufficient space is given, the inverted indexes can directly point to occurrences instead of blocks and need no scanning. Yet, as explained in the motivation of this thesis, using little space is very relevant for the current trend of maintaining the index distributed among the main memory of several processors. What our experiments show is that BOC-WT makes better use of the available space when there is not much to spend.

9.6 BOC-WT versus other self-indexes

In this section we compare the results of the BOC-WT strategy, using the most competitive self-index WPH+, with other self-indexes of the literature. We will focus only on those self-indexed structures that support fast searches of words or phrases composed of words and occupy space comparable to our BOC-WT. Therefore, we will not compare our proposal with classical full-text self-indexes that index any pattern of the text (strings of characters instead of words) but require spaces around 40-60% of the original text. Instead, we first compare WPH+ with two binary wavelet trees representing the sequence of words of the text. We then compare it with two word-based versions of two classical self-indexes, the word-based CSA and SSA, and then we compare the performance of WPH+ with the behavior of some full-text self-indexes that index a preprocessed byte-oriented compressed text and search words and phrase patterns.

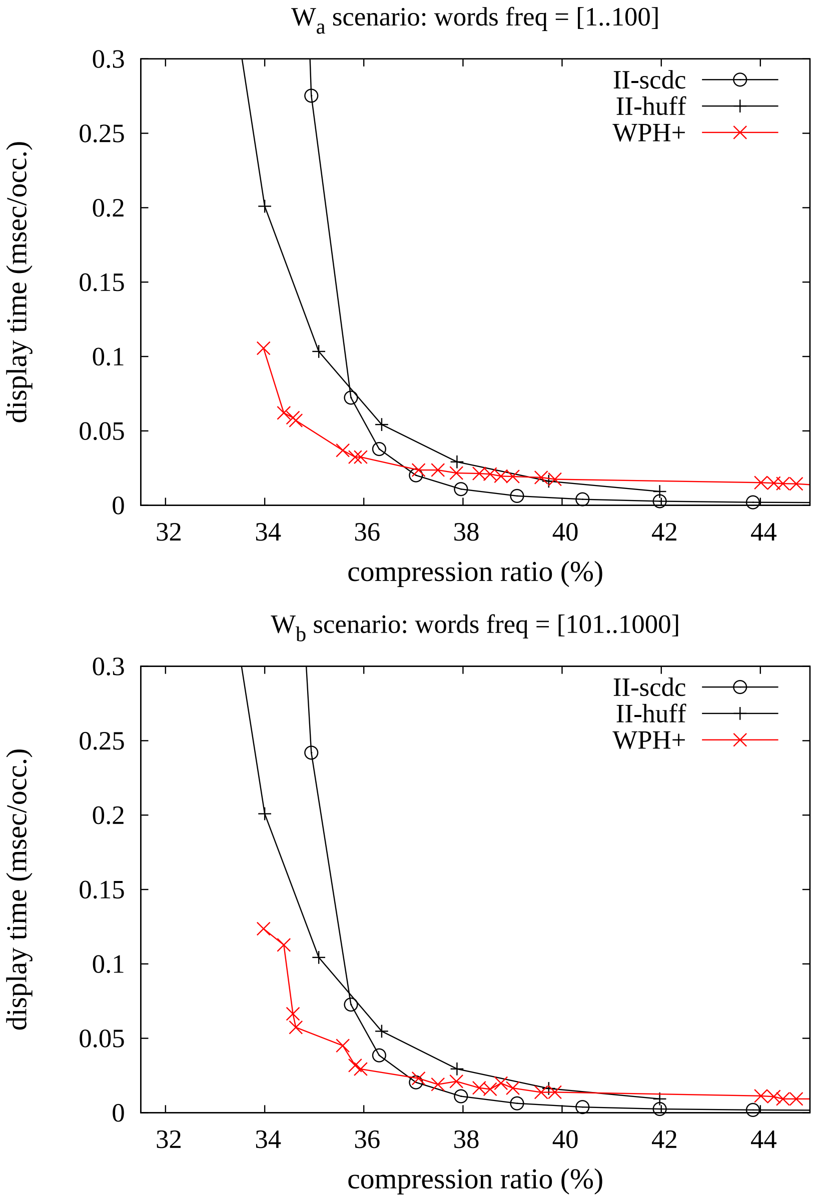


Figure 9.13: Time/space trade-off for *displaying* the occurrences of less frequent words with BOC-WT strategy over PH against inverted indexes.

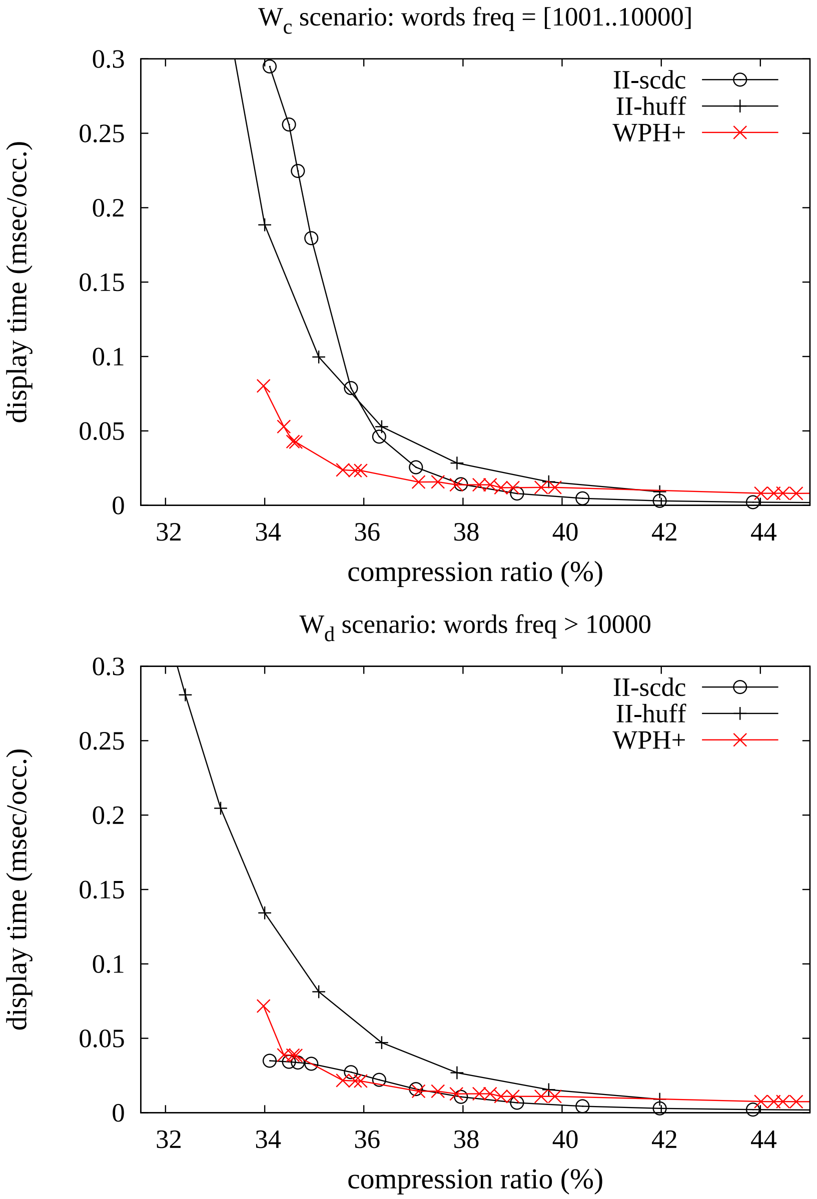


Figure 9.14: Time/space trade-off for *displaying* the occurrences of more frequent words with BOC-WT strategy over PH against inverted indexes.

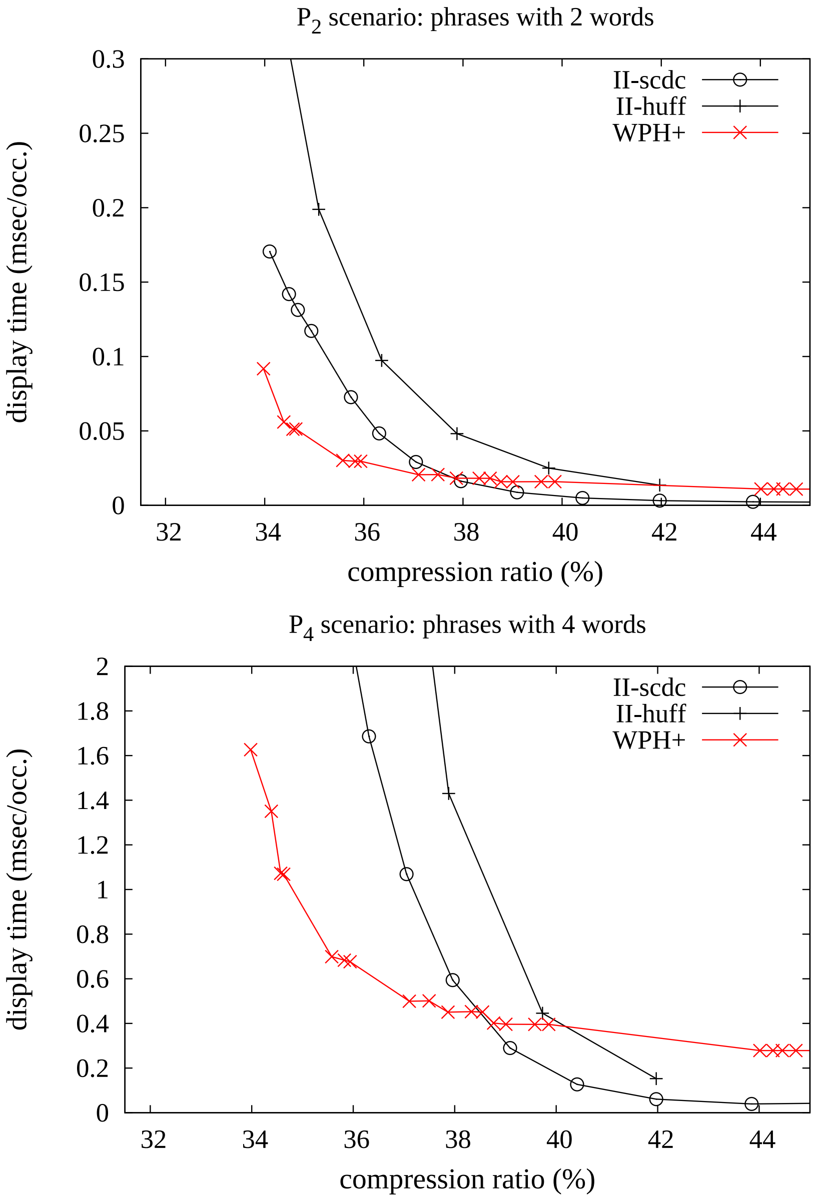


Figure 9.15: Time/space trade-off for *displaying* the occurrences of short phrase patterns with BOC-WT strategy over PH against inverted indexes.

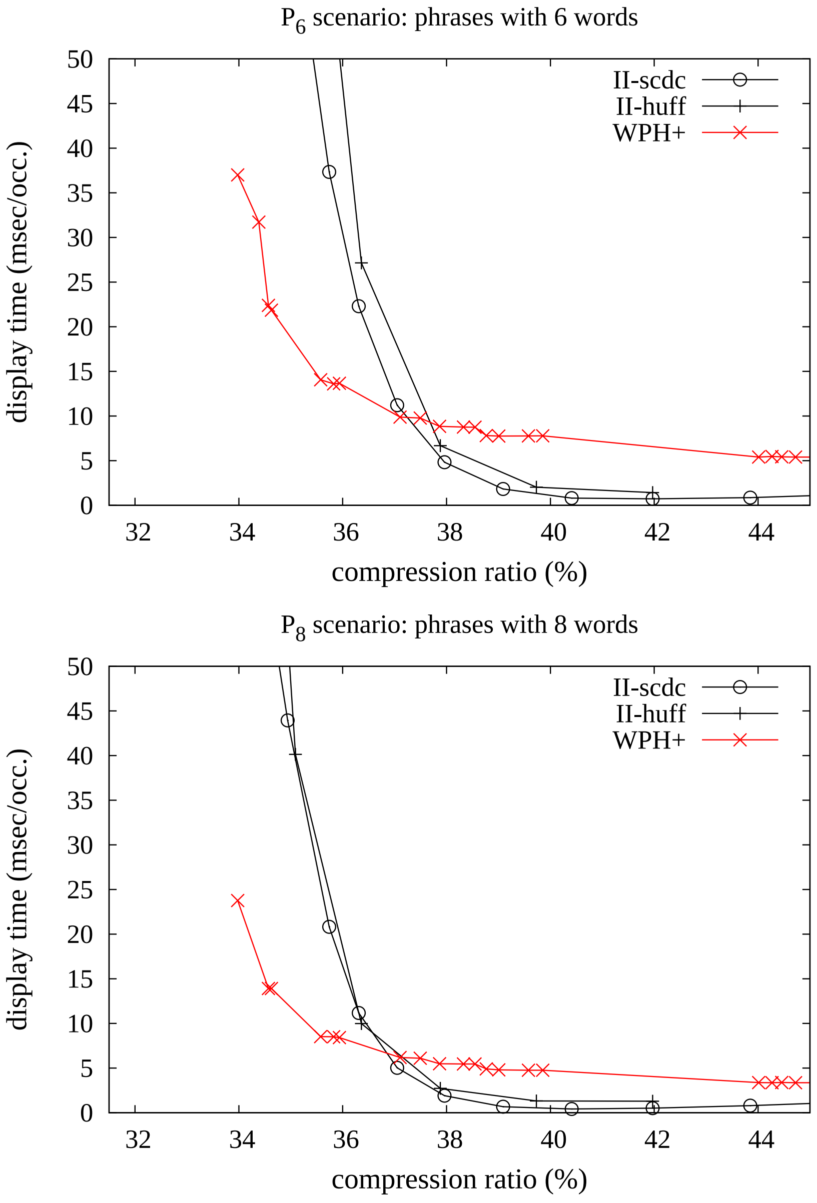


Figure 9.16: Time/space trade-off for *displaying* the occurrences of long phrase patterns with BOC-WT strategy over PH against inverted indexes.

9.6.1 BOC-WT versus word-based Wavelet Trees

In Section 2.3.2 we described how a binary balanced wavelet tree can be used to represent an arbitrary sequence S with an alphabet Σ of size σ, supporting access, rank and select operations. Huffman-shaped wavelet trees have been used to approach zero-order compression of sequences [GGV03, FGNV09, CN08]. We can also achieve compression over a balanced binary wavelet tree by compressing its bitmaps with Raman, Raman and Rao (RRR) technique [RRR02] (explained in Section 2.3.1). We compare our proposal with two binary wavelet trees, one using RRR and other giving the wavelet tree the shape of the Huffman tree, built over the sequence of words and separators that conform the vocabulary of the natural language text. Therefore, we can count the number of occurrences of the words with rank operations, we can locate the occurrences using select operations and extract the original text using access operations.

For the comparison, we create several Huffman-shaped wavelet trees with different sizes, varying the size for the extra structure used to compute fast binary rank and select operations. We also create several balanced binary wavelet trees using RRR with different sizes, varying its sampling parameter. We use the implementations of Francisco Claude available at the Compact Data Structures Library (libcds)[4]. We compare them with the performance of our BOC-WT data structure over PH with different sizes for the rank and select directory, that is, WPH+.

We use the same large text (ALL corpus) and the same first set of patterns than in the previous section, that is, four sets composed of single-word patterns with different frequency ranges: W_a, W_b, W_c and W_d with words occurring respectively $[1, 100]$,$[101, 1000]$,$[1001, 10000]$, and $[10001, \infty]$ times. We measured *locate* time and *display* time (in msec/occurrence).

Figures 9.17 and 9.18 show the performance of locating all the occurrences of several patterns. Figure 9.17 illustrates the behavior of WPH+, the binary Huffman-shaped wavelet tree, called *WTbitHuff*, and the balanced wavelet tree using RRR, called *WTbitRRR*, for scenarios W_a (top) and W_b (bottom), whereas Figure 9.18 shows times results for scenarios W_c (top) and W_d (bottom). Figures 9.19 and 9.20 show the performance of displaying snippets around the occurrences of several patterns. Figure 9.19 illustrates the behavior of WPH+, WTbitHuff and WTbitRRR for scenarios W_a (top) and W_b (bottom), whereas Figure 9.20 shows times results for scenarios W_c (top) and W_d (bottom).

WPH+ is more efficient in both locating and displaying the occurrences of patterns for all the scenarios. The binary Huffman-shaped wavelet tree built over a natural language text requires a high number of levels, hence accessing, counting or locating the symbols of the sequence become very slow operations. Moreover, since the alphabet is so large, around 885000 words, the Huffman-shaped wavelet tree has a large number of nodes and requires a large number of pointers to maintain the

[4] http://libcds.recoded.cl/

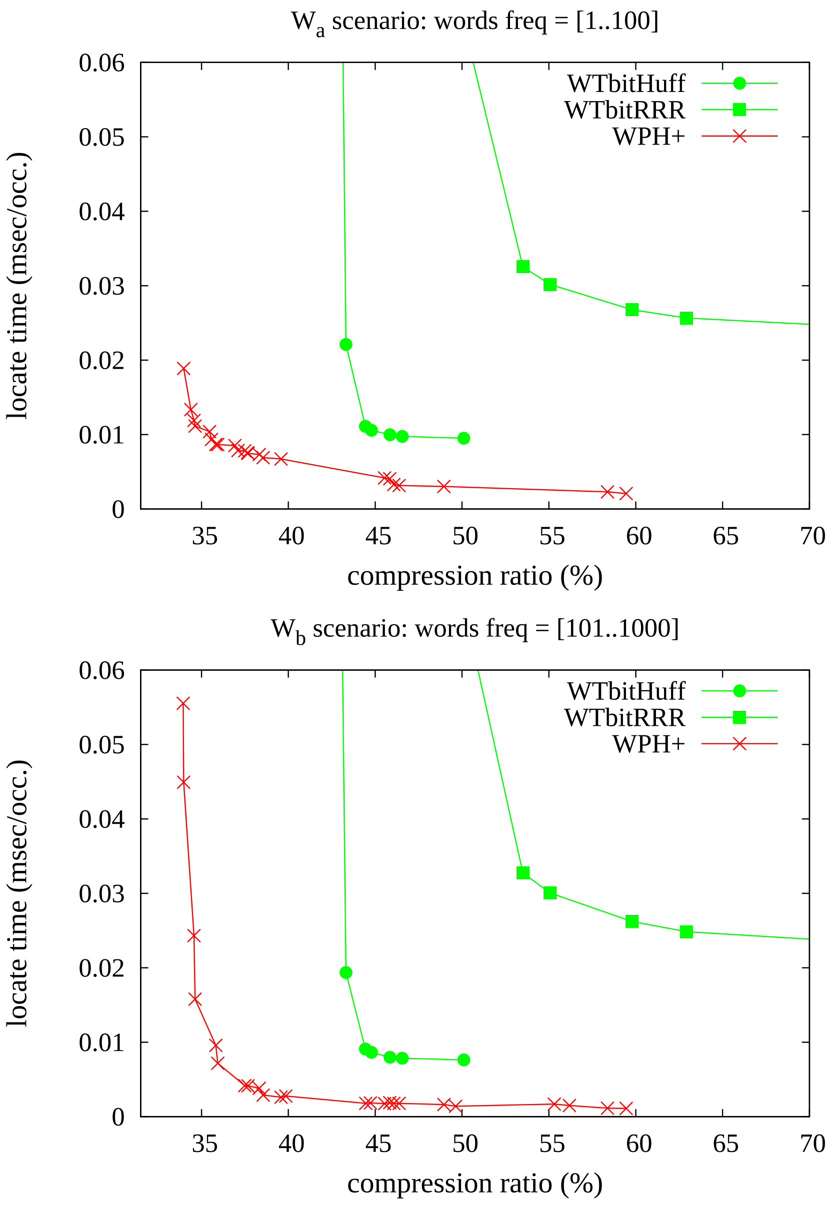


Figure 9.17: Time/space trade-off for *locating* less frequent words with BOC-WT strategy over PH against a word-based Huffman-shaped wavelet tree and a balanced binary wavelet tree using RRR.

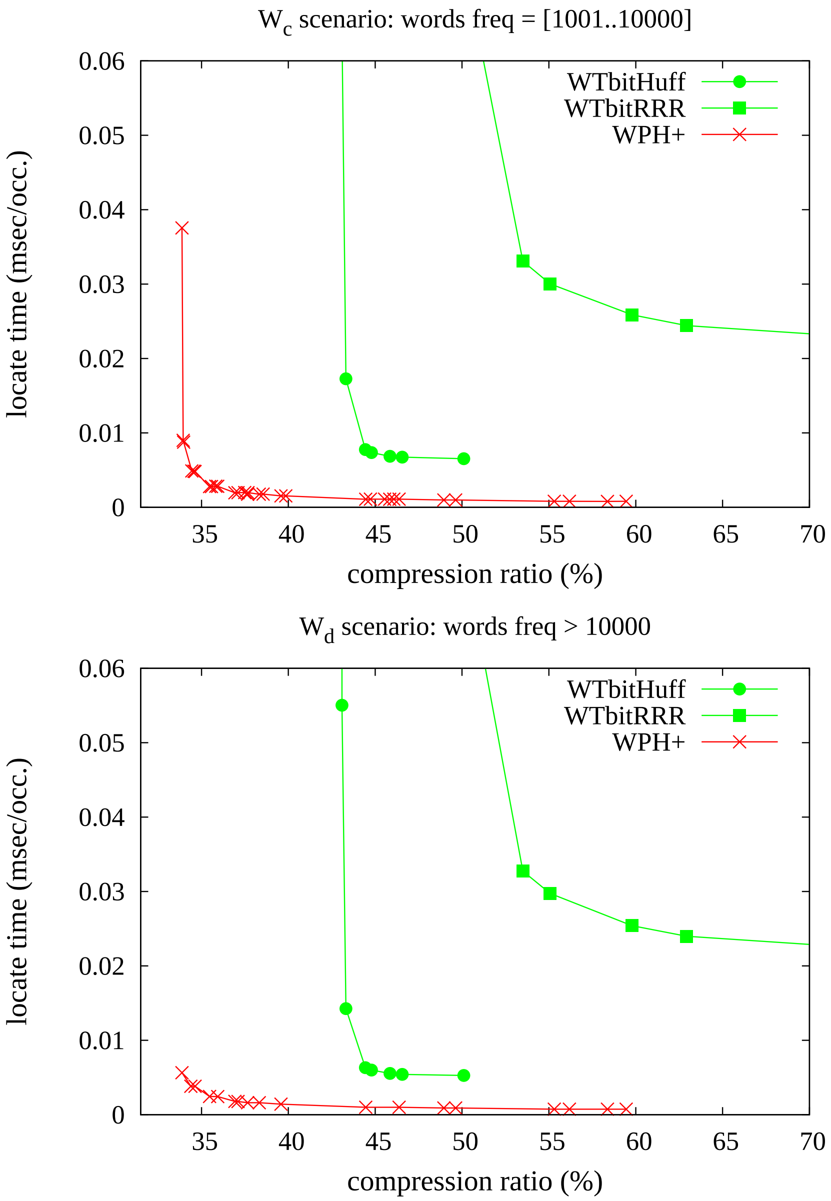


Figure 9.18: Time/space trade-off for *locating* more frequent words with BOC-WT strategy over PH against a word-based Huffman-shaped wavelet tree and a balanced binary wavelet tree using RRR.

tree shape. Therefore, WTbitHuff uses significantly more space than the zero-order entropy of the text (notice that the compression ratio obtained by binary Huffman code over ALL corpus is 28.55%). The balanced binary wavelet tree using RRR obtains practically the same time for all the scenarios, regardless the frequency of the word. Since it is a balanced tree, all the words are represented in the same level and they require the same number of select operations when searching for a word. Its space/time tradeoff is completely dominated by the WPH+. The differences become greater when display snippets around the occurrences. This is due to the fact that those occurrences are generally surrounded by very frequent words, such as prepositions or articles, and those frequent words require a higher number of rank operations to be decompressed than WTbitHuff or WPH+, where the leaves associated with those words are located in upper levels of the tree and can be reached computing a smaller number of rank operations.

9.6.2 Comparison with word-based self-indexes

Some word-based self-indexes have been developed, such as the WCSA and WSSA [BFN+08, BCF+11]. These self-indexes achieve compression ratios of 35-40% and provide indexed word-based searches. However, as they are built considering a vocabulary of the words of the text, not just characters, they cannot search for arbitrary text substrings, but only for words and phrases.

For the comparison, we create several indexes with different sizes, varying construction parameters such as the sample periods t_A, t_A^{-1} and t_Ψ for A, A^{-1} and Ψ in the case of WCSA and the sampling parameters t_{pos}, t_{bit_1} and t_{bit_2} for WSSA. This gives us an interesting space/time trade-off such that we can compare the performance of our BOC-WT data structure over PH (WPH+) with different sizes for the rank and select structures of blocks and superblocks. We use the same large text (ALL corpus) and the same sets of patterns than in the previous sections, measuring *locate* time and *display* time (in msec/occurrence).

Locate time Figures 9.21 to 9.24 show the performance of locating all the occurrences of several patterns. Figure 9.21 illustrates the behavior of WPH+, WCSA and WSSA for scenarios W_a (top) and W_b (bottom), whereas Figure 9.22 shows times results for scenarios W_c (top) and W_d (bottom). Figure 9.23 illustrates the behavior of WPH+, WCSA and WSSA for scenarios P_2 (top) and P_4 (bottom), whereas Figure 9.24 shows times results for scenarios P_6 (top) and P_8 (bottom).

We can observe from the results that WPH+ is extremely fast to locate the occurrences of individual words. However, both word-based self-indexes outperform WPH+ when searching for phrase patterns. Only when searching for phrases composed of two words WPH+ obtains efficient time results; for long phrase patterns WPH+ becomes considerably more inefficient compared to both WCSA and WSSA. This is an expected result since suffix arrays were designed to efficiently

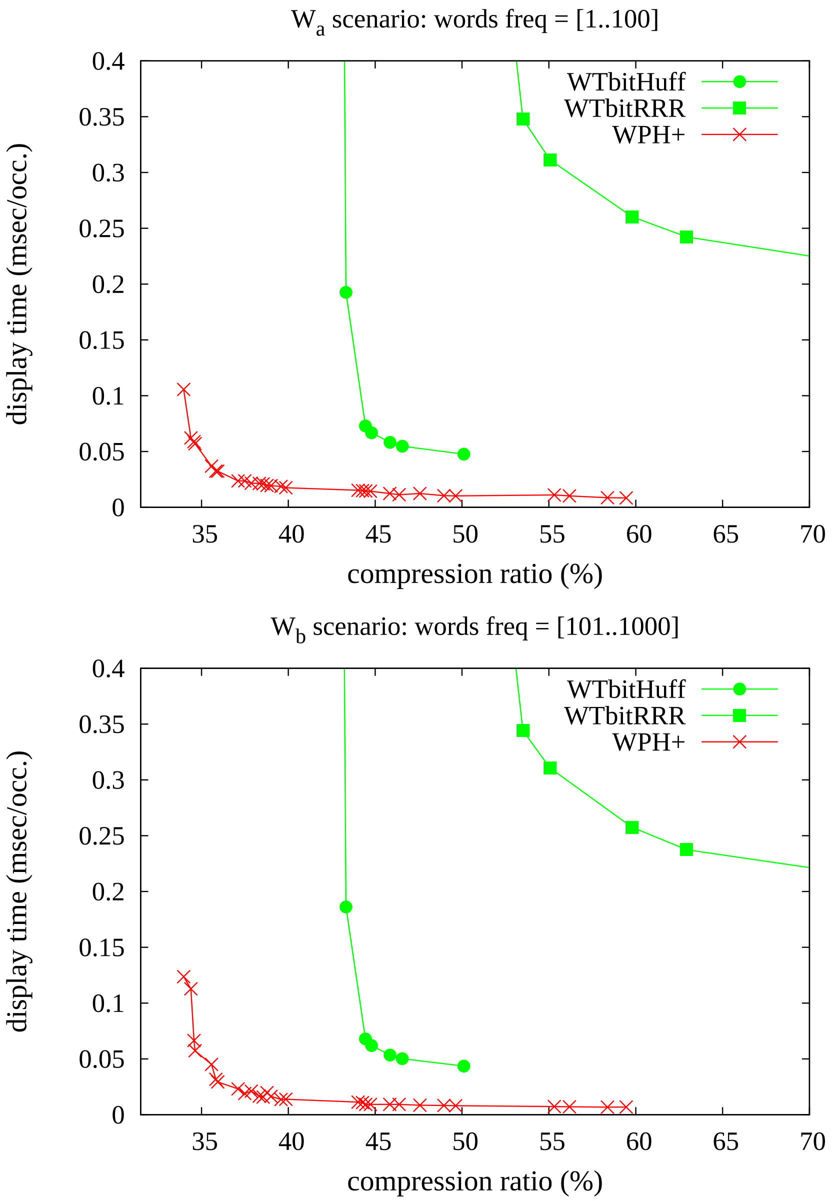
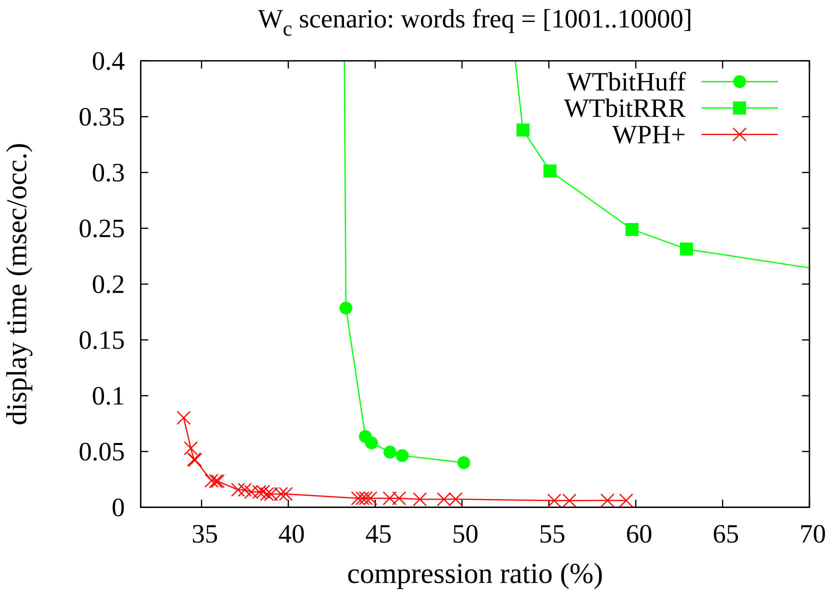


Figure 9.19: Time/space trade-off for *displaying* the occurrences of less frequent words with BOC-WT strategy over PH against a word-based Huffman-shaped wavelet tree and a balanced binary wavelet tree using RRR.



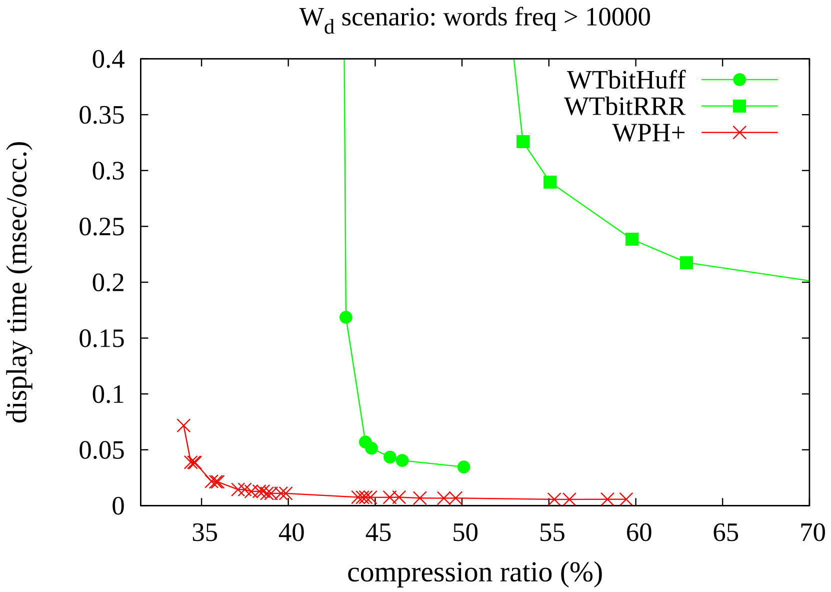


Figure 9.20: Time/space trade-off for *displaying* the occurrences of more frequent words with BOC-WT strategy over PH against a word-based Huffman-shaped wavelet tree and a balanced binary wavelet tree using RRR.

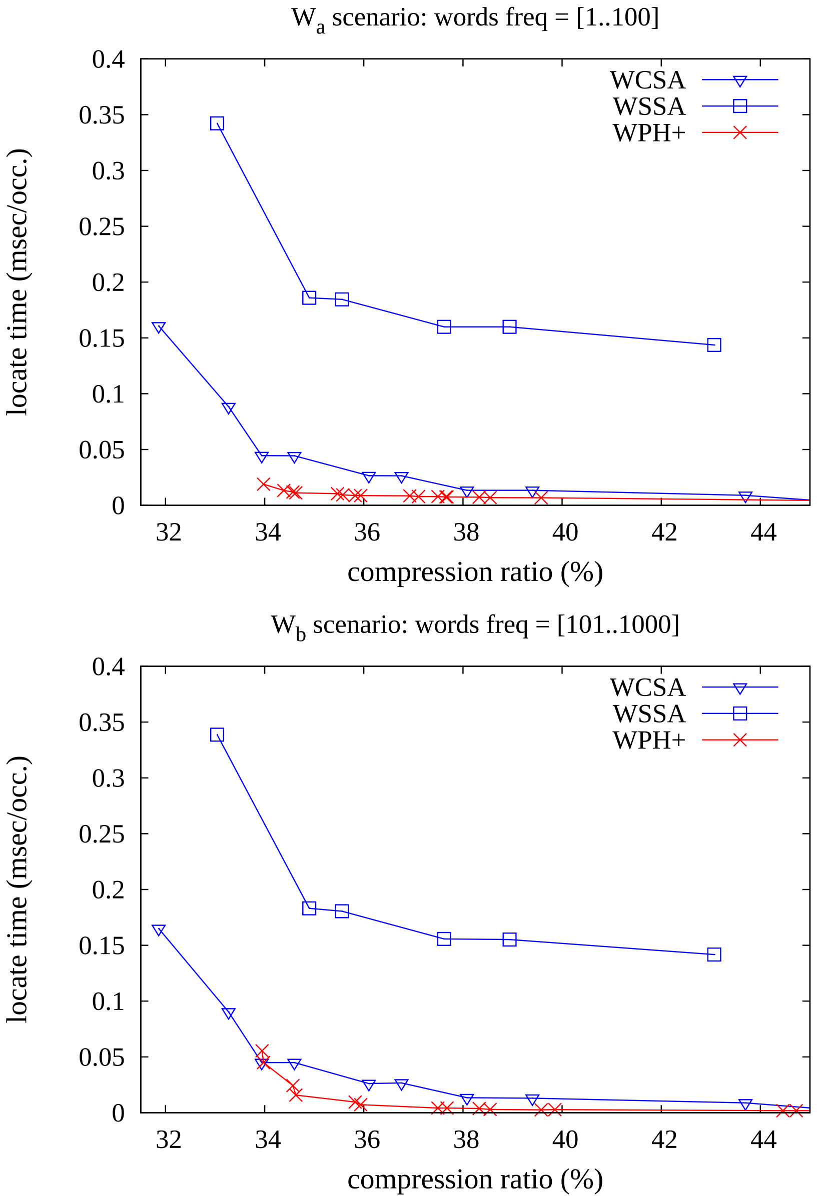


Figure 9.21: Time/space trade-off for *locating* less frequent words with BOC-WT strategy over PH against other word-based self-indexes.

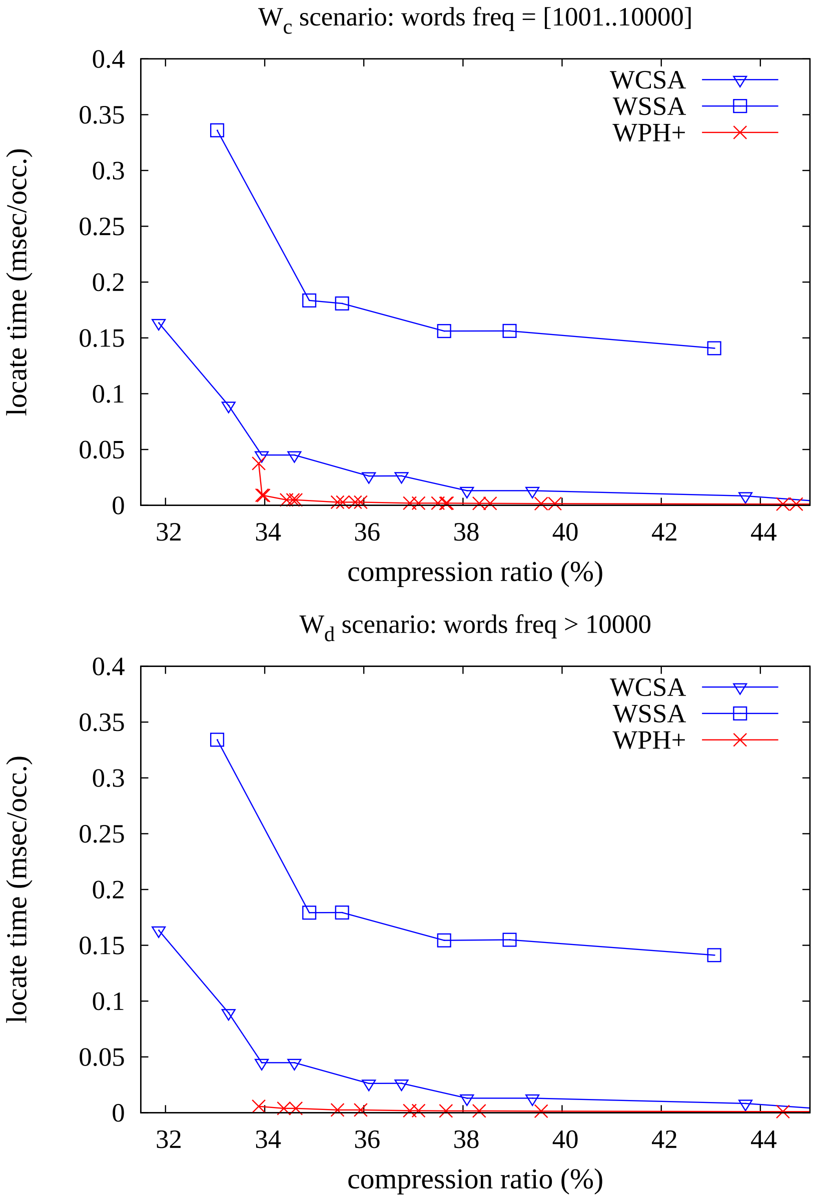


Figure 9.22: Time/space trade-off for *locating* more frequent words with BOC-WT strategy over PH against other word-based self-indexes.

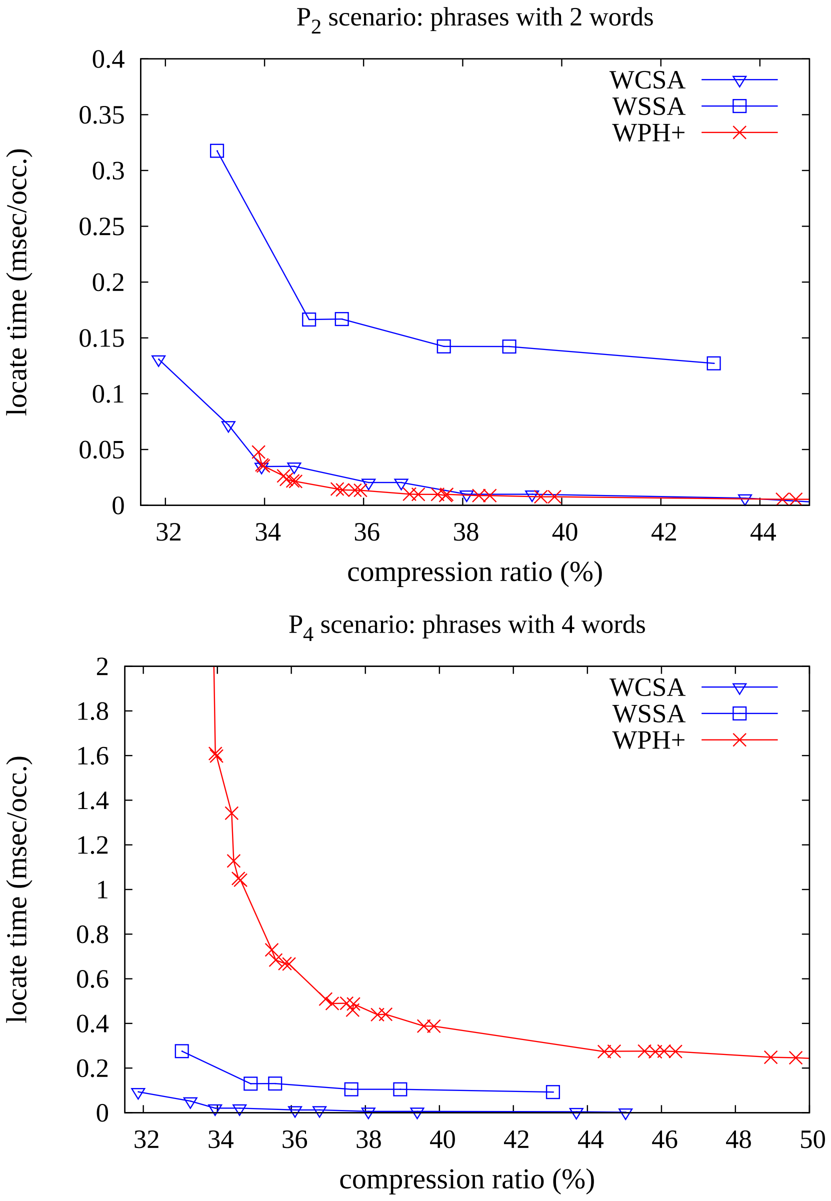


Figure 9.23: Time/space trade-off for *locating* short phrase patterns with BOC-WT strategy over PH against other word-based self-indexes.

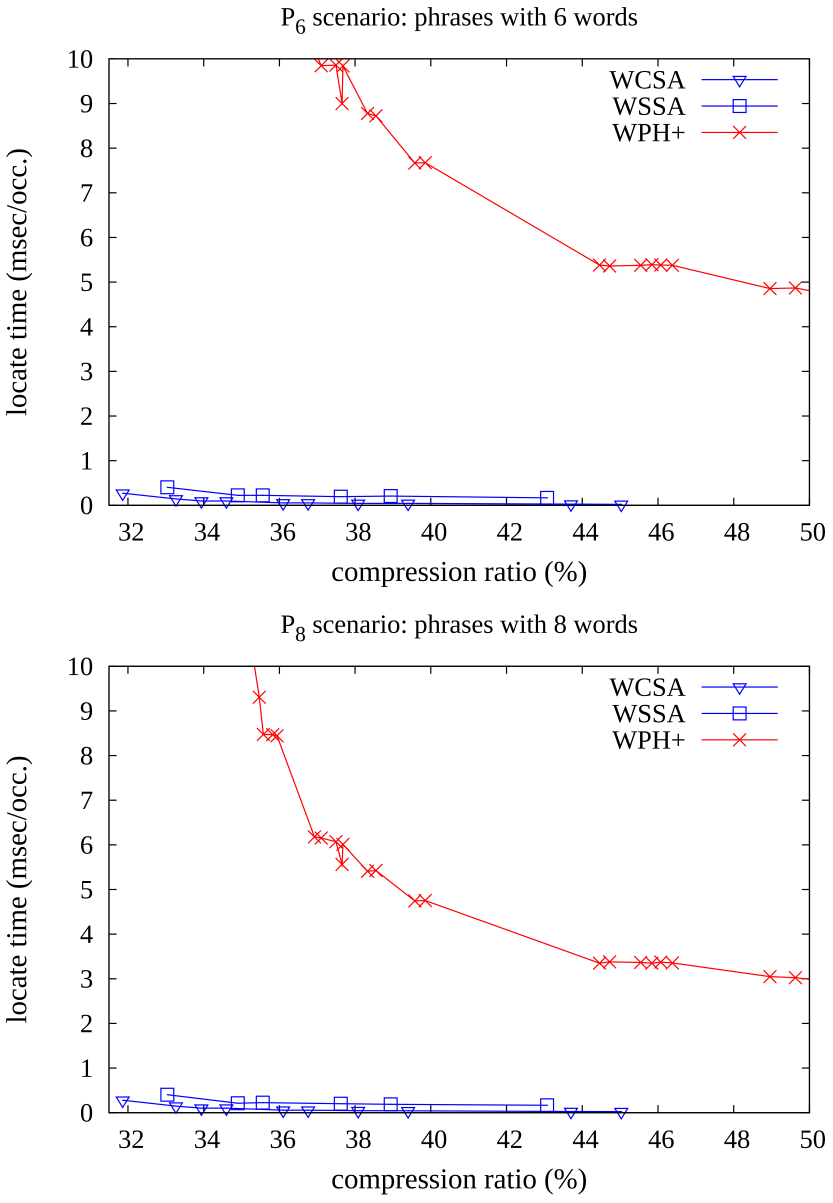


Figure 9.24: Time/space trade-off for *locating* long phrase patterns with BOC-WT strategy over PH against other word-based self-indexes.

count and locate all the occurrences of substrings of the text. WCSA and WSSA are two word-based self-indexes based on suffix arrays, hence, they easily recover all the occurrences of word phrases of the text.

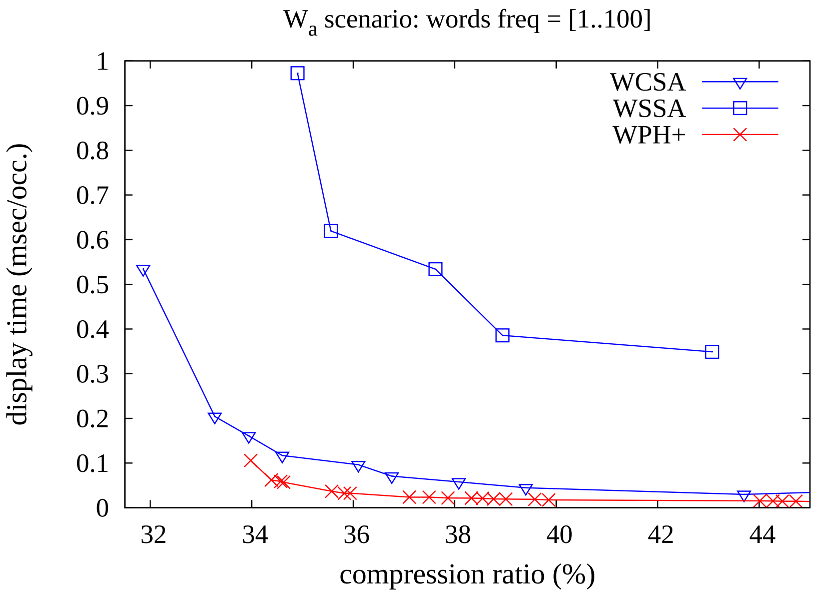
WPH built over ALL corpus occupies 33.32% of the text, when no rank or select structures are used. In the figures, we illustrate the behavior of several configurations of WPH+ using a structure for rank and select operations varying the sample period, all of them occupying more than 33.87% of the size of text. When very little space is used for rank and select structures, the compression ratio obtained gets close to this value, but it becomes very inefficient due to the sparseness in the samples of the rank and select directory of blocks and superblocks. The efficiency of WCSA and WSSA also decreases when we use less space, but they can index the same text using less than 33%.

Extract-snippet time Figures 9.25 to 9.28 show the performance of displaying snippets around the occurrences of several patterns. Figure 9.25 illustrates the behavior of WPH+, WCSA and WSSA for scenarios W_a (top) and W_b (bottom), whereas Figure 9.26 shows times results for scenarios W_c (top) and W_d (bottom). Figure 9.27 illustrates the behavior of WPH+, WCSA and WSSA for scenarios P_2 (top) and P_4 (bottom), whereas Figure 9.28 shows times results for scenarios P_6 (top) and P_8 (bottom).

The results obtained for the display operation are analogous to the results obtained for locating the occurrences. Remember that the display operation consists in first locating the occurrences and then extracting some portion of text around those occurrences. Therefore, since the extraction of the text is more efficient for our proposed BOC-WT strategy, display time results are slightly better compared to the locate operation. Let us compare, for instance, the top subfigure of Figure 9.23 and the top subfigure of Figure 9.27. We can observe how display operation is always better using WPH+ than WCSA, while the performance of both structures is very similar and there is practically no difference for the locate operation. WCSA is again the best choice to display some portions of the text around the occurrences of long phrase patterns.

9.6.3 Comparison with word-based preprocessed full-text self-indexes

Full-text self-indexes take space proportional to the compressed text, replace it, and permit fast indexed searching on it [NM07]. Those indexes work for any type of text, they typically index all the characters of the text, achieve compression ratios of 40-60%, and can extract any text substring and locate the occurrences of pattern strings in a time that depends on the pattern length and the output size, not on the text size (that is, searching is not a sequential process). Most can also count the number of occurrences of a pattern string much faster than by locating them.



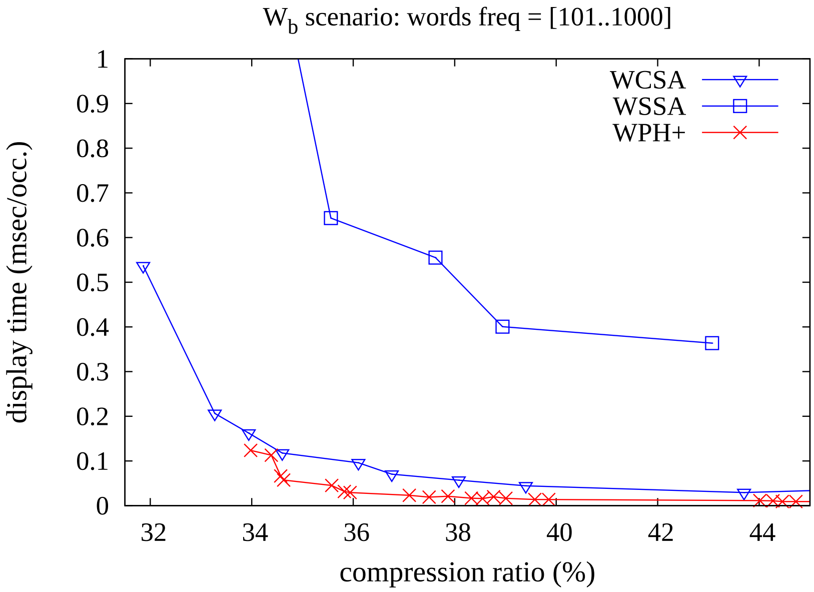
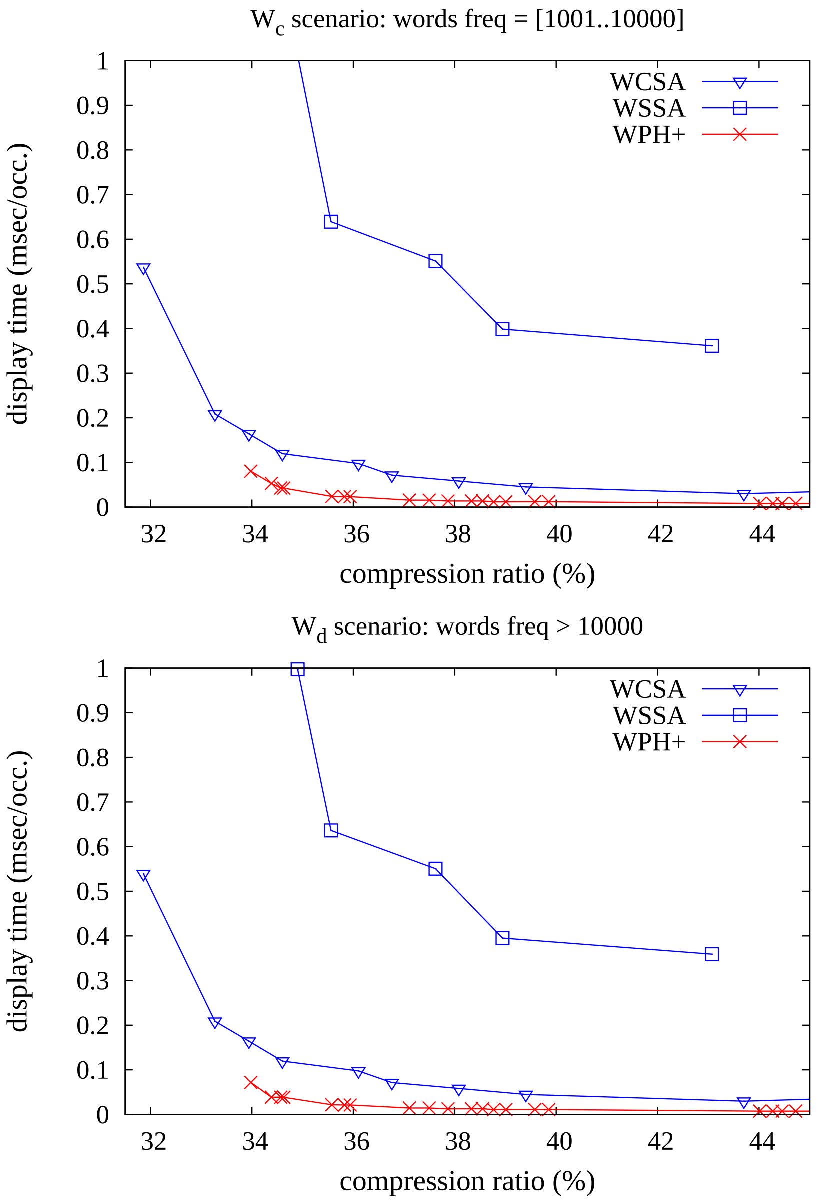


Figure 9.25: Time/space trade-off for *displaying* the occurrences of less frequent words with BOC-WT strategy over PH against other word-based self-indexes.



W_c scenario: words freq = [1001..10000]

W_d scenario: words freq > 10000

Figure 9.26: Time/space trade-off for *displaying* the occurrences of more frequent words with BOC-WT strategy over PH against other word-based self-indexes.

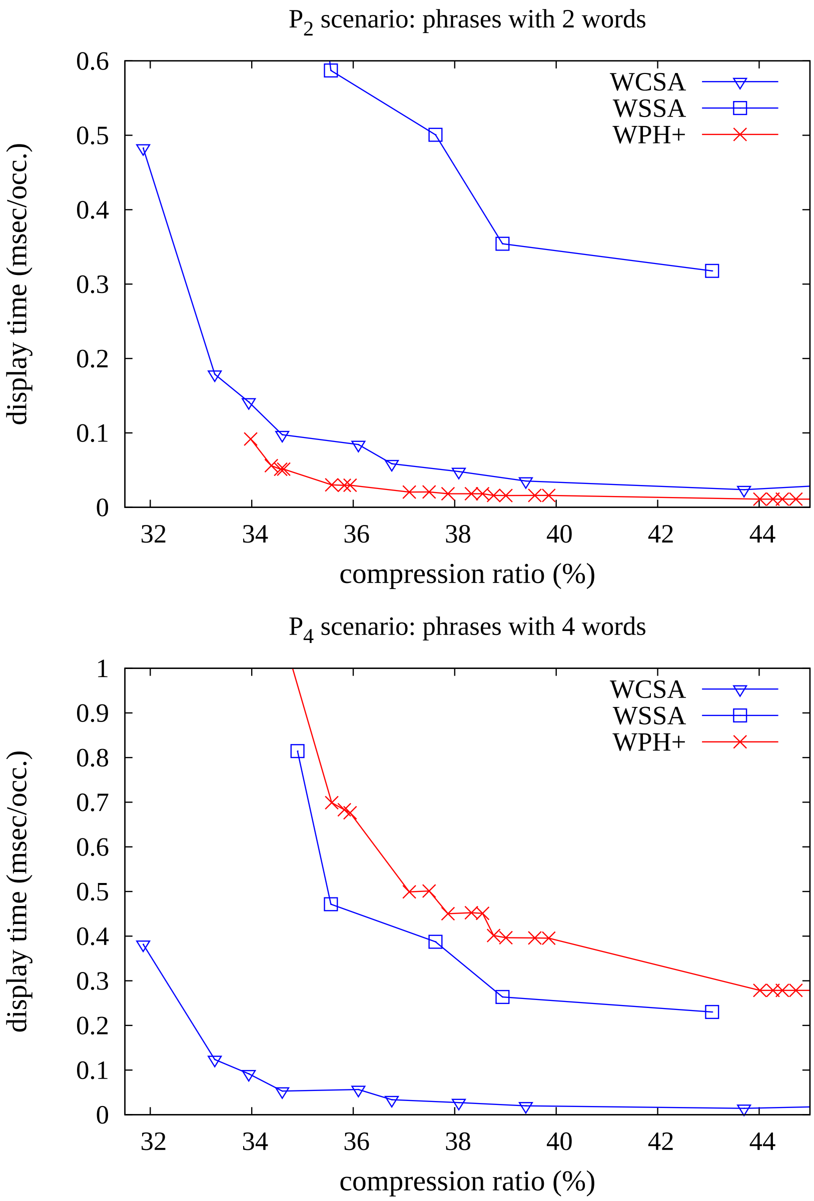
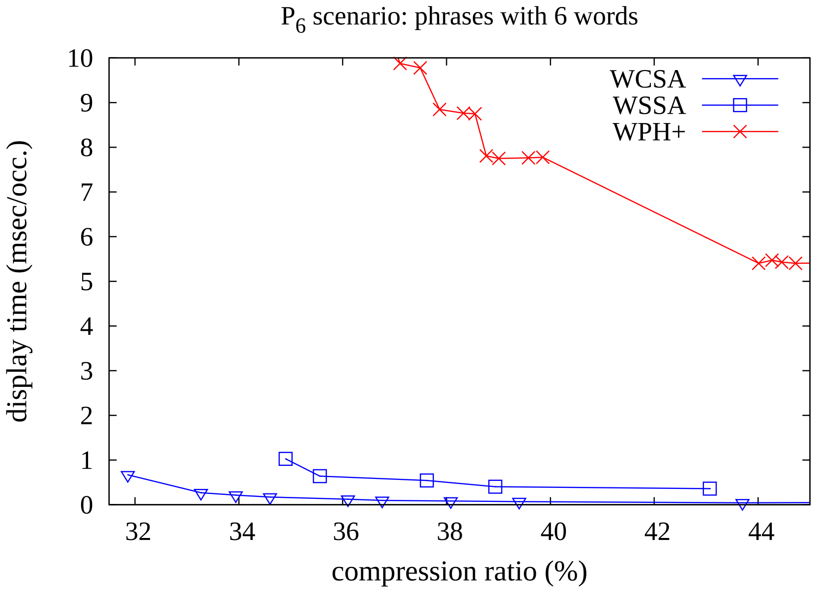


Figure 9.27: Time/space trade-off for *displaying* the occurrences of short phrase patterns with BOC-WT strategy over PH against other word-based self-indexes.



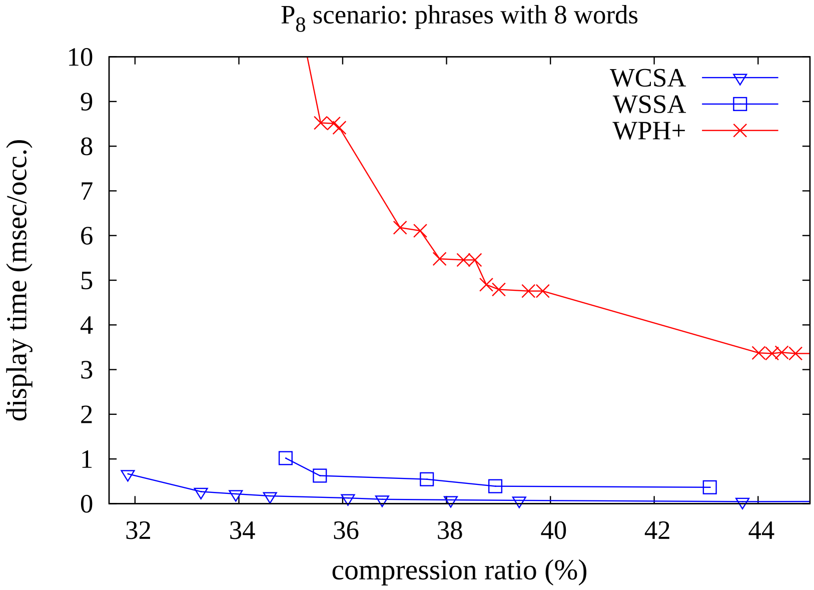


Figure 9.28: Time/space trade-off for *displaying* the occurrences of long phrase patterns with BOC-WT strategy over PH against other word-based self-indexes.

Fariña *et al.* [FNP08] showed that typical full-text self-indexes achieve much better space and time performance when indexing is preceded by a compression step using a prefix- and suffix-free semistatic word-based encoding. In their work, they use Tagged Huffman code and a new suffix-free Dense-Code-based compressor to compress the original text. Since both of these compressors use bytes as target alphabet, full-text self-indexes can directly index the compressed text, as they are designed to index a sequence of characters. With this approach, they can index the text using very little space, close to 35% of the original text instead of the approximately 60% of space than full-text self-indexes usually require. In addition, these preprocessed full-text self-indexes allow for efficient indexed searches of words and phrases using a very compact space, just by searching for the sequence of bytes which compose the pattern encoded by the word-based compressor. These benefits are obtained at the expense of losing the capability of searching for any kind of pattern of the original text, just words and phrases of words can be searched for if this word-based preprocessing step is done.

We considered four self-indexes from the *PizzaChili* site[5]: the Compressed Suffix Array (CSA) [Sad03], the Succinct Suffix Array (SSA) [MN05], the Alphabet-Friendly FM-index (AFFM) [FMMN07] and the LZ-index (LZI) [AN10], each of them built over text compressed with both TH and SCBDC. We also include in the comparison the results for the original full-text self-indexes, built over plain text.

We use Congressional Record 1993 corpus (CR) to compare the performance of WPH+ with a set of different configurations of the mentioned full-text self-indexes where different values of the parameter *sample-rate* were used to obtain different space/time tradeoffs for the indexes. We compare *counting* and *locating* times for two set of patterns, one with 100 individual words randomly chosen from the vocabulary, and other set with 1,000 different phrase patterns of composed of 4 words. The overall number of occurrences for such sets are $257,745$ and $330,441$ respectively. We also measure *extracting* time, such that we retrieve portions of text (we decompress 2,000 contiguous words) starting at randomly chosen positions.[6] We denote X+text the alternative using the full-text self-index X built over plain text, X+TH the alternative built over compressed text using TH and X+SCBDC the alternative built over compressed text using the suffix-free Dense-Code-based compressor. WPH+ stands for our BOC-WT data structure built over PH.

Count operation Figure 9.29 shows the time required by the different self-indexes to perform *count* operations, that is, the average time (in milliseconds per pattern) that each self-index needs to count all the occurrences of a pattern. At the top part of the figure, we illustrate the time to count patterns composed by just one

[5] Code available at `http://pizzachili.dcc.uchile.cl`.

[6] I would like to thank Jose R. Paramá for providing the graphical and numeric results of the experiments for these full-text indexes and for the detailed explanations about the preprocessing method.

word, whereas the figure at the bottom displays the time to count the number of occurrences of phrase patterns composed of 4 words.

As we can observe, WPH+ does not *count* occurrences of phrase patterns as efficiently as it counts occurrences of individual words. This is due to the fact that the algorithm to count the occurrences of patterns with several words requires a bottom-up traversal of the tree in order to locate the occurrences of the least frequent word of the phrase, and then it occasionally requires several top-down traversals to discard false matches of the pattern. On the contrary, counting individual words is a very efficient procedure, that involves just a simple bytewise rank operation.

WPH+ obtains the best time results among all the alternatives for counting occurrences of individual words when the size of the index is bigger than 40% of the text. Other self-indexes, such as AFFM+SCBDC, CSA+PH and CSA+SCBDC can achieve more compact spaces and better times. Notice that WPH built over CR corpus cannot occupy less than 32.33% of the text, and all its configurations close to that space use very slow rank and select structures, and thus count operation becomes inefficient.

Locate operation Figure 9.30 shows the time required by *locate* operation, that is, the average time (in milliseconds per occurrence) that each self-index needs to locate all the occurrences of a pattern. At the top part of the figure, we illustrate the time to locate the occurrences of individual words, whereas the figure at the bottom displays the time to locate the occurrences of phrase patterns of 4 words.

We can observe in the top part of Figure 9.30 that locating all the occurrences of individual words is a very efficient operation for WPH+ compared to the other self-indexes. However, locating occurrences of phrase patterns (bottom part of the figure) requires a more complex procedure and times are slower than for locating patterns of just 1 word. Comparing the results for WPH+ and the other self-indexes in the figure, we can see that WPH+ becomes the best option when we use little space, but other alternatives, such as CSA over both TH and SCBDC, obtain better times for compression ratios around 45%. LZ-index becomes the preferred choice for larger spaces, that is, when the indexes occupy more than 75% of the text.

Extract operation Figure 9.31 shows the time required by *extract* operation. We measure the average time (in milliseconds per character extracted) that each self-index needs to decompress 2,000 words starting at 1,000 randomly chosen position of the text.

As we can observe from the figure, *extract* operation is much faster for WPH+ than for the other self-indexes. This operation requires a sequential processing of the nodes of the tree-shaped structure in addition to several rank operations to initialize the pointers at those nodes. Since the portion of text extracted is

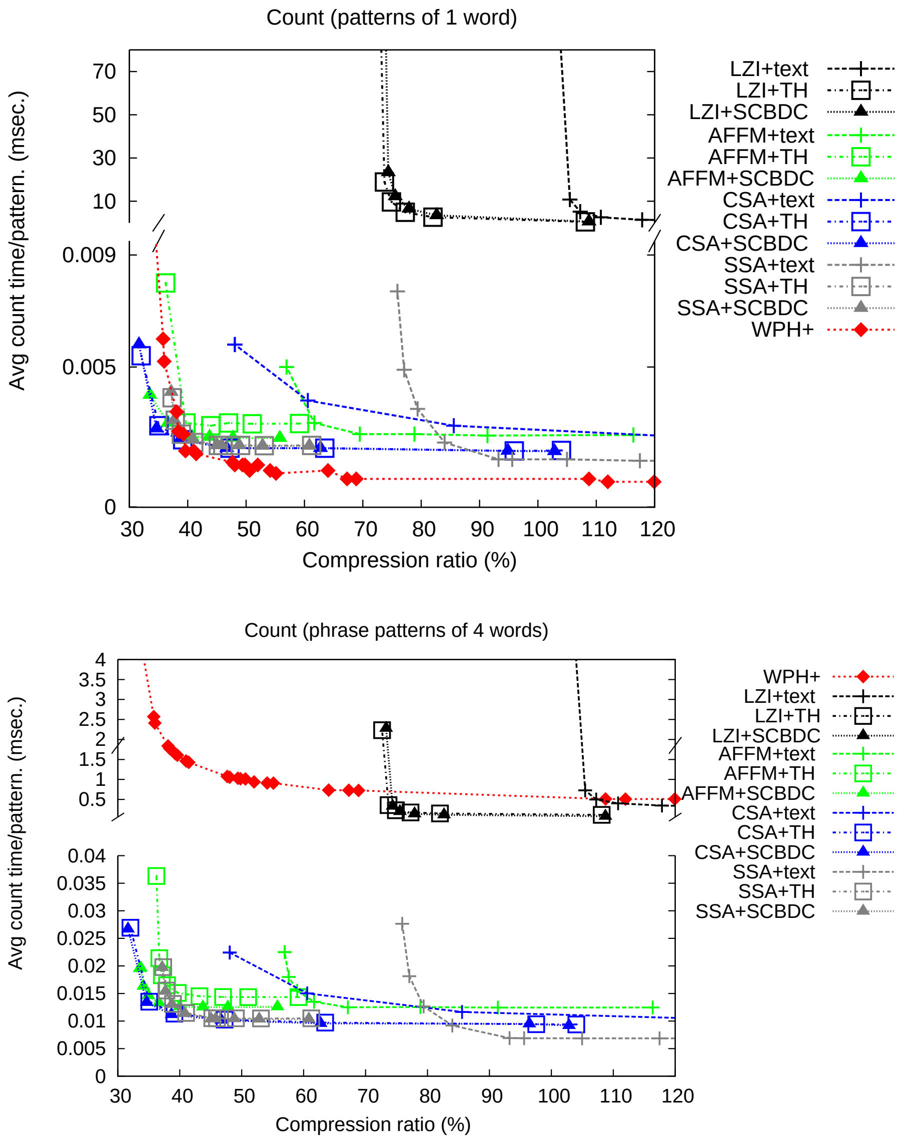


Figure 9.29: Time results for *count* operation compared to other self-indexes.

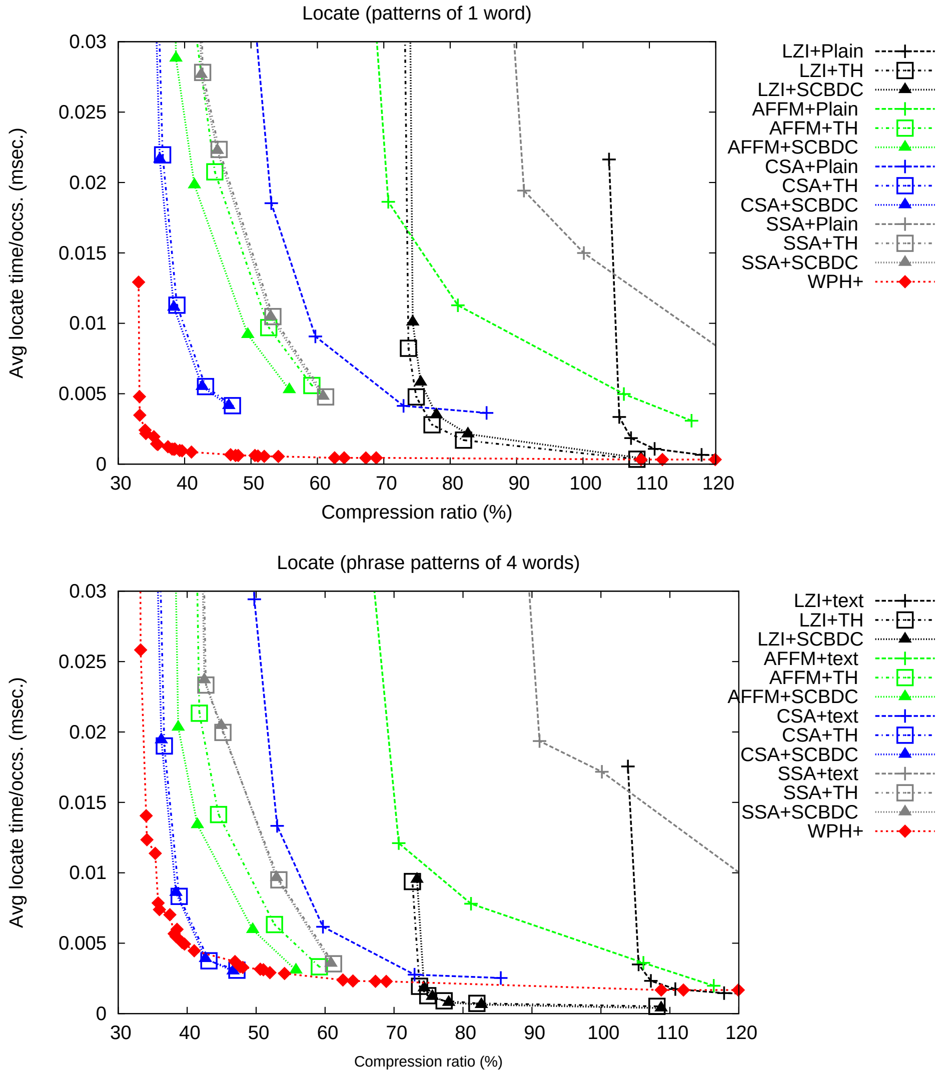


Figure 9.30: Time results for *locate* operation compared to other self-indexes.

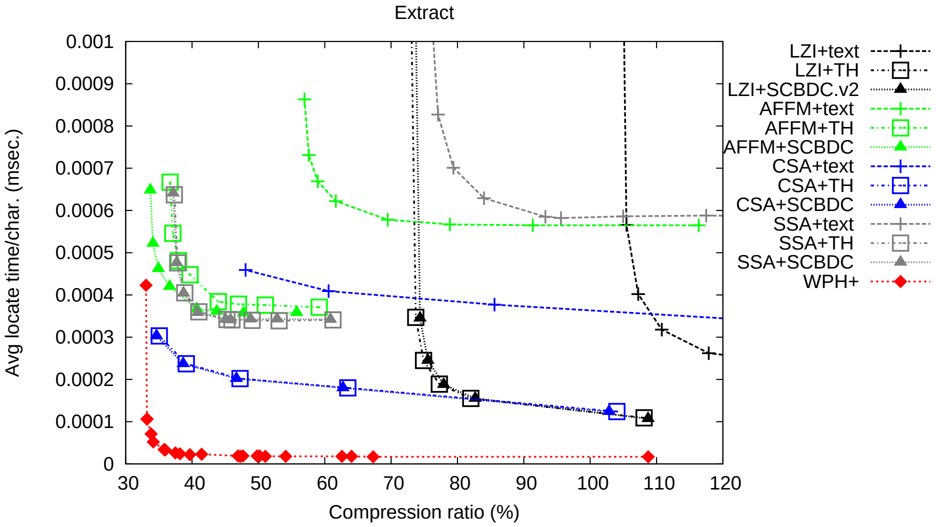


Figure 9.31: Time results for *extract* operation compared to other self-indexes.

significant (we extracted 2,000 contiguous words), these rank operations do not worsen significantly the overall performance of the decompression process. The other self-indexes require a more complex procedure to extract portions of text, so WPH+ becomes the preferred choice when retrieving snippets of the text.

As we can observe in the experimental comparison of this section, BOC-WT strategy obtains the best time results for counting and locating individual words, as well as for extracting portions of text, compared to some word-based preprocessed full-text self-indexes. However, locating, and especially, counting phrase patterns are not as efficient as for the other self-indexes. Locating phrase patterns using WPH+ is still the preferred alternative when using very little space. However, we must remember that WPH+ and the word-based preprocessed full-text self-indexes can only search for whole words or phrases of words in the compressed text, unlike the full-text self-indexes built over plain text. These full-text self-indexes built over plain text occupy much more memory than WPH+, but are more flexible as they can search for any kind of string.

Chapter 10

Discussion

10.1 Main contributions

It has been long established that semistatic word-based byte-oriented compressors, such as Plain Huffman, End-Tagged Dense Code and Restricted Prefix Byte Codes, are useful not only to save space and time, but also to speed up sequential search for words and phrases. However, the more efficient compressors such as PH and RPBC are not that fast at searching or random decompression, because they are not self-synchronizing. In this part of the thesis we have proposed a new data structure called BOC-WT that performs a simple reorganization of the bytes of the codewords obtained when a text is being compressed, such that it can produce clear codewords boundaries for those compressors. This gives better search capabilities and random access than those of the byte-oriented compressors, even those that pay some compression degradation to mark codeword boundaries (TH, ETDC).

As the reorganization permits carrying out all those operations efficiently over PH, the most space-efficient byte-oriented compressor, the usefulness of looking for coding variants that sacrifice compression ratio for synchronization strategies to improve the searching or decoding performance is questioned: the proposed data structure over Plain Huffman (WPH) will do better in almost all aspects.

The reorganization has also surprising consequences related to implicit indexing of the compressed text. Block-addressing indexes over compressed text have been long considered the best low-space structure to index a text for efficient word and phrase searches. They can trade space for speed by varying the block size. We have shown that BOC-WT provides a powerful alternative to these inverted indexes. By adding a small extra structure to the BOC-WT, search operations are speeded up so sharply that the structure competes successfully with block-addressing inverted indexes that take the same space on top of the compressed text. Especially, our structure is superior when little extra space on top of the compressed text is permitted. We have also compared these implicit indexing properties of the BOC-WT data structure with other word-based self-indexes, obtaining efficient time results for locating individual words and extracting portions of text. Searching for phrase patterns can also be solved by our new data structure, but other self-indexes outperform BOC-WT in this aspect. However, BOC-WT is still the preferred choice for locating and displaying the occurrences of short phrases composed of two words.

10.2 Other Applications

BOC-WT is a new data structure that can represent any natural language text in a compressed and self-indexed way. However, it can be particularized or extended such that it becomes an attractive solution in other scenarios. More specifically, we now mention two works where BOC-WT has been applied to different domains. The first one consists on a modification of the data structure to efficiently represent XML documents. The second one adapts BOC-WT data structure to the most studied Information Retrieval problem: ranking and searching over document collections. A new search engine was developed, where BOC-WT was used to obtain relevant documents for user queries. For this goal, the parallelization of the BOC-WT data structure was also studied.

10.2.1 A compressed self-indexed representation of XML documents

Brisaboa *et al.* [BCN09] presented a direct application of the BOC-WT data structure presented in this thesis to represent any XML document in a compressed and self-indexed form, called XML Wavelet Tree (XWT). It permits to compute any query or procedure that could be performed over the original XML document in a more efficient way using the XWT representation, since it is shorter and has some indexing properties.

The XWT data structure just consists in applying the BOC-WT data structure using (s, c)-Dense Code over the XML document. Two different vocabularies are considered. These two vocabularies are created during the parsing step of the XML document. One stores the different start- and end-tags and therefore the structure of the document. The other stores the rest of the words. With this distinction, it is possible to keep all the tags in the same branch of the XWT. As they follow the document order, the relationships among them are maintained as in the original XML document. Hence, structural queries can be efficiently solved using this data structure: only those nodes storing the structure of the document are accessed, and the rest of the compressed text can be omitted.

The compressed representation of the XML document is then obtained in a similar way than it was explained for BOC-WT, only taking into account the particularity of using different vocabularies to isolate the tags of the document. Accessing to random positions of the document, retrieving the original XML document or doing searches on it, such as counting, locating words or phrases, can be performed using the same algorithms explained in Section 8.2. Other XPath queries can also be easily answered by traversing the tree, such as obtaining the pairs of start-end tags containing a word or searching attributes values (which is translated in the XWT as a phrase search and the same algorithm as explained for searching for phrases in BOC-WT can be used).

10.2.2 Searching document collections

BOC-WT data structure can also be used to represent a document collection. Let $\mathbb{D} = D_1, \ldots, D_N$ be a collection of N documents, where each document D_i is modeled as a sequence of terms (or words) from a vocabulary Σ of size $|\Sigma|$. Conjunctive queries of the form $t_1 \wedge t_2 \cdots \wedge t_k$, asking to report the documents that contain all the terms $t_1, \ldots, t_k$, are one of the most common kinds of queries issued to text retrieval systems.

González [Gon09] analyzes the behavior of BOC-WT against an inverted index structure inside a complete search engine implemented in a cluster of nodes. The indexes were built over a real collection of documents, consisting in a subset of the UK Web and real logs from Yahoo! UK are used to test the performance of BOC-WT and the inverted index. The experimental evaluation indicates that the inverted index is more efficient in searching times due to several information that is needed for the ranking step and is already precalculated and stored in the inverted index. BOC-WT requires some extra computations that worsen searching times. However, it offers flexibility and independence of the ranking method used, which might be of great interest. Several ranking strategies can be implemented without rebuilding the index or using any extra space. In addition, the original text can be retrieved from the data structures that compose the BOC-WT, such that snippet extraction can be performed without using any other machine nor accessing to secondary memory, in time comparable to that of locating the occurrences. This becomes an attractive advantage, since it avoids the need of having extra servers storing the collection of documents and reduces communication costs between the nodes of the cluster.

Since the experimental evaluation showed that each alternative has advantages and disadvantages, the author proposes a hybrid approach that combines both strategies to exploit their advantages and reduce their disadvantages. This hybrid approach consists in using the BOC-WT data structure during the query processing step in order to generate an inverted index for the terms most referenced by the queries over a period of time. This strategy takes into account the fact that repetitions of the same terms are common in real queries, some terms appear in many queries for short periods of time whereas some others keep recurring for long periods of time. In both cases, the inverted lists of those terms are usually consulted several times, hence, it is an attractive choice to keep those inverted lists in a specific-purpose cache. Inside this cache, the inverted index dynamically generated using BOC-WT is stored and replaced when needed. Thus, there is no space consumption for infrequently searched terms.

Using this schema, the search engine can process a real dataset of queries in similar times and spaces than an inverted index. In addition, the whole document collection is maintained in main memory, such that snippet generation is also possible, while an inverted index would need extra space to store the text in compressed form, and snippet generation might require some accesses to secondary memory.

Arroyuelo *et al.* [AGO10] recently presented a study on the support of conjunctive queries in self-indexed text retrieval systems. They referenced our proposed BOC-WT data structure as an example of a new trend in compressed indices, since it does not store the occurrence lists, but permits generating them on the fly. With this strategy, considerable space savings are obtained but query times are increased. In order to solve some operations that are fundamental in IR, such as conjunctive queries over document collections, BOC-WT data structure can be adapted by concatenating all the documents and indexing the whole collection as a unique text T, or more generally, any rank/select data structure can be adapted similarly. A special separator symbol \$, different to any symbol of the vocabulary, is used to create T, so that the sentence is build as $T[1..n] = \$D_1\$D_2\$\ldots\$D_N\$$, where each document D_i has assigned a unique identifier i. Given any position $1 \leq j \leq n$, the document identifier corresponding to position j can be computed as $rank_{\$}(T, j)$. Given a document identifier $1 \leq i \leq N$, the starting position within T for document D_i can be computed as $select_{\$}(T, i) + 1$.

As we have seen, BOC-WT competes successfully with an inverted index when reporting all the occurrences of a query term t. This operation is relevant for text searching, where all the occurrences need to be found. However, for conjunctive queries this approach is not efficient, since we should search for every occurrence of t, and then determine the document that contains each one, so that the documents are reported without repetitions. This approach is inefficient when there are many occurrences of t, but just a few documents actually contain it. The authors proposed an algorithm to solve this query in time proportional to the number of documents, rather to the number of occurrences of the query. Basically, to find all the documents containing a term t they proceed as follows. They locate the first occurrence of t within T using a select operation and compute the document identifier corresponding to that occurrence, which is reported. Then they jump up to the end of the current document using another select operation and perform a rank operation to count the number of occurrences of t up to that position. Then, they jump to the next document containing the following occurrence of t using a select operation, and repeat the procedure until they reach the end of the collection. In addition, they presented several algorithms to perform conjunctive queries $t_1 \wedge t_2 \cdots \wedge t_k$ that are more efficient than obtaining the occurrences lists and intersecting them. The behavior of these algorithms depends on the practical implementation of rank and select operations. When comparing their proposal with inverted indexes, they showed that their algorithms are about 5-7 times slower but inverted indexes require about 1.5 times their space when snippet extraction is required (that is, the space for storing the text must be also account). Hence, further work needs to be done in order to obtain a query performance similar to that of inverted indexes.

Part III

Compact Representation of Web Graphs

Chapter 11

Introduction

The directed graph representation of the World Wide Web has been extensively used to analyze the Web structure, behavior and evolution. However, those graphs are huge and do not fit into main memory, whereas the required graph algorithms are inefficient in secondary memory. Compressed graph representations reduce their space while allowing efficient navigation in compressed form. As such, they allow running main-memory graph algorithms on much larger Web subgraphs. In the following chapters of this thesis we present a Web graph representation based on a very compact tree structure that takes advantage of large empty areas of the adjacency matrix of the graph.

We start this chapter by introducing the usage of Web graphs in Information Retrieval and the need of a navigable compact representation in Section 11.1. Then, we revise in Section 11.2 some basic concepts and properties of Web graphs and we finish the chapter by studying the current state-of-the-art in Web graph compression in Section 11.3.

11.1 Motivation

The World Wide Web structure can be regarded as a directed graph at several levels, the finest grained one being pages that point to pages. Many algorithms of interest to obtain information from the Web structure are essentially basic algorithms applied over the Web graph. One of the classical references on this topic [KKR$^+$99] shows how the HITS algorithm to find hubs and authorities on the Web starts by selecting random pages and finding the induced subgraphs, which are the pages that point to or are pointed from the selected pages. Donato *et al.* [DMLT05] show how several common Web mining techniques, used to discover the structure and evolution of the Web graph, build on classical graph algorithms such as depth-first search, breath-first-search, reachability, and weakly and strongly connected components. A more recent work [STKA07] presents a technique for Web spam detection that boils down to algorithms for finding strongly connected components, for clique enumeration, and for minimum cuts.

The problem of how to run typical graph algorithms over those huge Web graphs is always present in those approaches. Even the simplest external memory graph algorithms, such as graph traversals, are usually non disk-friendly [Vit01]. This has pushed several authors to consider *compressed graph representations*, which aim to offer memory-efficient graph representations that still allow for fast navigation without decompressing the graph. The aim of this research is to propose a new compression method such that classical graph algorithms can be run in main memory over much larger graphs than those affordable with a plain representation.

11.2 Basic concepts on Web graphs

Let us consider a graph $G = (V, E)$, where V is the set of vertices (or nodes) and E is the set of edges, which is a subset of $V \times V$. We denote $n = |V|$ the number of nodes of the graph and $m = |E|$ the number of edges. We call the direct neighbors of a node $v \in V$ those $u \in V$ such that $(v, u) \in E$, and reverse neighbors of a node $v \in V$ those $u \in V$ such that $(u, v) \in E$. A graph in which the edges are not ordered pairs of nodes, so the edge (u, v) is identical to the edge (v, u) is called an undirected graph. On the contrary, edges of directed graphs are ordered pairs, and there can be nodes $(u, v) \in E$ but $(v, u) \notin E$.

There are two standard strategies for graph representations. One uses the *adjacency lists* of each node and the other uses the *the adjacency matrix*.

- Adjacency lists representation: It consists of a set of n lists L_u, one per each node $u \in E$. Each list L_u contains all the neighbors of node u, that is, $L_u = \{v|(u, v) \in E\}$. Since the sum of the lengths of all the lists is m, this representation requires $m \log n$ bits for directed graphs. To represent undirected graphs, each undirected edge must be duplicated such that it appears in the adjacency list of the two nodes associated to that edge. Hence, the undirected graph representation using adjacency lists requires $2m \log n$ bits..

- Adjacency matrix representation: It consists in a boolean matrix $A = \{a_{i,j}\}$ of size $n \times n$, one row and one column for each node $u \in V$ where the cell $a_{u,v}$ is 1 if $(u, v) \in E$ and 0 otherwise. It requires n^2 bits for directed graphs, one bit for each matrix cell, and $n(n + 1)/2$ bits for undirected graphs, since the matrix is symmetric.

The adjacency lists representation is a better option when the graph is sparse, because it requires space proportional to the edges that are present in the graph. If the graph is dense, the adjacency matrix is a good choice, due to the compact representation of each edge, with just 1 bit. Besides the space tradeoff, these alternative representations of graphs also behave differently with the navigational operations. Finding all the neighbors of a node in an adjacency list consists in a simple reading of the list, in optimal time. With an adjacency matrix, an entire row must be scanned, which takes $O(n)$ time, or at best $O(n/\log n)$ in the RAM model. Whether there is an edge between two given nodes of the graph can be answered in constant time with an adjacency matrix, by just checking the associated cell of the matrix; however, with the adjacency lists representation, it requires time proportional to the degree of the source node.

In particular, a *Web graph* is a directed graph that contains a node for each Web page and there exists a directed edge (p, q) if and only if page p contains a hyperlink to page q. Then, a Web page q is a direct neighbor of a Web page p if p contains a link pointing to q and the reverse neighbors of a Web page p are all those Web pages that have a link pointing to p. Therefore, we can also define the adjacency matrix of a Web graph of n pages as a square matrix $\{a_{ij}\}$ of size $n \times n$, where each row and each column represents a Web page. Cell $a_{p,q}$ is 1 if there is a hyperlink in page p towards page q, and 0 otherwise. As on average there are about 15 links per Web page, this matrix is extremely sparse.

It is customary in compressed Web graph representations to assume that page identifiers are integers, which correspond to their position in an array of URLs. The space for that array is not accounted for, as it is independent of the Web graph compression method. Moreover, it is assumed that URLs are alphabetically sorted, which naturally puts together the pages of the same domains, and thus locality of reference translates into closeness of page identifiers. We follow this assumption in the application of our method, explained in the next chapter.

Most of the state-of-the-art techniques achieve compact representations of Web graphs by explicitly exploiting their statistical properties [BV04], such as:

- Skewed distribution: In- and out-degrees of the nodes of a Web graph are distributed according to power laws [BKM$^+$00]. The probability that a Web page has i links is $1/i^\theta$ for some parameter $\theta > 0$. Several experiments give rather consistent values of $\theta = 2.1$ for the in-degree distribution, and $\theta = 2.72$ in the case of the out-degree.

- Locality of reference: Most of the links of a Web graph are *navigational* links to Web pages of the same site ("home", "next", "previous", etc.). If Web pages are sorted alphabetically by URL, most pages will have links to pages with close identifier numbers. This permits the usage of gap encoding techniques.

- Similarity of the adjacency lists: The set of neighbors of a page is usually very similar to the set of neighbors of some other page. For instance, Web pages of a certain site often share many navigational links (for example, if they have a common menu). This peculiarity can be exploited to achieve compression by using a reference to a similar list and enumerating the differences as a list of edits. This characteristic is also known as *copy property*.

The properties of Web graphs can also be visualized and exploited in their adjacency matrix:

- Due to the locality of reference and the alphabetically ordering of the URLs, many 1s are placed around the main diagonal (that is, page i has many pointers to pages nearby i).
- Due to the copy property (similarity of the adjacency lists), similar rows are common in the matrix.
- Due to skewness of distribution, some rows and columns have many 1s, but most have very few.

11.3 State of the art

We now describe the most important works in Web graph compression. They are focused on obtaining a compact representation of the Web that permits the efficient extraction of the direct neighbors of any Web page. The space requirements for these methods is commonly measured in bits per edge (bpe), that is, is computed using the number of bits that are necessary to operate with them in main memory divided by the number of edges of the Web graph.

11.3.1 Boldi and Vigna: WebGraph Framework

The most famous representative of the Web graph compressing trend is surely the *WebGraph Framework*, by Boldi and Vigna [BV04]. It is associated to the site `http://webgraph.dsi.unimi.it`, which by itself witnesses the level of maturity and sophistication that this research area has reached.

The WebGraph compression method is indeed the most successful member of the family of approaches to compress Web graphs based on their statistical properties [BBH$^+$98, BKM$^+$00, AM01, SY01, RSWW01, RGM03]. Boldi and Vigna's representation allows fast extraction of the neighbors of a page while spending just a few bits per link (about 2 to 6, depending on the desired navigation performance).

The WebGraph Framework includes, in addition to the algorithms for compressing and accessing Web graphs, a set of new instantaneous codes which are suitable for storing this type of graphs, since they are especially designed for distributions commonly found when compressing Web graphs. It also includes data sets for very large graphs and a complete documented implementation in Java, with a clearly defined API to facilitate the use and set-up for the experimental evaluation of their technique.

The WebGraph method represents the adjacency lists of a Web graph by exploiting their similarity by *referentiation*: since URLs that are close in lexicographic order are likely to have similar successor lists (as they belong to the same site, and probably to the same level of the site hierarchy), they represent an adjacency list as an edit list. They use an integer as a reference to a node having a similar list, and a bit string that indicates the successors that are common to both lists. They also include a list of extra nodes for the remaining nodes that are not included in the reference list. Their representation of adjacency lists uses differential compression and some other techniques in order to obtain better space compression.

An example of the referentiation used in Boldi and Vigna's method is shown in Tables 11.1 and 11.2. The first table illustrates the plain representation of the adjacency lists of some nodes of the Web graph, where the first column details the node identifier, the second column indicates the outdegree of that node, that is, the number of direct neighbors of the node, and the third column displays the complete list of direct neighbors. As we can see, the adjacency lists of nodes 15, 16 and 18 are very similar, so the adjacency lists of nodes 16 and 18 will be represented by means of the adjacency list of node 15. Hence, Table 11.2 shows how the plain representation of the whole adjacency list for each node is replaced by the information of the reference node, the copy list and the extra nodes list. Since the adjacency lists of nodes 16 and 18 are represented via copy lists of the adjacency list of node 15, the third column stores the reference node (15) as a differential value. The fourth column of Table 11.2 shows the copy lists, that is, the bit string indicating which elements of the referred adjacency list are present in the adjacency list of the current node. The neighbors of the current node that are not included in the adjacency list of the referred node are included in the list of extra nodes in the last column of the table. For instance, the reference node used for the representation of the adjacency list of node 18 is node 15, since the difference valued stored to indicate it is 3, as we can see in the third column of Table 11.2. In addition, the edit list at column 4 indicates which links are shared among the adjacency list of node 15 and the adjacency list of node 18. Since only the first four bits are set, only the first four links of the adjacency list of node 15 are common to the adjacency list of node 18, that is, node 18 has links to nodes 13, 14, 16 and 17[1]. In addition, the last column of the table indicates that node 50 is also a neighbor of node 18.

The method uses two parameters: a *window size W* and the *maximum refer-*

[1] We need to previously obtain the adjacency list of node 15 in an analogous way. However, the adjacency list of node 15 is easier to retrieve since it does not use any reference node. This fact can be extracted from the table, as it indicates that the reference node is itself. Therefore, no copy list is stored and the complete adjacency list is explicitly enumerated in the "Extra nodes" column.

Node	Outdegree	Successors
...	...	...
15	11	13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	15, 16, 17, 22, 23, 24, 315, 316, 317, 3041
17	0	
18	5	13, 15, 16, 17, 50
...	...	...

Table 11.1: Adjacency lists for some nodes of a graph.

Node	Outd.	Ref.	Copy list	Extra nodes
...	...	...	...	...
15	11	0		13, 15, 16, 17, 18, 19, 23, 24, 203, 315, 1034
16	10	1	01110011010	22, 316, 317, 3041
17	0			
18	5	3	11110000000	50
...	...	...	...	...

Table 11.2: Representation of the adjacency lists using copy lists.

ence count R. That is, the successor lists of the last W nodes are considered as possible references, except those which would cause a recursive reference of more than R chains. The parameter R is essential for deciding the tradeoff between compression time and compression ratio, whereas W only affects the tradeoff between compression time and compression ratio.

11.3.2 Claude and Navarro: Re-Pair Based Compression

More recently, Claude and Navarro [CN10c] showed that most of the properties of Web graphs are elegantly captured by applying Re-Pair compression [LM00] on the adjacency lists. Their technique offers better space/time tradeoffs than WebGraph, that is, they offer faster navigation than WebGraph when both structures use the same space. Yet, WebGraph is able of using less space if slower navigation can be tolerated. In addition, the Re-Pair based compression can be adapted to work well in secondary memory.

Claude and Navarro use an approximate version of the original linear-time Re-Pair technique [LM00], which works on any sequence and uses very little memory on top of the sequence they want to compress. Therefore, they concatenate the adjacency lists of the Web graph and since Re-Pair is a phrase-based compressor, the regularities presented in the adjacency lists of the Web graph are exploited in order to compress the graph.

Re-Pair is a grammar-based compression algorithm consisting of repeatedly finding the most frequent pair of symbols in a sequence of integers and replacing it with a new symbol, until no more replacements are convenient. It identifies the most frequent pair in the sequence and then adds a new rule in the grammar dictionary to replace that pair by the new symbol created.

In their proposal [CN10c] they use this technique to represent a Web graph G. They create a sequence of integers $T(G)$, where $T(G)$ is the concatenation of the representations of all the adjacency lists. The adjacency list of node v_i is defined as $T(v_i) = \overline{v}_i v_{i_1} v_{i_2} v_{i_3} \ldots v_{i_{r_i}}$, where $\overline{v}_i$ is a unique special identifier that marks the beginning of the adjacency list and is not present in any other list and $v_{i_j}, 1 \leq j \leq r_i$, are the nodes pointed from v_i. Then, Re-Pair technique is used over the sequence $T(G)$ to obtain compression. The special marks $\overline{v}_i$ are not substituted by any other symbol, since they appear just once, so they still mark the beginning of each adjacency list in the Re-Pair compressed sequence. This allows direct navigation in optimal time, since it involves a simple successive extraction of the symbols from the beginning of the compressed adjacency lists, but not reverse navigation (that is, finding the pages that point to a given page), which must be carried out with not so efficient searches over the sequence.

In a more recent proposal [CN10b], Claude and Navarro modified their representation to also allow for reverse navigation. They combine grammar-based compression with concepts of binary relations. A graph $G = (V, E)$ can be regarded as a binary relation on $V \times V$ such that techniques as Barbay *et al.* [BGMR06, BHMR07] for binary relations can be used to support forward and reverse traversal operations. Using these techniques directly over the graph does not obtain good results. However, Re-Pair compression on graphs can also be regarded as the decomposition of the graph binary relation into two: *i)* nodes are related to the Re-Pair symbols that conform their compressed adjacency list, and *ii)* Re-Pair symbols are related to the graph nodes they expand to. Hence, they represent the graph as the composition of these two binary relations, using the technique of Barbay *et al.* [CN10b] over the compressed $T(G)$ sequence and the compressed sequence of Re-Pair symbols of the dictionary of rules (which is represented using some compression techniques [GN07]). Direct neighbors of a node can be retrieved by finding all the Re-Pair symbols that conform its adjacency list (first relation) and then the graph nodes each such symbol expands to (second relation). On the other hand, reverse neighbors can be obtained by first finding all the Re-Pair symbols (nonterminals) that expand to the node (second relation), and then, for each such symbol, all the nodes having an adjacency list where the symbol participates (first relation).

Depending on the compact data structure used to represent the sequence and carry out the rank and select operations over it, they obtain an interesting space/time tradeoff. They present two alternatives: `Re-Pair WT`, which uses a wavelet tree, obtains very compact spaces and reasonable navigation times, and `Re-Pair GMR`, which uses a GMR representation [GMR06], obtains an efficient navigation and occupies more space. The wavelet tree and GMR data structures to represent sequences were explained in Section 2.3.2.

11.3.3 Asano, Miyawaki and Nishizeki

Asano *et al.* [AMN08] achieve even less than 2 bits per link by explicitly exploiting regularity properties of the adjacency matrix of the Web graphs, such as horizontal, vertical, and diagonal runs. In exchange for achieving much better compression, their navigation time is substantially higher, as they need to uncompress full domains in order to find the neighbors of a single page.

They obtain very compact space by representing differently the *intra-host links* and the *inter-host links*. The first links, intra-host, are links between two pages in the same host, whereas an inter-host link is a link between two pages in distinct hosts. Their method exploits the fact that there are many more intra-host links than inter-host links.

To represent all the intra-host links inside one host, they represent only the 1's of the adjacency matrix of that host using six types of blocks. Each type of block, consisting of several consecutive 1's in the matrix, corresponds to some kind of locality:

- Blocks with just one isolated 1, called *singleton* blocks. They do not represent any kind of locality.

- A *horizontal block* consists of two or more horizontally consecutive 1-elements. Horizontal blocks are generated due to the fact that pages often link to consecutive pages of the same host, that is, Web pages with similar URLs.

- A *vertical block* consists of two or more vertically consecutive 1-elements. Pages of the same host often share a link to a common page, for instance, to the home page of the domain.

- An *L-shaped block* is the union of a horizontal block and a vertical block sharing the upper leftmost 1-element. We can find this type of block, for instance, when there is an index page that contains links to some consecutive pages, and those consecutive pages have a link to return to the index page.

- A *rectangular block* is a submatrix where all the elements are 1's and the submatrix has more than one consecutive row and more than one consecutive column. It consists in a combination of the localities represented with horizontal and vertical blocks, where several consecutive pages have intra-host links to common consecutive pages.

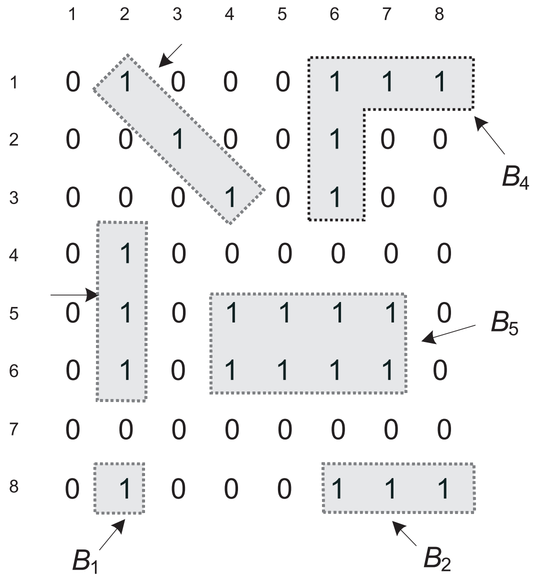
- A *diagonal block* consists of two or more 1-elements downward diagonally consecutive from upper left to lower right. This pattern can be found when a navigational link "next" (or "previous") is present in several consecutive pages.

Figure 11.1 illustrates the different types of blocks that appear in the adjacency matrix of a host: singleton blocks, such as B_1, horizontal blocks such as B_2, vertical blocks such as B_3, L-shaped blocks such as B_4, rectangular blocks such as B_5 or diagonal blocks such as B_6.

Hence, this technique represents the adjacency matrix of each host as a list of *signatures* of these blocks, indicating for each block its type, its beginning element and its dimension (which is represented differently depending on the type of the block). More specifically, they represent a block B in the adjacency matrix A of a host by a quadruplet $sig(B) = (br(B), bc(B), type(B), d(B))$, where:

- $br(B)$ and $bc(B)$ define the position of the *beginning element* of B in the matrix, that is, the upper leftmost element of a block B, which is denoted by $b(B)$. $br(B)$ is the row number of $b(B)$, and $bc(B)$ is the column number, then $b(B) = A_{br(B),bc(B)}$. For the example of Figure 11.1, the beginning element of block B_2 is $b(B_2) = A_{8,6}$, since the upper leftmost element is placed at row 8 and column 6.

- $type(B)$ denotes the *type* of a block B. For instance, $type(B_3) = Vertical$.



$$\begin{array}{c|cccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ \hline 1 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 2 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 3 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 6 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 0 \\ 7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 8 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \end{array}$$

Figure 11.1: Several blocks presented in the adjacency matrix.

- $d(B)$ corresponds to the *dimension* of block B. The dimension $d(B)$ of an L-shaped or rectangular block B is defined to be an ordered pair $(er(B)-br(B)+1, ec(B)-bc(B)+1)$, where $er(B)$ is the row number of the lowest element in B and $ec(B)$ is the column number of the rightmost one. The dimension $d(B)$ of block B of the other types is defined to be the number of elements in B. A singleton block can be represented without the dimension, because the dimension of every singleton block is 1. For instance, the dimension of the vertical block B_2 is $d(B_2) = 3$, since it has 3 ones, whereas the dimension of the rectangular block B_5 consists of the pair $(2, 4)$. This dimension is computed by noticing that the beginning element is at row $br(B_5) = 5$ and column $bc(B_5) = 4$, the lowest element of the block is at row $er(B_5) = 6$ and the rightmost element at column $ec(B_5) = 7$; hence, the formula above gives us the pair $(6-5+1, 7-4+1) = (2, 4)$.

For the example of Figure 11.1, the host would be represented as a list of all the blocks, where each block is represented as follows:

- $sig(B_1) = (8, 2, Singleton)$
- $sig(B_2) = (8, 6, Horizontal, 3)$
- $sig(B_3) = (4, 2, Vertical, 3)$

- $sig(B_4) = (1, 6, Lshaped, (3, 3))$
- $sig(B_5) = (5, 4, Rectangular, (2, 4))$
- $sig(B_6) = (1, 2, Diagonal, 3)$

Inter-host links are also compressed with this technique by regarding them as intra-host links. For this sake, new local indices are assigned to the destinations of inter-host links. If there are n pages in a host and that host has m inter-host links, the new local indexes $n + i - 1$ for $i = 1 \ldots m$ will consecutively assigned to replace the m original indices of the destinations of the inter-host links. Hence, the method constructs a new intra-destination list for each page in the host, which is the union of two lists: one is the intra-destination list of pages, and the other is the list of new local indices for the inter-host links. For each host, it is necessary to store, in a table, a pair of the new local index and original index of the destination of each inter-host link.

Therefore, the final representation of the Web graph is obtained by the compression of intra-host links and inter-host links all together for each host, where the input is the new intra-destination lists for each host. Their technique obtains better compression ratio than Boldi and Vigna, but access times to the neighbors list of a node are considerably higher. However, their experimental evaluation only includes the results for very small graphs.

11.3.4 Buehrer and Chellapilla: Virtual Node Miner

Buehrer and Chellapilla [BC08] propose a Web graph compression technique that not only obtains good compression ratios but also permits some community discovery, since it can find global patterns in the Web graph. In addition, their method, called *Virtual Node Miner*, has some other interesting properties, such as not requiring a particular ordering of the nodes of the graph, and supporting several available coding schemes. Moreover, it is highly scalable and support incremental updates.

In the context of the Web, a community can be seen as a group of pages related to a common interest. Regarding the Web graph, communities have been associated with the existence of a locally dense subgraph, and more specifically, they are commonly abstracted as a set of pages that form a complete bipartite graph or biclique[2]. Therefore, it is very frequent to find patterns inside Web graphs such that a group of nodes points to another set of nodes. This particularity is exploited by the authors of this work to obtain a compressed representation of a Web graph.

[2] A bipartite graph is a graph whose nodes can be divided into two disjoint sets U and V such that every edge connects a vertex in U to one in V; that is, U and V are independent sets. A biclique is a complete bipartite graph, where every vertex of the first set is connected to every vertex of the second set.

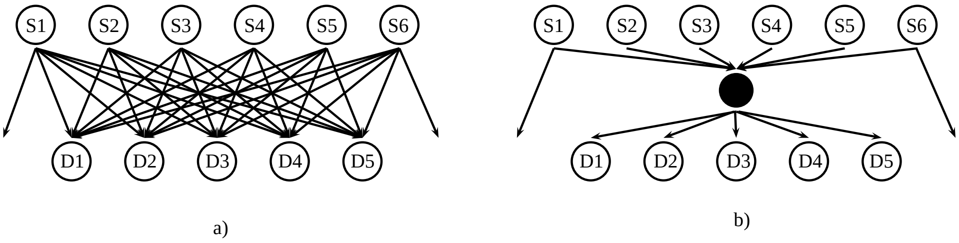


Figure 11.2: Several links in a) are compressed into one virtual node in b).

The idea of the *Virtual Node Miner* method consists in searching for communities in the Web by finding bicliques inside the Web graph, and representing them in a compact way. Figure 11.2 (left) illustrates an excerpt of a Web graph containing a complete bipartite subgraph, where six nodes (S1, ..., S6) share links to five common destination nodes (D1, ..., D5). Instead of representing all the intra-links of the community, their algorithm reduces the number of edges by generating a new node, called *virtual node*. Figure 11.2 (right) shows how this artificial node assembles the ingoing and outgoing links of the community such that an important space reduction is obtained. For the example of the figure, just 11 links are represented instead of the 30 original ones. In many cases for real Web graphs, they can represent thousands of edges with a single link to a virtual node.

They address the problem of finding those virtual nodes by using a common data mining algorithm, the *frequent itemset mining* approach [AIS93], which we will not explain here since it escapes the objectives of this thesis. A previous step must be performed such that mining Web graph, consisting of hundreds of millions of nodes, becomes possible. Hence, *Virtual Node Miner* first clusters similar vertices in the graph and then it finds patterns in those clusters. When those patterns are found, it removes the links involved and replaces them with virtual nodes. The algorithm repeats this procedure until there are no more patterns to discover. Finally, a coding scheme is used to encode the remaining edges.

Their experimental evaluation indicates that their technique achieves a 10- to 15-fold compression on most real word Web graph data sets, using 1.5 to 3 bpe. Moreover, it shows that the algorithm is scalable. For instance, their method can compress a 3 billion edge graph in 2.5 hours on a single machine.

11.3.5 Apostolico and Drovandi: Compression by Breadth First Search

Apostolico and Drovandi presented in 2009 a method for graph compression that permits a fast retrieval of the information of the nodes [AD09]. The idea of their paper is to order the nodes of the Web graph following a Breadth First Search

(BFS) strategy instead of using the lexicographic order, while still retaining the main features of the Web graphs (locality and similarity). Hence, they do not assume any previous knowledge of the Web graph (many other works from the literature are based on the lexicographic ordering of URLs) and their algorithm depends only on their topological structure.

They compress the Web graph using a two-phases algorithm. During the first phase, they perform a breadth-first traversal of Web graph and index each node according to the order in which it is expanded. Hence, two connected nodes are likely to be assigned close index values. In addition, since two adjacent nodes of the Web graph often share many neighbors, the similarity property of the adjacency lists is also captured with this method. They separately compress consecutive chunks of l nodes, where l is a parameter called *compression level*. During the second phase, the adjacency list of each node is encoded exploiting all the redundancies presented (references to identical rows, gap encoding for close indexes, etc).

They obtain a very compact space (about 1 to 4 bpe), smaller than Asano *et al.*, maintaining an average retrieval time comparable to Boldi and Vigna. In addition, they introduce a very efficient query to determine whether two nodes are connected, that is, if one page p has a link to a page q without the need to always extract the adjacency list for p. The average time for this operation is less than 60% of the retrieval time of the whole adjacency list.

11.4 Our goal

As we have seen, there are several proposals to compress the graph of the Web that obtain compact spaces by different approaches such as extracting patterns from the graph or exploiting the similarities of the adjacency lists.

Some of these techniques are focused only on achieving the most compact space possible, whereas most of them allow the efficient extraction of the direct neighbors of any Web page. However, more sophisticated navigation is desirable for several Web analyses. For instance, these methods do not extract so efficiently the reverse neighbors of a Web page, which is an interesting operation for several applications. The standard approach to achieve this direct and reverse navigation is to represent the graph and its transpose, such that the reverse navigation is answered using the direct neighbors retrieval over the transposed graph. Yet, this approach basically doubles the space needed for the Web graph representation and the redundancy between both graphs is not exploited. Hence, our goal is to intrinsically capture the properties of the Web graph to solve direct and reverse navigation efficiently over the compressed representation of the graph without also representing its transpose.

In the following chapters we present our proposal of a new compression method for Web graphs that achieves a very compact space and enables the extraction of both direct and reverse neighbors of a Web page in a uniform way, in addition to supporting other navigation operations over the Web graph.

Chapter 12

Our proposal: k^2-tree representation

In this chapter we present a new compact representation for a Web graph that takes its adjacency matrix and builds a tree that can be stored in a compact space. It supports the classical operations, such as retrieving all the pages that are pointed by a given Web page, without the need of decompressing all the Web graph. In addition, it allows for reverse neighbor retrieval and extra functionality such as range searches or retrieval of single links.

The chapter is organized as follows. Section 12.1 describes the tree representation conceptually, including the basic operations supported by our representation and how they are carried out over the tree. Section 12.2 describes the data structures and algorithms used to efficiently store and manage the tree representation. Section 12.3 proposes a variation of the method that improves both time and space requirements. Section 12.4 describes some extra functionalities supported by our proposal and analyzes their time complexity. Finally, in Section 12.5 we propose some alternatives to the k^2-tree technique whose aim is to improve the efficiency of the method.

12.1 Conceptual description

In this section, we present a tree-shaped representation of the adjacency matrix of a Web graph that supports the basic navigation over the graph, such as retrieving the list of direct or reverse neighbors. We first describe conceptually our proposal, called k^2-tree, detailing how it is built, and finally, we show how that basic navigation is supported in the tree.

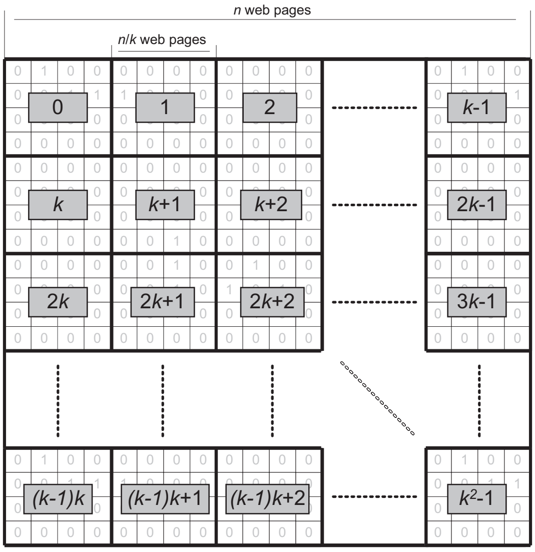
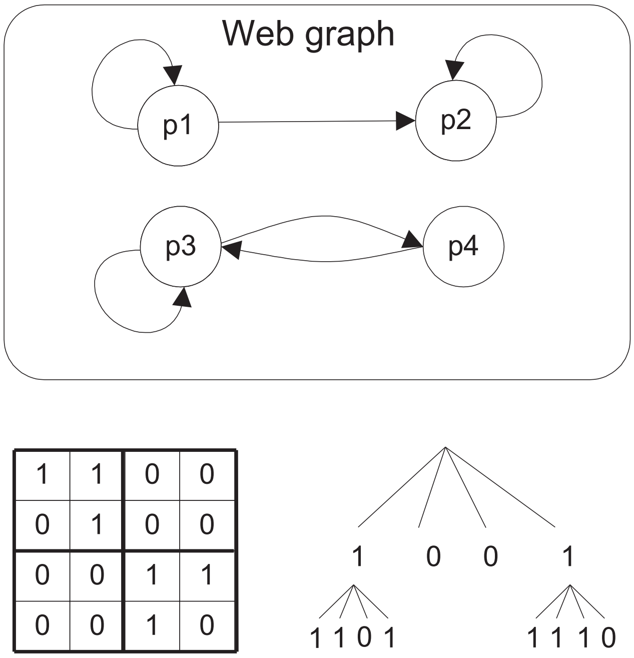


Figure 12.1: Subdivision of the adjacency matrix into k^2 submatrices, indicating their ordering.

We propose a compact representation of the adjacency matrix that exploits its sparseness and clustering properties. The representation is designed to compress large matrix areas with all 0s into very few bits. We represent the adjacency matrix by a k^2-ary tree, which we call k^2*-tree*.

Assume for simplicity that the adjacency matrix of the Web graph is a square matrix of size $n \times n$, where n is a power of k, we will soon remove this assumption. Conceptually, we start dividing the adjacency matrix following a MX-Quadtree strategy [Sam06, Section 1.4.2.1] into k^2 submatrices of the same size, that is, k rows and k columns of submatrices of size n^2/k^2. Each of the resulting k^2 submatrices will be a child of the root node and its value will be 1 iff there is at least one 1 in the cells of the submatrix. A 0 child means that the submatrix has all 0s and therefore the tree decomposition ends there; thus 0s are leaves in our tree. The children of a node are ordered in the tree starting with the submatrices in the first (top) row, from left to right, then the submatrices in the second row from left to right, and so on, as shown in Figure 12.1.

Once the level 1 of the tree, which contains the children of the root node, has been built, the method proceeds recursively for each child with value 1. The procedure stops when we reach submatrices full of 0s, or when we reach a $k \times k$ submatrix of the original adjacency matrix, that is, we reach the last level of the tree. In this last level, the bits of the nodes correspond to the adjacency matrix cell values,



1	1	0	0
0	1	0	0
0	0	1	1
0	0	1	0

Figure 12.2: Representation of a Web graph (top) by its adjacency matrix (bottom left) and the k^2-tree obtained (bottom right).

following the node ordering we have previously defined. Hence, it is easy to see that the height of the k^2-tree is $h = \lceil \log_k n \rceil$, since we stop the tree construction of the $n \times n$ adjacency matrix when we reach a level with submatrices of size $k \times k$, subdividing the side of each square submatrix by k in each step.

Figure 12.2 illustrates a small Web graph consisting of 4 Web pages, p1, p2, p3 and p4. Its 4×4 adjacency matrix is shown in the bottom left part of the figure. At the bottom right part of the figure we illustrate the 2^2-tree built for this example. Its height is $h = \lceil \log_2 4 \rceil = 2$, where the level 1 corresponds to the children of the root node and level 2 contains the original cell values of the 2×2 submatrices of the adjacency matrix that are not full of zeroes. Following the ordering previously defined, those submatrices containing at least one 1 are the first one (top-left) and the fourth one (bottom-right).

We have previously assumed that n was a power of k. If n is not a power of k, we conceptually extend our matrix to the right and to the bottom with 0s, making it of width $n' = k^{\lceil \log_k n \rceil}$, that is, rounding up n to the next power of k, n'. This does not cause a significant overhead as our technique is efficient to handle large areas of 0s.

Note that, since the height of the tree is $h = \lceil \log_k n \rceil$, a larger k induces a shorter tree, with fewer levels, but with more children per internal node. Figures 12.3 and 12.4 show an example of the same adjacency matrix of a Web graph (we use the first 11×11 submatrix of graph CNR [BV04]), and how it is expanded to an $n' \times n'$ matrix for n' a power of $k = 2$ (Figure 12.3) and of $k = 4$ (Figure 12.4). The figures also show the k^2-trees corresponding to those k values.

As we can see, each node contains a single bit of data: 1 for the internal nodes and 0 for the leaves, except for the last level of the tree, where all the nodes are leaves and they represent some bit values of the adjacency matrix. Level 0 corresponds to the root and its k^2 children are represented at level 1. Each child is a node and therefore it has a value 0 or 1. All internal nodes in the tree (i.e., with value 1) have exactly k^2 children, whereas leaves (those nodes with value 0 or at the last level of the tree) have no children. Notice that the last level of the tree represents cells in the original adjacency matrix, but most empty cells in the original adjacency matrix are not represented in this level because, where a large area with 0s is found, it is represented by a single 0 in a higher level of the tree.

12.1.1 Navigating with a k^2-tree

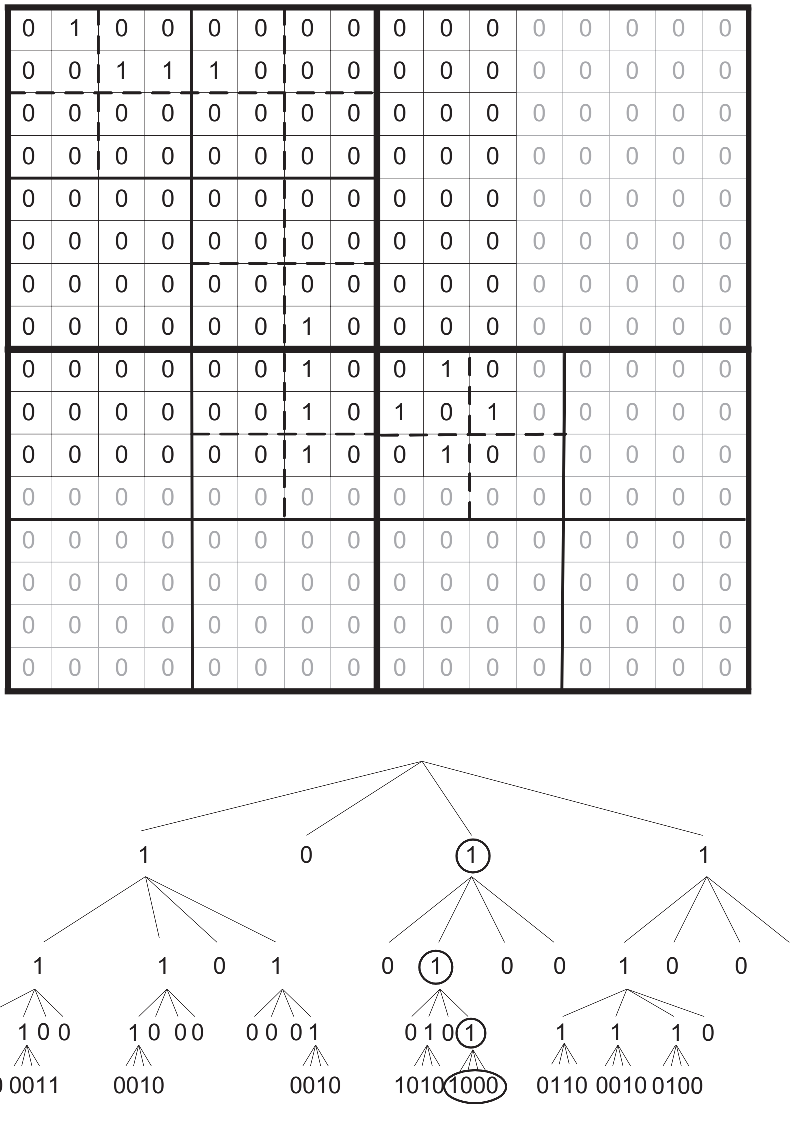
In this section we explain how the basic navigation is carried out using the k^2-tree representation of the Web graph, that is, how the direct and reverse neighbors of a certain page are obtained.

Direct neighbors To obtain the pages pointed by a specific page p, that is, to find direct neighbors of page p, we need to find the 1s in row p of the matrix.

We will proceed with a top-down traversal over the tree representation, starting at the root and travelling down the tree until we reach the leaves, choosing exactly k children of each node at each level. We will illustrate this procedure with an example and then we will generalize the algorithm in the next section.

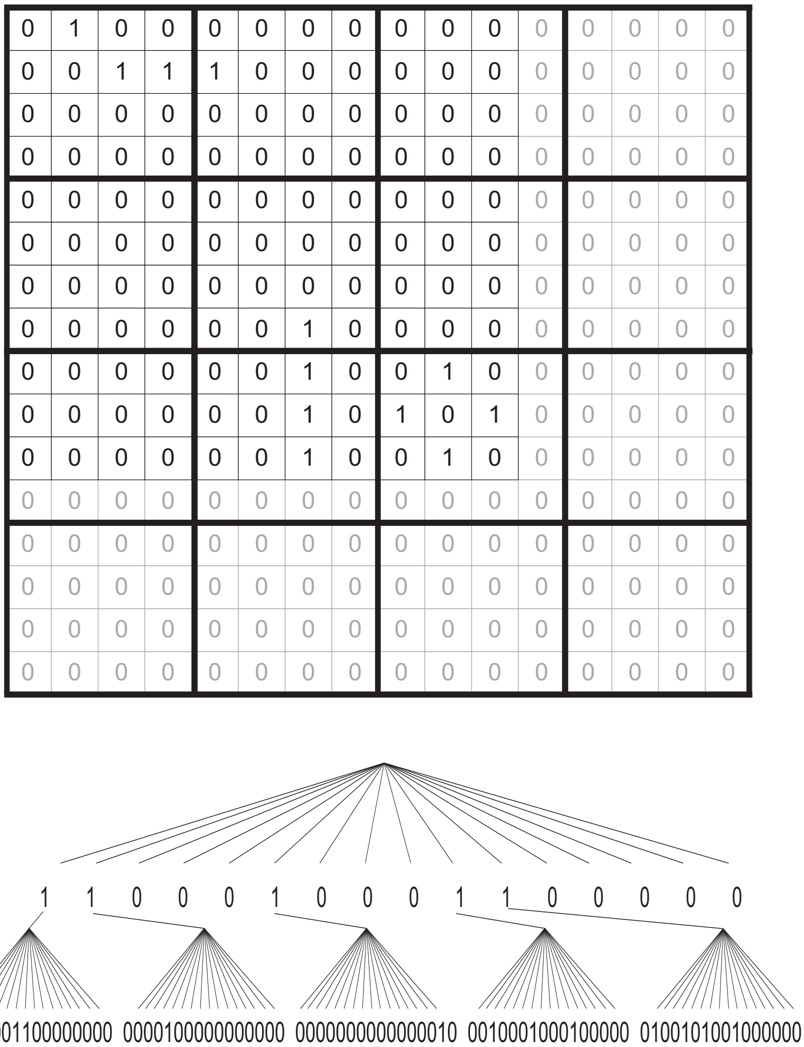
Example We want to find the pages pointed by the first page in the example of Figure 12.2, that is, find the 1s of the first matrix row. We start at the root of the 2^2-tree and compute which children of the root node overlap with the first row of the matrix. These are the first two children, that is, the two submatrices of the top, so we traverse down the tree to these two children:

- The first child is a 1, thus it has children. To figure out which of its children are useful we repeat the same procedure. We compute in the corresponding submatrix (the one at the top left corner) which of its children represent cells overlapping with the first row of the original matrix. These are the first and the second children. They are placed at the last level of the tree and their



0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0
0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 12.3: Expansion and subdivision of the adjacency matrix (top) and resulting tree (bottom) for $k = 2$. The bits marked with circles are used in Section 12.2 to illustrate an example of navigation using the k^2-tree.



0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0
0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 12.4: Expansion and subdivision of the adjacency matrix (top) and resulting tree (bottom) for $k = 4$.

values are both 1. These two 1s represent two direct neighbors of the Web page of the query, and as they are at the first and second columns of the original matrix (according to the path traversed from the root node to the leaves as we will explain in the next section), they represent the Web pages 1 and 2.

- The second child of the root represents the second submatrix, but its value is 0. This means that all the cells of the adjacency matrix in this area are 0, so we stop the top-down traversal through this branch at this point.

Now we know that the Web page represented by this first row has a link to itself and another to page 2.

Reverse neighbors An analogous procedure retrieves the list of reverse neighbors. To obtain which pages point to page q, we need to locate which cells have a 1 in column q of the matrix. Thus, we carry out a symmetric algorithm, using columns instead of rows.

Let us illustrate the procedure again over the example in Figure 12.2. For instance, if we want to know the pages that point to the last page (placed at the rightmost column) we compute the children of the root node that represent submatrices overlapping with that column. These are the second and the fourth children, that is, the k rightmost submatrices of the adjacency matrix. The second child has value 0, therefore no pages in those rows point to the last page. The fourth child has a 1, therefore we compute which of its children represent submatrices overlapping cells in the last column; these are the second and the fourth. The second child, which is placed at the last level of the tree, has a 1 value, so it represents a reverse neighbor of the page q of the query, whereas the fourth child has a 0, so it does not represent any reverse neighbor of Web page q. Following the path from the root node to this 1 value we can know that it is the third row in the original adjacency matrix (this will be explained in the next section), so we can conclude that only page 3 points to the last page.

Summarizing, searching for direct or for reverse neighbors in the k^2-tree is completely symmetric. In either case we perform a top-down traversal of the tree, where the only difference is the formula to compute the children of each node used in the next step. If we want to search for direct(reverse) neighbors in a k^2-tree, we go down through k children forming a row(column) inside the matrix, more specifically, those submatrices that overlap with the row(column) of the Web page of the query.

12.2 Data structures and algorithms

Our data structure is essentially a compact tree of N nodes. There exist several such representations for general trees [Jac89a, MR01, BDM$^+$05, GRRR06], which asymptotically approach the information-theoretic minimum of $2N + o(N)$ bits. In our case, where there are only arities k^2 and 0, the information-theoretic minimum of $N + o(N)$ bits is achieved by a so-called "ultra-succinct" representation [JSS07] for general trees.

Our representation is much simpler, and similar to the so-called Level-Ordered Unary Degree Sequence (LOUDS) [Jac89a, DRR06], which consists in an unlabeled general tree representation that achieves the asymptotic optimum of two bits per node. Using LOUDS, the tree is represented reporting the degree of each node in (left-to-right) level-order. The degree sequence is encoded using unary codes, where a degree d is represented by the string 1^d0 (already explained in Section 3.1). Then, the encoded sequence of degrees has length $2n - 1$ bits for a tree with n nodes: $n - 1$ 1s (each node is associated with one 1 in the encoded representation of the degree of its parent, except for the root node) and n 0s (the 0 bit that ends all the degree representation with unary codes for all the nodes). The encoded sequence is accompanied by a rank and select directories, as explained in Section 2.3.1. This representation allows the basic navigation over the tree, such as the computation of parent, first child or next sibling, and also permits the access to children by number, previous siblings and counting of children. LOUDS tree representation would not achieve $N + o(N)$ bits if directly applied to our trees.

Our data structure can be regarded as a simplified variant of LOUDS for the case where arities are just k^2 and 0, following the strategy of C-tries [Mal76]. It achieves the information-theoretic minimum of $N+o(N)$ bits, provides the traversal operations we require (basically move to the i-th child, although also parent is easily supported) in constant time, and is simple and practical.

12.2.1 Data structures

We represent the whole adjacency matrix via the k^2-tree in a very compact way using two bit arrays:

T *(tree)*: stores all the bits of the k^2-tree except those in the last level. The bits are placed following a levelwise traversal: first the k^2 binary values of the children of the root node, then the values of the second level, and so on.

L *(last level leaves)*: stores the last level of the tree. Thus it represents the value of (some) original cells of the adjacency matrix.

We create over T an auxiliary structure that enables us to compute *rank* queries efficiently. In practice we use an implementation that uses 5% of extra space on top of the bit sequence and provides fast queries. Instead of that implementation, we can also use another alternative that requires 37.5% extra space and is much faster [GGMN05].

We do not need to perform *rank* operations over the bits in the last level of the tree, since *rank* operations are needed only to navigate through the compact representation of the tree, that is, to travel down from a node to its children; this is the practical reason to store them in a different bitmap (L). Thus the space overhead for *rank* is paid only over T.

Hence, the final representation of the Web graph using the k^2-tree technique over its adjacency matrix consists of the concatenation of those two bit arrays, $T : L$, and the extra structure to support *rank* operations over T efficiently.

12.2.1.1 Space analysis

Assume the graph has n pages and m links. Each link is a 1 in the matrix, and in the worst case it induces the storage of one distinct node per level, for a total of $\lceil \log_{k^2}(n^2) \rceil$ nodes. Each such (internal) node costs k^2 bits, for a total of $k^2 m \lceil \log_{k^2}(n^2) \rceil$ bits. However, especially in the upper levels, not all the nodes in the path to each leaf can be different. In the worst case, all the nodes exist up to level $\lfloor \log_{k^2} m \rfloor$ (only since that level there can be m different internal nodes at the same level). From that level, the worst case is that each of the m paths to the leaves is unique. Thus, in the worst case, the total space in bits is

$$\sum_{\ell=1}^{\lfloor \log_{k^2} m \rfloor} k^{2\ell} + k^2 m \left(\lceil \log_{k^2} n^2 \rceil - \lfloor \log_{k^2} m \rfloor \right) = k^2 m \left(\log_{k^2} \frac{n^2}{m} + O(1) \right).$$

This shows that, at least in a worst-case analysis, a smaller k yields less space occupancy. For $k = 2$ the space is $4m(\log_4 \frac{n^2}{m} + O(1)) = 2m \log_2 \frac{n^2}{m} + O(m)$ bits, which is asymptotically twice the information-theoretic minimum necessary to represent all the matrices of $n \times n$ with m 1s. In Chapter 13, which includes the experimental evaluation of the proposal, we will see that on Web graphs the space is much better than the worst case, as Web graphs are far from uniformly distributed.

Finally, the expansion of n to the next power of k can, in the horizontal direction, force the creation of at most k^ℓ new children of internal nodes at level $\ell \geq 1$ (level $\ell = 1$ is always fully expanded unless the matrix is all zeros). Each such child will cost k^2 extra bits. The total excess is $O(k^2 \cdot k^{\lceil \log_k n \rceil - 1}) = O(k^2 n)$ bits, which is usually negligible. The vertical expansion is similar.

12.2.2 Finding a child of a node

Our levelwise traversal satisfies the following property, which permits fast navigation to the i-th child of node x, $child_i(x)$ (for $0 \leq i < k^2$):

Lemma 12.1 *Let x be a position in T (the first position being 0) such that $T[x] = 1$. Then $child_i(x)$ is at position*[1] $rank(T, x) \cdot k^2 + i$ *of* $T : L$

Proof $T : L$ is formed by traversing the tree levelwise and appending the bits of the tree. We can likewise regard this as traversing the tree levelwise and appending the k^2 bits of the children of the 1s found at internal tree nodes. By the time node x is found in this traversal, we have already appended k^2 bits per 1 in $T[0, x - 1]$, plus the k^2 children of the root. As $T[x] = 1$, the children of x are appended at positions $rank(T, x) \cdot k^2$ to $rank(T, x) \cdot k^2 + (k^2 - 1)$.

Example To represent the 2^2-tree of Figure 12.3, arrays T and L have the following values:

$$T = 1011\ \ 1101\ \ 0100\ \ 1000\ 1100\ \ 1000\ \ 0001\ \ 0101\ \ 1110,$$
$$L = 0100\ \ 0011\ \ 0010\ \ 0010\ \ 1010\ \ 1000\ \ 0110\ \ 0010\ \ 0100.$$

In T each bit represents a node. The first four bits represent the nodes $0, 1, 2$ and 3, which are the children of the root. The following four bits represent the children of node 0. There are no children for node 1 because it is a 0, then the children of node 2 start at position 8 and the children of node 3 start at position 12. The bit in position 4, which is the fifth bit of T, represents the first child of node 0, and so on.

For the following, we mark with a circle the involved nodes in Figure 12.3. We compute where the second child of the third node is, that is, child 1 of node 2. If we compute *rank* until the position of the bit representing node 2, $rank(T, 2) = 2$, we obtain that there are 2 nodes with children until that position because each bit 1 represents a node with children. As each node has 4 children, we multiply by 4 the number of nodes to know where it starts. As we need the second child, this is $child_1(2) = rank(T, 2) * 2^2 + 1 = 2 * 4 + 1 = 9$. In position 9 there is a 1, thus it represents a node with children and its fourth child can be found at $child_3(9) = rank(T, 9)*2^2+3 = 7*4+3 = 31$. Again it is a 1, therefore we can repeat the process to find its children, $child_0(31) = rank(T, 31) * 2^2 + 0 = 14 * 4 + 0 = 56$. As $56 \geq |T|$, we know that the position belongs to the last level, corresponding to offset $56 - |T| = 56 - 36 = 20$ (to 23) in L.

12.2.3 Navigation

To find the direct(reverse) neighbors of a page $p(q)$ we need to locate which cells in row a_{p*} (column a_{*q}) of the adjacency matrix have a 1. We have already explained that these are obtained by a top-down tree traversal that chooses k out of the k^2

[1] $rank(T, x)$ stands for $rank_1(T, x)$ for now on, that is, it returns the number of times bit 1 appears in the prefix $T_{1,x}$

children of a node, and also described how to obtain the i-th child of a node in our representation. The only missing piece is the formula that maps global row numbers to the children number at each level.

Recall $h = \lceil \log_k n \rceil$ is the height of the tree. Then the nodes at level ℓ represent square submatrices of size $k^{h-\ell}$, and these are divided into k^2 submatrices of size $k^{h-\ell-1}$. Cell (p_ℓ, q_ℓ) at a matrix of level ℓ belongs to the submatrix at row $\lfloor p_\ell / k^{h-\ell-1} \rfloor$ and column $\lfloor q_\ell / k^{h-\ell-1} \rfloor$. For instance, the root at level $\ell = 0$ represents the whole square matrix of width $k^h = n$.

Let us call p_ℓ the relative row position of interest at level ℓ. Clearly $p_0 = p$ (since we have the original matrix at level 0), and row p_ℓ of the submatrix of level ℓ corresponds to children number $k \cdot \lfloor p_\ell / k^{h-\ell-1} \rfloor + j$, for $0 \leq j < k$. The relative position in those children is $p_{\ell+1} = p_\ell \bmod k^{h-\ell-1}$. Similarly, column q corresponds to $q_0 = q$ and, in level ℓ, to children number $j \cdot k + \lfloor q_\ell / k^{h-\ell-1} \rfloor$, for $0 \leq j < k$. The relative position at those children is $q_{\ell+1} = q_\ell \bmod k^{h-\ell-1}$.

For instance, assume that we want to obtain the direct neighbors of Web page 10 of the Web graph represented in Figure 12.3. This Web page is represented at row $p_0 = 10$ at level $\ell = 0$, since the whole adjacency matrix is considered at this level. When $\ell = 1$ the relative position of Web page 10 inside the two submatrices of size 8×8 of the bottom of the matrix is $p_1 = 10 \bmod 8 = 2$. The relative row inside the submatrices of size 4×4 that overlap with row 10 at level $\ell = 2$ is $p_2 = 2 \bmod 4 = 2$, and finally, the relative position of row 10 inside the submatrices of size 2×2 that overlap with the row at level $\ell = 3$ is $p_3 = 2 \bmod 2 = 0$.

The algorithms for extracting direct and reverse neighbors are described in Algorithms 12.1 and 12.2. The one for direct neighbors is called **Direct**$(k^h, p, 0, -1)$, where the parameters are: current submatrix size, row of interest in current submatrix, column offset of the current submatrix in the global matrix, and the position in $T : L$ of the node to process (the initial -1 is an artifact because our trees do not represent the root node). Values T, L, and k are global. The one for reverse neighbors is called **Reverse**$(k^h, q, 0, -1)$, where the parameters are the same except that the second is the column of interest and the third is the row offset of the current submatrix. It is assumed that n is a power of k and that $rank(T, -1) = 0$.

We note that the algorithms output the neighbors in order. Although we present them in recursive fashion for clarity, an iterative variant using a queue of nodes to process turned out to be slightly more efficient in practice.

12.2.3.1 Time analysis

The navigation time to retrieve a list of direct or reverse neighbors has no worst-case guarantees better than $O(n)$, as a row $p - 1$ full of 1s followed by p full of 0s could force a **Direct** query on p to go until the leaves across all the row, to return nothing.

Algorithm 12.1: **Direct**(n, p, q, z) returns direct neighbors of element x_p

```
if $z \geq |T|$ then /* last level */
    if $L[z - |T|] = 1$ then output $q$
else/* internal node */
    if $z = -1$ or $T[z] = 1$ then
        $y = rank(T, z) \cdot k^2 + k \cdot \lfloor p/(n/k) \rfloor$
        for $j = 0 \ldots k-1$ do
            Direct$(n/k, p \bmod (n/k), q + (n/k) \cdot j,\ y + j)$
        end
    end
end
```

Algorithm 12.2: **Reverse**(n, q, p, z) returns reverse neighbors of element x_q

```
if $z \geq |T|$ then /* last level */
    if $L[z - |T|] = 1$ then output $p$
else/* internal node */
    if $z = -1$ or $T[z] = 1$ then
        $y = rank(T, z) \cdot k^2 + \lfloor q/(n/k) \rfloor$
        for $j = 0 \ldots k-1$ do
            Reverse$(n/k, q \bmod (n/k), p + (n/k) \cdot j, y + j \cdot k)$
        end
    end
end
```

However, this is unlikely. Assume the m 1s are uniformly distributed in the matrix. Then the probability that a given 1 is inside a submatrix of size $(n/k^\ell) \times (n/k^\ell)$ is $1/k^{2\ell}$. Thus, the probability of entering the children of such submatrix is (brutally) upper bounded by $m/k^{2\ell}$. We are interested in k^ℓ submatrices at each level of the tree, and therefore the total work is on average upper bounded by $m \cdot \sum_{\ell=0}^{h-1} k^\ell / k^{2\ell} = O(m)$. This can be refined because there are not m different submatrices in the first levels of the tree. Assume we enter all the $O(k^t)$ matrices of interest up to level $t = \lfloor \log_{k^2} m \rfloor$, and from then on the sum above applies. This is

$$O\left(k^t + m \cdot \sum_{\ell=t+1}^{h-1} \frac{k^\ell}{k^{2\ell}}\right) = O\left(k^t + \frac{m}{k^t}\right) = O\left(\sqrt{m}\right)$$

time. This is not the ideal $O(m/n)$ (average output size), but much better than $O(n)$ or $O(m)$.

Again, if the matrix is clustered, the average performance is indeed better than under uniform distribution: whenever a cell close to row p forces us to traverse the tree down to it, it is likely that there is a useful cell at row p as well. This can be observed in the experimental evaluation described in Chapter 13.

12.2.4 Construction

Assume our input is the $n \times n$ adjacency matrix. Construction of our tree is easily carried out bottom-up in linear time and optimal space (that is, using the same space as the final tree).

Our procedure builds the tree recursively. It consists in a depth-first traversal of the tree that outputs the a bit array T_ℓ for each level of the tree. If we are at the last level, we read the k^2 corresponding matrix cells. If all are zero, we return zero and we do not output any bit string, since that zone of zeroes is not represented with any bit at the final representation of the graph; otherwise we output their k^2 values and return 1. If we are not at the last level, we make the k^2 recursive calls for the children. If all return zero, we return zero, otherwise we output the k^2 answers of the children and return 1.

The output for each call is stored separately for each level, so that the k^2 bits that are output at each level are appended to the corresponding bit array T_ℓ. As we fill the values of each level left-to-right, the final T is obtained by concatenating all levels but the last one, which is indeed L.

Algorithm 12.3 shows the construction process. It is invoked as **Build**$(n, 1, 0, 0)$, where the first parameter is the submatrix size, the second is the current level, the third is the row offset of the current submatrix, and the fourth is the column offset. After running it we must carry out $T = T_1 : T_2 : \ldots : T_{h-1}$ and $L = T_h$.

Algorithm 12.3: **Build**(n, ℓ, p, q), builds the tree representation

```
$C$ = empty sequence
for $i = 0 \ldots k-1$ do
    for $j = 0 \ldots k-1$ do
        if $\ell = \lceil \log_k n \rceil$ then /* last level */
            $C = C : a_{p+i,q+j}$
        else/* internal node */
            $C = C :$ Build$(n/k, \ell+1, p+i\cdot(n/k), q+j\cdot(n/k))$
        end
    end
end
if $C = 0^{k^2}$ then return 0
$T_\ell = T_\ell : C$
return 1
```

The total time is clearly linear in the number of elements of the matrix, that is, $O(n^2)$. However, starting from the complex matrix is not feasible in practice for real Web graphs. Hence, we use instead the adjacency lists representation of the matrix, that is, for each Web page p we have the list of Web pages q such that p has a link pointing to q. By using the adjacency lists we can still achieve the same time by setting up n cursors, one per row, so that each time we have to access a_{pq}

we compare the current cursor of row p with value q. If they are equal, we know $a_{pq} = 1$ and move the cursor to the next node of the list for row p. Otherwise we know $a_{pq} = 0$. This works because all of our queries to each matrix row p are increasing in column value.

In this case, when the input consists of the adjacency list representation of the graph, we could try to achieve time proportional to m, the number of 1s in the matrix. For this sake we could insert the 1s one by one into an initially empty tree, building the necessary part of the path from the root to the corresponding leaf. After the tree is built we can traverse it levelwise to build the final representation, or recursively to output the bits to different sequences, one per level, as before. The space could still be $O(k^2 m(1+\log_{k^2} \frac{n^2}{m}))$, that is, proportional to the final tree size, if we used some dynamic compressed parentheses representation of trees [CHLS07]. The total time would be $O(\log m)$ per bit of the tree.

Note that, as we produce each tree level sequentially, and also traverse each matrix row (or adjacency list) sequentially, we can construct the tree on disk in optimal I/O time provided we have main memory to maintain $\log_k n$ disk blocks to output the tree, plus B disk blocks (where B is the disk page size in bits) for reading the matrix. The reason we do not need the n row buffers for reading is that we can cache the rows by chunks of B only. If later we have to read again from those rows, it will be after having processed a submatrix of $B \times B$ (given the way the algorithm traverses the matrix), and thus the new reads will be amortized by the parts already processed. This argument does not work on the adjacency list representation, where we need the n disk page buffers.

12.3 A hybrid approach

As we can observe in the examples of the previous section, if the adjacency matrix is very sparse, the greater k is, the more space L needs, because even though there are fewer submatrices in the last level, they are larger. Hence we may spend k^2 bits to represent very few 1s. Notice for example that when $k = 4$ in Figure 12.4, we store some last-level submatrices containing a unique 1, spending 15 more bits that are 0. On the contrary, when $k = 2$ (Figure 12.4) we use fewer bits for that last level of the tree.

We can improve our structure if we use a larger k for the first levels of the tree and a small k for the last levels. This strategy takes advantage of the strong points of both approaches:

- We use large values of k for the first levels of subdivision: the tree is shorter, so we will be able to obtain the list of neighbors faster, as we have fewer levels to traverse.

- We use small values of k for the last levels: we do not store too many bits for each 1 of the adjacency matrix, as the submatrices are smaller.

Figure 12.5 illustrates this hybrid solution, where we perform a first subdivision with $k = 4$ and a second subdivision with $k = 2$. We store the first level of the tree in T_1, where the subdivision uses $k = 4$ and the second level of the tree in T_2, where the subdivision uses $k = 2$. In addition, we store the 2×2 submatrices in L, as before.

$$T_1 = 1100010001100000,$$
$$T_2 = 1100\ \ 1000\ \ 0001\ \ 0101\ \ 1110,$$
$$L\ = 0100\ \ 0011\ \ 0010\ \ 0010\ \ 1010\ \ 1000\ \ 0110\ \ 0010\ \ 0100.$$

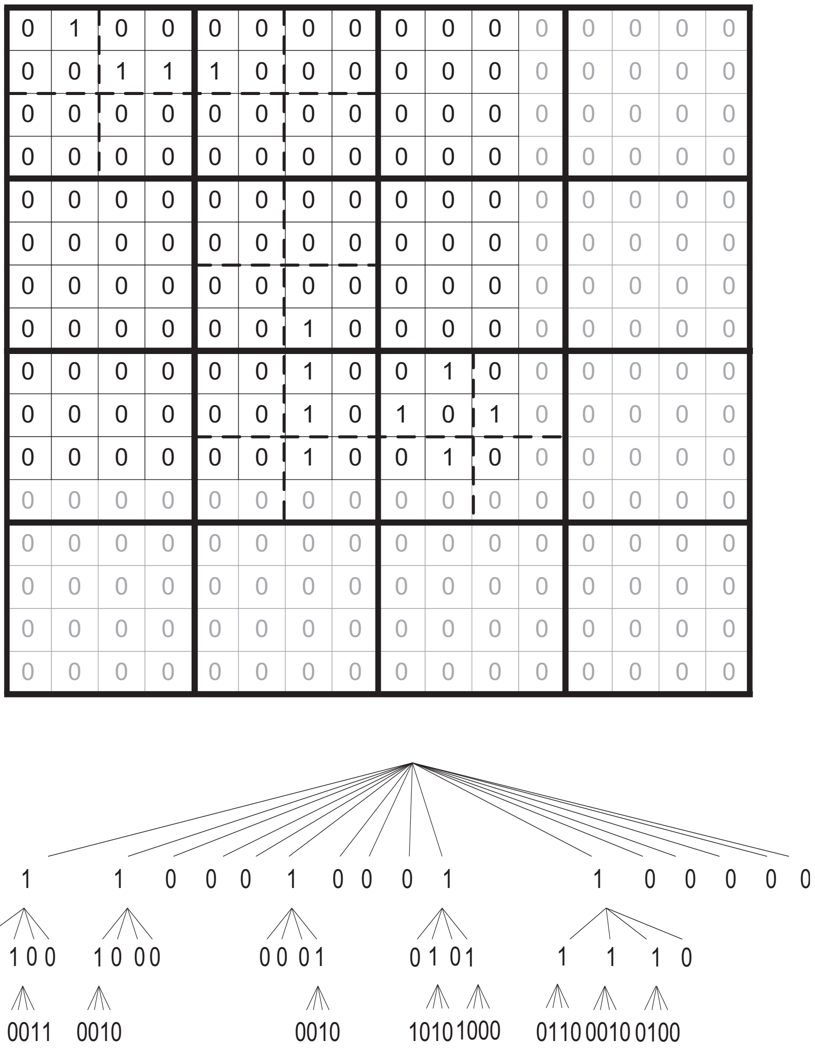
The algorithms for direct and reverse neighbors are similar to those explained for fixed k. Now we have a different sequence T_ℓ for each level, and L for the last level. There is a different k_ℓ per level, so Lemma 12.1 and algorithms **Direct** and **Reverse** for navigation in Section 12.2.3 must be modified accordingly. We must also extend n to $n' = \Pi_{\ell=0}^{h-1} k_\ell$, which plays the role of k^h in the uniform case.

12.4 Extended functionality

While alternative compressed graph representations [BV04, CN10c, AMN08] are limited to retrieving the direct, and sometimes the reverse, neighbors of a given page, we show now that our representation allows for more sophisticated forms of retrieval than extracting direct and reverse neighbors.

12.4.1 Single link retrieval

First, in order to determine whether a given page p points to a given page q, most compressed (and even some classical) graph representations have no choice but to extract all the neighbors of p (or a significant part of them) and see if q is in the set. We can answer such query in $O(\log_k n)$ time, by descending to exactly one child at each level of the tree, such that we can determine if the cell a_{pq} of the adjacency matrix is 1 (page p points to page q) or 0 (page p does not point to page q). We start at the root node and we descend recursively to the child node that represents the submatrix containing the cell a_{pq} of the adjacency matrix. Then, the algorithm is similar to the algorithm for retrieving direct neighbors, but choosing only the appropriate child to go down through the tree. More precisely, at level ℓ we descend to child $k \cdot \lfloor p/k^{h-\ell-1} \rfloor + \lfloor q/k^{h-\ell-1} \rfloor$, if it is not a zero, and compute the relative position of cell (p, q) in the submatrix just as in Section 12.2.3. If we reach the last level and find a 1 at cell (p, q), then there is a link, otherwise there is not.



0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	1	0	1	0	1	0	0	0	0	0
0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 12.5: Expansion, subdivision, and final example tree using different values of k.

Example We want to know if page 2 points to page 3, that is, we want to know if there is a 1 at cell $a_{2,3}$ of the adjacency matrix of Figure 12.2. We start at the root of the 2^2-tree and descend to the second child of the root node, since the cell $a_{2,3}$ belongs to the second submatrix of the adjacency matrix. Since we find a 0, then page 2 does not point to page 3. If we want to know if page 3 has a link to itself, then we start from the root node and go down through the fourth child of the node that represents the submatrix where the cell $a_{3,3}$ is located. There is a 1 there, indicating that this submatrix has at least one 1. Since the cell $a_{3,3}$ of the original adjacency matrix is the first cell of this submatrix, then we check the bit value contained in the first child of the node. It contains a 1, hence page 3 has a link pointing to itself.

The algorithm for checking whether one Web page p points to another Web page q is described in Algorithm 12.4. It is called **CheckLink** with $(k^h, p, q, -1)$ as parameters for: current submatrix size, row of interest in current submatrix, column of interest in current submatrix, and the position in $T : L$ of the node to process (again, we use the initial -1 to represent the root node). In addition, it is assumed that n is a power of k and that $rank(T, -1) = 0$.

Algorithm 12.4: **CheckLink**(n, p, q, z) returns 1 iff Web page p points to Web page q and 0 otherwise

```
if $z \geq |T|$ then /* leaf */
    return $L[z - |T|]$
else/* internal node */
    if $z = -1$ or $T[z] = 1$ then
        $y = rank(T, z) \cdot k^2$
        $y = y + \lfloor p/(n/k) \rfloor \cdot k + \lfloor q/(n/k) \rfloor$
        CheckLink$(n/k, p \bmod (n/k), q \bmod (n/k), y)$
    else
        return 0
    end
end
```

Hence, the worst-case navigation time to check if a Web page p points to another Web page q is $O(\log_k n)$, since a full traversal from the root node to a leaf node is required for every pair of connected Web pages.

12.4.2 Range queries

A second interesting operation is to find the direct neighbors of page p that are within a *range* of pages $[q_1, q_2]$ (similarly, the reverse neighbors of q that are within a range $[p_1, p_2]$). This is interesting, for example, to find out whether p points to a domain, or is pointed from a domain, in case we sort URLs in lexicographical

Algorithm 12.5: **Range**$(n, p_1, p_2, q_1, q_2, d_p, d_q, z)$

```
if $z \geq |T|$ then /* leaf */
    if $L[z - |T|] = 1$ then output $(d_p, d_q)$
else/* internal node */
    if $z = -1$ or $T[z] = 1$ then
        $y = rank(T, z) \cdot k^2$
        for $i = \lfloor p_1/(n/k) \rfloor \ldots \lfloor p_2/(n/k) \rfloor$ do
            if $i = \lfloor p_1/(n/k) \rfloor$ then $p'_1 = p_1 \bmod (n/k)$
            else $p'_1 = 0$
            if $i = \lfloor p_2/(n/k) \rfloor$ then $p'_2 = p_2 \bmod (n/k)$
            else $p'_2 = (n/k) - 1$
            for $j = \lfloor q_1/(n/k) \rfloor \ldots \lfloor q_2/(n/k) \rfloor$ do
                if $j = \lfloor q_1/(n/k) \rfloor$ then $q'_1 = q_1 \bmod (n/k)$
                else $q'_1 = 0$
                if $j = \lfloor q_2/(n/k) \rfloor$ then $q'_2 = q_2 \bmod (n/k)$
                else $q'_2 = (n/k) - 1$
                Range$(n/k, p'_1, p'_2, q'_1, q'_2, d_p + (n/k) \cdot i, d_q + (n/k) \cdot j, y + k \cdot i + j)$
            end
        end
    end
end
```

order. The algorithm is similar to **Direct** and **Reverse** in Section 12.2.3, except that we do not enter all the children $0 \leq j < k$ of a row (or column), but only from $\lfloor q_1/k^{h-\ell-1} \rfloor \leq j \leq \lfloor q_2/k^{h-\ell-1} \rfloor$ (similarly for p_1 to p_2).

Another operation of interest is to find all the links from a range of pages $[p_1, p_2]$ to another $[q_1, q_2]$. This is useful, for example, to extract all the links between two domains. The algorithm to solve this query indeed generalizes all of the others we have seen: extract direct neighbors of p ($p_1 = p_2 = p$, $q_1 = 0$, $q_2 = n-1$), extract reverse neighbors of q ($q_1 = q_2 = q$, $p_1 = 0$, $p_2 = n-1$), find whether a link from p to q exists ($p_1 = p_2 = p$, $q_1 = q_2 = q$), find the direct neighbors of p within range $[q_1, q_2]$ ($p_1 = p_2 = p$), and find the reverse neighbors of q within range $[p_1, p_2]$ ($q_1 = q_2 = q$). Figure 12.5 gives the algorithm. It is invoked as **Range** $(n, p_1, p_2, q_1, q_2, 0, 0, -1)$.

The total number of nodes of level ℓ that can overlap area $[p_1, p_2] \times [q_1, q_2]$ is $(\lfloor p_2/k^{h-\ell-1} \rfloor - \lfloor p_1/k^{h-\ell-1} \rfloor + 1) \cdot (\lfloor q_2/k^{h-\ell-1} \rfloor - \lfloor q_1/k^{h-\ell-1} \rfloor + 1) \leq ((p_2 - p_1 + 1)/k^{h-\ell-1} + 1) \cdot ((q_2 - q_1 + 1)/k^{h-\ell-1} + 1) = A/(k^2)^{h-\ell-1} + P/k^{h-\ell-1} + 1$, where $A = (p_2 - p_1 + 1) \cdot (q_2 - q_1 + 1)$ is the area to retrieve and $P = (p_2 - p_1 + 1) + (q_2 - q_1 + 1)$ is half the perimeter. Added over all the levels $0 \leq \ell < \lceil \log_k n \rceil$, the time complexity adds up to $O(A + P + \log_k n) = O(A + \log_k n)$. This gives $O(n)$ for retrieving direct and reverse neighbors (we made a finer average-case analysis in Section 12.2.3.1), $O(p_2 - p_1 + \log_k n)$ or $O(q_2 - q_1 + \log_k n)$ for ranges of direct or reverse neighbors, and $O(\log_k n)$ for queries on single links.

Moreover, we can check if there exists a link from a range of pages $[p_1, p_2]$ to another $[q_1, q_2]$ in a more efficient way than finding all the links in that range. If we just want to know if there is a link in the range, complete top-down traversals of the tree can be avoided if we reach an internal node that represents a submatrix of the original adjacency matrix that is entirely contained in the sought range and it is represented with a 1 bit in the k^2-tree. This means that there is at least one 1 inside that submatrix, and thus there is a link in the range of the query. This operation is performed analogously to the range query described in Algorithm 12.5, except that it checks if the current submatrix is completely contained in the sought range; in this case it finishes by returning a true value, avoiding the traversal to the leaf level and any other extra top-down traversal of the k^2-tree.

12.5 An enhanced variation of the k^2-tree technique

In this section we propose two modifications of the k^2-tree technique whose aim is to improve the efficiency of the method. The first one, explained in Section 12.5.1, consists in compacting the leaves representation of the k^2-tree, using an encoding scheme for sequences of integers. The other improvement, explained in Section 12.5.2, partitions the adjacency matrix of the Web graph into several submatrices and creates one k^2-tree for each one, such that the construction for the whole adjacency matrix becomes more efficient.

12.5.1 Using DACs to improve compression

The last level of the k^2-tree is stored as a bitmap L, as explained in detail in Section 12.2, which represents all the $k \times k$ submatrices of the original adjacency matrix containing at least one 1. These submatrices are represented consecutively following a depth-first traversal, composing a sequence of k^2-bit strings, each string representing a submatrix of the last level.

Instead of using a plain representation for all these submatrices, which uses a fixed number of bits for their representation (k^2 bits), we can create a vocabulary with all possible $k \times k$ submatrices and compact the sequence of submatrices using a variable-length encoding scheme that assigns a shorter code for those submatrices that appear more frequently. In order to preserve the efficient navigation over the compressed representation of the Web graph, we must guarantee fast access to any cell inside those encoded submatrices. Hence, we need a variable-length encoding scheme that supports direct access to any position of the encoded sequence, and at the same time, represents the sequence in a compact way. Thus, we use the Directly Addressable Codes (DACs), which is the method presented in the first part of this thesis, in Chapter 4.

The first step to replace the plain representation of the last level of the tree by a compact one consists in creating a vocabulary of all the $k \times k$ submatrices that appear in the adjacency matrix. This vocabulary is sorted by frequency such that the most frequent submatrices are located in the first positions. Once the vocabulary has been created, each matrix of the last level of the tree is replaced by its position in the sorted vocabulary. Therefore, the most frequent submatrices are associated with smaller integer values, and those which are not so frequent obtain larger integers. This frequency distribution can be exploited by a variable-length encoding scheme for integers to achieve a compact space. Consequently, the last level of tree is not longer represented by a sequence of k^2-bit strings corresponding to the submatrices, but by a sequence of integers, consecutively disposed according to the tree subdivision, where each integer represents one of the possible $k \times k$ submatrices of the vocabulary. As we have said, a variable-length encoding scheme can take advantage of the distribution, but direct access to any integer of the sequence must be supported to maintain the efficiency of the navigation algorithms. Hence, the sequence of integers is compacted using the variable-length encoding scheme we called Directly Addressable Codes.

As we have already mentioned, a larger k improves navigation, since it induces a shorter tree; however, space requirements become unaffordable, since the last level of the tree must store k^2 bits for every non-zero submatrix, even for those submatrices containing just one 1. In order to deal with the disadvantages of using a large k in the subdivision of the tree, the hybrid approach was previously proposed in Section 12.3. The modification of the data structure of the k^2-tree presented in this section can also minimize the effects caused by a large k value at the last level of the tree thanks to the fact that we will not store all the k^2 bits of the submatrices. Hence, we can use a greater value of k for the last level without dramatically worsening the space of the Web graph representation and obtaining better navigation performance. The number of possible matrices of size $k \times k$ will increase, but only a few of them will appear in practice, due to sparseness, clustering and statistical properties of the matrix. Moreover, by using a greater k we can obtain better time and space results due to the fact that this shortens the tree that represents the adjacency matrix. Hence, fewer bits are used for bitmap T and fewer levels must be traversed until the last level is reached. To obtain a higher k value for the last level we can just use a large fixed k value for all the levels of the tree, or use the hybrid approach using the desired large k for the leaves.

By following this approach, we can exploit the different patterns described by Asano *et al.* [AMN08], such as horizontal, vertical, and diagonal runs. Submatrices containing those patterns will appear more frequently, and consequently, fewer bits will be used for their representation and better compression will be obtained.

12.5.2 Partition of the adjacency matrix

Another minor improvement consists in the partition of the original adjacency matrix into a grid of several large square submatrices of size $S \times S$ bits, obtaining P^2 submatrices where $P = n/S$. Then, P^2 k^2-tree are constructed, one for each submatrix of the partition.

With this modification, the practical time for the construction decreases, and more importantly, navigation time improves due to the fact that the P^2 k^2-tree become shorter than the original k^2-tree. This can be seen as using the hybrid approach with $k_1 = P$ for the first level of the tree, however, it becomes useful to make this distinction in practice for the first level, since it facilitates the construction and increases the locality of reference.

Chapter 13

Experimental evaluation

We devote this chapter to presenting the performance of the new technique called k^2-tree, proposed in Chapter 12, exhibiting the empirical results obtained by different variants of the technique. The behavior of our method is also compared to other proposals of the state of the art that support direct and reverse neighbors. We show experimentally that our technique offers a relevant space/time tradeoff to represent Web graphs, that is, it is much faster than those that take less space, and much smaller than those that offer faster navigation. Thus our representation can become the preferred choice for many Web graph traversal applications: Whenever the compression it offers is sufficient to fit the Web graph in main memory, it achieves the best traversal time within that space. Furthermore, we show that our representation allows other queries on the graph that are not usually considered in compressed graph representations in an efficient way, such as single link retrieval or range searches.

We start by describing the experimental setup in Section 13.1, then in Section 13.2 we compare all the variants of the k^2-tree technique that have been proposed in the previous chapter. Section 13.3 includes a comparison between the best results of our proposal and different strategies already known in the field. Section 13.4 analyzes the results of the extended navigation supported by our technique, and Section 13.5 studies the time and space analyses presented in the previous chapters and the behavior of the proposal for random graphs. Finally, 13.6 summarizes the main conclusions extracted from the experimental evaluation.

13.1 Experimental framework

We ran several experiments over some Web graphs from the *WebGraph* project, some of them gathered by UbiCrawler [BCSV04]. These data sets are made available to

Table 13.1: Description of the graphs used.

File	Pages	Links	Size (MB)
CNR (2000)	325,577	3,216,152	14
EU (2005)	862,664	19,235,140	77
Indochina (2002)	7,414,866	194,109,311	769
UK (2002)	18,520,486	298,113,762	1,208
Arabic (2005)	22,744,080	639,999,458	2,528

the public by the members of the *Laboratory for Web Algorithmics*[1] at the *Università Degli Studi Di Milano.*

Table 13.1 gives the main characteristics of the graphs used. The first column indicates the name of the graph (and the WebGraph version used). Second and third columns show the number of pages and links, respectively. The last column gives the size of a plain adjacency list representation of the graphs (using 4-byte integers).

The machine used in our tests is a 2GHz Intel®Xeon® (8 cores) with 16 GB RAM. It ran Ubuntu GNU/Linux with kernel version 2.4.22-15-generic SMP (64 bits). The compiler was gcc version 4.1.3 and `-O9` compiler optimizations were set. Space is measured in bits per edge (bpe), by dividing the total space of the structure by the number of edges (i.e., links) in the graph. Time results measure average CPU user time per neighbor retrieved: We compute the time to search for the neighbors of all the pages (in random order) and divide by the total number of edges in the graph.

13.2 Comparison between different alternatives

We first study our approach with different values of k. Table 13.2 shows 12 different alternatives of our method over the EU graph using different values of k. All build on the *rank* structure that uses 5% of extra space [GGMN05]. The first column names the approaches as follows: $'2\times 2'$, $'3\times 3'$ and $'4\times 4'$ stand for the alternatives where we subdivide the matrix into 2×2, 3×3 and 4×4 submatrices, respectively, in every level of the tree. On the other hand, we denote $'H-i'$ the hybrid approach where we use $k=4$ up to level i of the tree, and then we use $k=2$ for the rest of the levels. The second and third columns indicate the size, in bytes, used to store the tree T and the leaves L, respectively. The fourth column shows the space needed in main memory by the representation (e.g., including the extra space for *rank*), in bits per edge. Finally, the last two columns show the times to retrieve the

[1] http://law.dsi.unimi.it/

Variant	Tree (bytes)	Leaves (bytes)	Space (bpe)	Direct (μs/e)	Reverse (μs/e)
2×2	6,860,436	5,583,057	5.21076	2.56	2.47
3×3	5,368,744	9,032,928	6.02309	1.78	1.71
4×4	4,813,692	12,546,092	7.22260	**1.47**	**1.42**
$H - 1$	6,860,432	5,583,057	5.21077	2.78	2.62
$H - 2$	6,860,436	5,583,057	5.21077	2.76	2.59
$H - 3$	6,860,412	5,583,057	**5.21076**	2.67	2.49
$H - 4$	6,861,004	5,583,057	5.21100	2.53	2.39
$H - 5$	6,864,404	5,583,057	5.21242	2.39	2.25
$H - 6$	6,876,860	5,583,057	5.21760	2.25	2.11
$H - 7$	6,927,924	5,583,057	5.23884	2.10	1.96
$H - 8$	7,159,112	5,583,057	5.33499	1.97	1.81
$H - 9$	8,107,036	5,583,057	5.72924	1.79	1.67

Table 13.2: Comparison of our different approaches over graph EU.

direct (fifth column) and reverse (sixth) neighbors, measured in microseconds per link retrieved (μs/e).

We observe that, when we use a fixed k, we obtain better times when k is greater, because we are shortening the height of the tree, but the compression ratio worsens, as the space for L becomes dominant and many 0s are stored in there.

If we use a hybrid approach, we can maintain a compression ratio close to that obtained by the $'2 \times 2'$ alternative (notice that the same space is used to represent the leaves) while improving the time, until we get close to the $'4 \times 4'$ alternative. The best compression is obtained for $'H - 3'$, even better than $'2 \times 2'$. Figure 13.1 shows similar results graphically, for the four middle-large graphs (EU, Indochina, UK and Arabic), illustrating space on top and time to retrieve direct neighbors on the bottom. It can be seen that the space does not worsen much if we keep $k = 4$ up to a moderate level, whereas times improve consistently. A medium value, say switching to $k = 2$ at level 7, looks as a good compromise.

We can also observe in Table 13.2 that retrieval times for reverse neighbors are always slightly better than for direct neighbors. This is due to the fact that there is more locality in the transposed graph of the Web graph, which also makes it more compressible. We will see in Section 13.5 that retrieval times are dependent of the distribution of the links along the adjacency matrix, such that better results are obtained when the Web graph exhibits more locality.

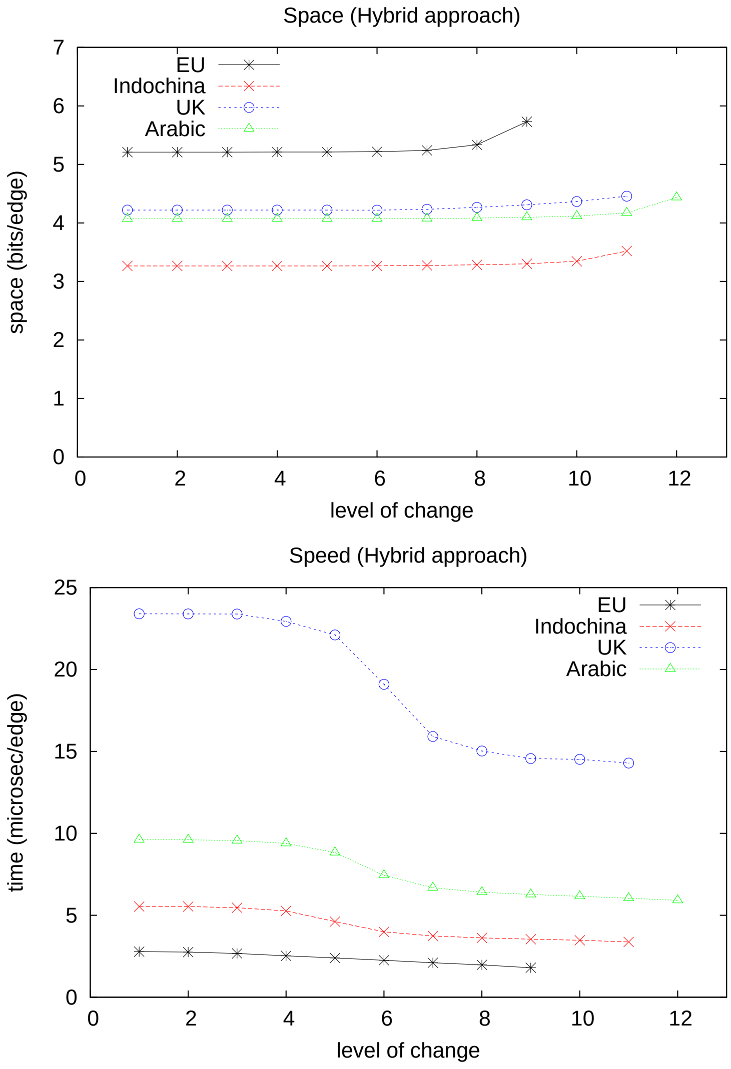


Figure 13.1: Space/time behavior of the hybrid approach when we vary the level where we change the value of k.

Configuration	Tree (bytes)	Leaves (bytes)	Voc. Leaves (bytes)	Space (bpe)	Time (μs/e)
2×2	6,860,436	5,583,057	0	5.21	2.56
4×4	4,813,692	12,546,092	0	7.22	1.47
$b = 8$ $k = 4$	4,813,692	7,401,867	25,850	5.13	1.49
$b = 4$ $k = 4$	4,813,692	6,023,699	25,850	4.55	1.60
$b = 2$ $k = 4$	4,813,692	5,721,535	25,850	4.43	1.72
opt $k = 4$	4,813,692	5,568,435	25,850	**4.36**	1.64
$b = 8$ $k = 8$	4,162,936	5,403,021	1,679,232	4.71	**1.29**
$b = 4$ $k = 8$	4,162,936	4,958,858	1,679,232	4.53	1.46
$b = 2$ $k = 8$	4,162,936	5,154,420	1,679,232	4.61	1.68
opt $k = 8$	4,162,936	4,812,243	1,679,232	4.47	1.44

Table 13.3: Space and time results when compressing graph EU using DACs for the leaves representation.

According to the results, the faster alternative consists in using a large k; however, this worsens the compression ratio, since the leaves representation occupies a considerable amount of space. Hence, we now study the approach where we apply DACs to the sequence of $k \times k$ submatrices of the leaf level. Table 13.3 analyzes the space consumption and navigational times for different configurations of this approach for graph EU compared to the original k^2-tree representations $'2 \times 2'$ and $'4 \times 4'$. The first column indicates the configuration used for the k^2-tree. We built three representations using $k = 4$ for all the levels of the tree, and representing the leaf level with DACs using different fixed values for the parameter b of the encoding scheme: $b = 8$, $b = 4$ and $b = 2$. We also configured DACs ("opt") with the values obtained by the optimization algorithm presented in Chapter 4.2, which minimizes the space usage. Analogously, we built four representations of the k^2-tree using $k = 8$ for all the levels of the tree and fixed $b = 8$, $b = 4$, $b = 2$ for DACs, and also the optimal configuration "opt", over the sequence that represents the leaves of the tree. The second, third and fourth columns indicate the size, in bytes, used to store the tree T, the compact representation of the leaves using DACs and the vocabulary of the $k \times k$ leaf submatrices, respectively. The fifth column shows the total space needed by the representation in bits per edge and the last column shows the times to retrieve direct neighbors, measured in microseconds per link retrieved (μs/e).

If we compare the results obtained by the approach using DACs when $k = 4$ to those obtained by the $'4 \times 4'$ alternative, we can observe that while the space consumption for the tree representation is maintained, the space usage for the leaves is reduced, achieving a decrease in the total space of around 2.5 bpe, which also beats the space usage of the smallest representation obtained for this graph using the hybrid approach (5.21 bpe, which was obtained by $'H-3'$ in Table 13.2). This compact representation outperforms alternative $'2\times 2'$ both in space and time, but is not as efficient as alternative $'4\times 4'$ for the direct neighbors extraction. This is due to the fact that leaf submatrices are no longer represented in plain form, so the compact representation degrades to a certain extent the navigational times (several *access* operations are needed over the sequence of integers that represents the last level of the tree). Using $b=8$ for the block size parameter of the DACs encoding scheme still improves the space usage of alternative $'4\times 4'$, while obtaining a very close retrieval time, since the sequence is represented using DACs with very few levels and operations are done in bytes with no need of bitwise operations. By using a lower value of b, such as $b=4$ or $b=2$, space requirements decrease while navigational time increases (DACs encoded sequence needs a higher number of levels to represent the submatrices of the leaf level). The smallest space usage is obtained when we use the optimal values for DACs. With this configuration the navigational time is faster than with $b=2$ but slower than with $b=4$.

As we have already said, better time performance can be achieved if we use larger k values, without causing an increase of the space usage. Actually, with the compact representation of the leaves better compression ratios can be obtained. When $k=8$, the size of the representation of the leaves is smaller (there is a lower number of non-zero submatrices of size 8×8), but the size of the vocabulary of non-zero submatrices becomes considerable greater. Overall, the total size of the representation can be smaller than using $k=4$. For instance, an 8^2-tree using DACs with $b=4$ achieves a more compact and faster representation than alternative $'4\times 4'$. In this case, when $k=8$ there is a large number of different non-zero submatrices in the leaf level. Hence, a lower value of b for DACs, such as $b=2$, does not only worsen the navigational time but it also degrades the space requirements of the representation. If we use DACs with the optimal configuration, we obtain the smallest space among all the representations when $k=8$, and even better navigation time than using $b=4$. Hence, using the optimal values obtained for DACs obtains attractive results both in time and space.

We now study the space/time tradeoff for all the alternatives previously mentioned over different graphs. Figures 13.2 and 13.3 represent the time needed to retrieve direct neighbors (μs/e) over graphs `EU`, `Indochina`, `UK` and `Arabic`. We include our alternatives $'2\times 2'$, $'3\times 3'$, $'4\times 4'$, and `Hybrid5`. `Hybrid5` represents in a curve all the time and space results obtained by the $'H-i'$ hybrid k^2-trees where we use $k=4$ up to level i of the tree, and then we use $k=2$ for the rest of the levels. All of these alternatives are built using the slower solution for *rank* that needs just 5% of extra space [GGMN05]. In addition, we draw another curve for the hybrid

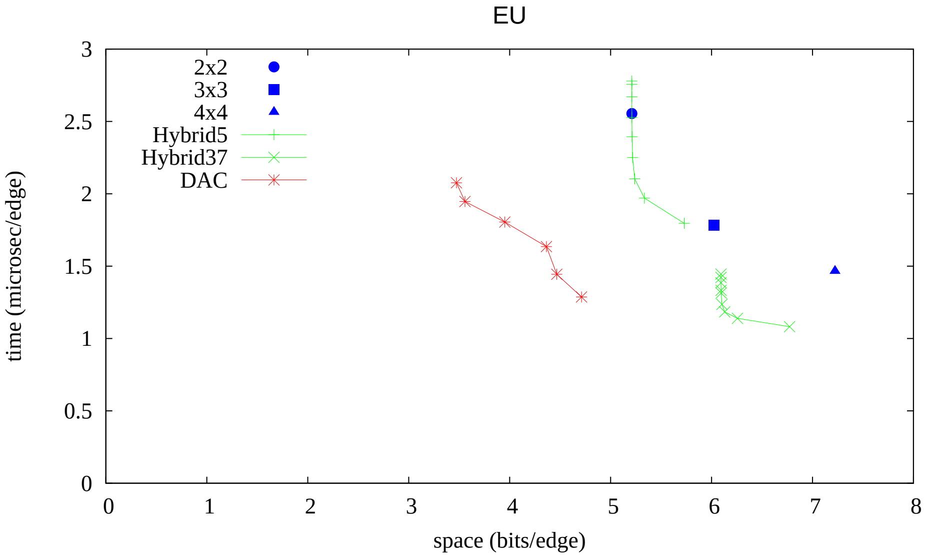
Table 13.4: Results of k^2-tree technique (with DACs) over large graphs.

File	Pages	Links	Space (bpe)	Time (μs/e)	WebGraph (bpe)
it-2004	41,291,594	1,150,725,436	1.76	9.26	2.55
sk-2005	50,636,154	1,949,412,601	1.97	4.39	2.87
webbase-2001	118,142,155	1,019,903,190	2.88	66.49	5.34
uk-2007-05	105,896,555	3,738,733,648	1.75	7.63	2.17

approach, `Hybrid37`, which uses the faster *rank* method that needs 37.5% extra space on top of T. We also include a line of points for different configurations of the Directly Addressable Codes approach, denoted by `DACs`. Some tuning parameters are modified in order to obtain those points, such as the value of k for the different levels of the tree or the value of b for the DACs encoding scheme. We can observe that the best space results are obtained by the `DACs` alternative, where the last level of the tree is represented in a compact way such that large k values can be used without worsening the compression ratio. The best navigation times are obtained by the `Hybrid37`, since it uses a faster but heavier *rank* structure. Alternatively, `DACs` could improve its navigational times if it used that structure, but it also would worsen its space usage.

As a proof of concept of the scalability of our technique, we show now the space and time results for some large Web graphs. We took four graphs from http://law.dsi.unimi.it website, which were compressed using LLP [BRSV11] and WebGraph [BV04]. We built a compact representation of k^2-tree using the same configuration for the four graphs, which consisted in partitioning the original adjacency matrix into several submatrices of size $2^{22} \times 2^{22}$. Then, we create for each submatrix a hybrid k^2-tree using $k = 4$ for the first 6 levels and $k = 2$ for the rest. The subdivision is not continued until the last level, leaving the last 3 levels of the tree without subdividing. Hence, $k = 8$ at the last level of the tree. Finally, we use DACs with the the optimal values to minimize the space required by the representation of the leaves.

Table 13.4 shows some space and time results for these large Web graphs. The first column indicates the name of the graph. Second and third columns show the number of pages and links, respectively. The fourth column shows the total space needed by the representation in bits per edge and the fifth column shows the time to retrieve direct neighbors, measured in microseconds per link retrieved (μs/e). Finally, the last column shows the sum of the space used by the highly compressed versions of each graph and its transpose obtained by WebGraph technique [BV04]. Note that their representation does not support fast random access. We can ob-



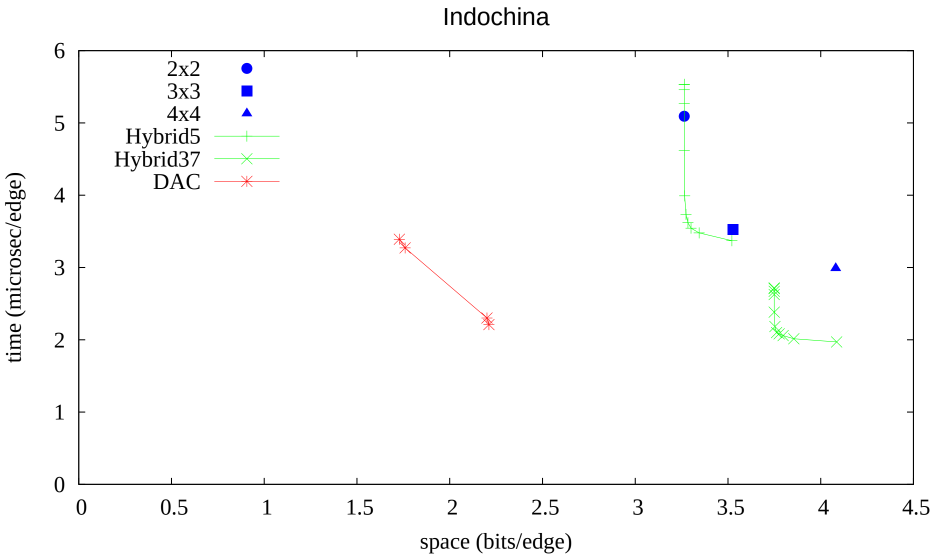
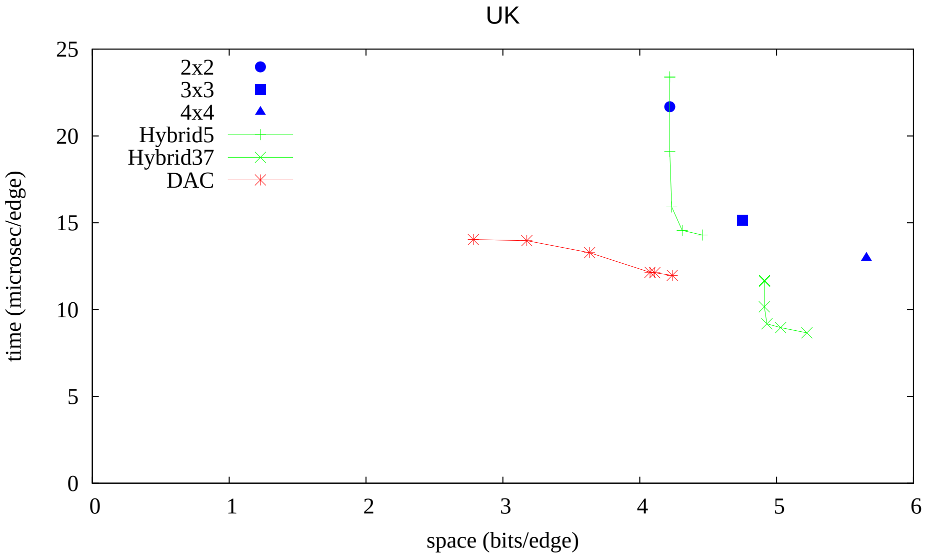


Figure 13.2: Space/time tradeoff to retrieve direct neighbors for `EU` (top) and `Indochina` (bottom) graphs.



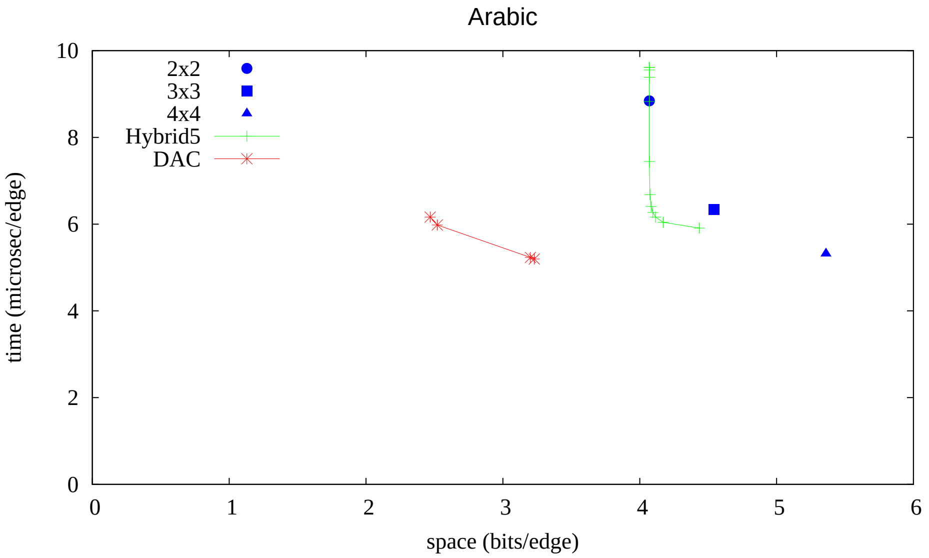


Figure 13.3: Space/time tradeoff to retrieve direct neighbors for `UK` (top) and `Arabic` (bottom) graphs.

Crawl	k^2-tree	Re-Pair WT	Re-Pair GMR	Re-Pair (div+rev)	WebGraph (dir+rev)	Asano *et al.* ×2
EU	**3.47**	3.93	5.86	7.65	7.20	5.56
Indochina	**1.73**	2.30	3.65	4.54	2.94	
UK	**2.78**	3.98	6.22	7.50	4.34	
Arabic	**2.47**	2.72	4.15	5.53	3.25	

Table 13.5: Space consumption (in bpe) of the most compact k^2-tree representation for different Web graphs, and previous work.

serve that our technique achieves significantly better spaces while supporting fast navigation over the compressed graph.

13.3 Comparison with other methods

13.3.1 Space usage

We first analyze our proposal in terms of space. We extract from the most recent paper of Claude and Navarro [CN10b] an up-to-date comparison table with the minimum space consumption (in bits per edge) of the alternatives in the literature that support both direct and reverse navigation within reasonable time, that is, much faster than decompressing the whole graph. The space results reported in that paper for our technique k^2-tree did not include the latest modifications of the structure detailed in Section 12.5, where DACs are used to represent the leaf level. Hence, we update those values according to the most recent results obtained, and illustrate that comparison in Table 13.5.

This comparison includes the space required (in bpe) for the four bigger crawls. The first column is devoted to our proposal. The space reported corresponds to the following configurations:

- For graph EU, we use a partition of the original adjacency matrix into several submatrices of size $2^{18} \times 2^{18}$. Then, we create for each submatrix a hybrid k^2-tree using $k = 4$ for the first 5 levels and $k = 2$ for the rest. The subdivision is not continued until the last level, leaving the last 3 levels of the tree without subdividing. Hence, $k = 8$ at the last level of the tree. Finally, we use DACs with the the optimal values to minimize the space required by the representation of the leaves.
- For graph Indochina, the submatrices generated after the first partition of the adjacency matrix are of size $2^{20} \times 2^{20}$, and then we use the same configuration

than for graph `EU`, that is, a hybrid approach with $k = 4$ for the first 5 levels, $k = 2$ for the rest except for the last level, using $k = 8$ for the leaves, which are represented with DACs using the configuration obtained by the optimization algorithm.

- For graph `UK`, the submatrices after the partition are of size $2^{22} \times 2^{22}$ and k is changed from $k = 4$ to $k = 2$ at level 6 of the tree. We also use $k = 8$ for the last level of the tree and DACs with the optimal b values for the representation of leaves.

- We use this last configuration also for graph `Arabic`.

Second and third columns of Table 13.5 correspond to alternatives `Re-Pair WT` and `Re-Pair GMR` presented by Claude and Navarro [CN10b] (already explained more in detail in Section 11.3.2). The comparison also includes the space obtained by the original proposal of Claude and Navarro [CN10c] that retrieves just direct neighbors. In this case, both the graph and its transpose are represented in order to achieve reverse navigation as well (`Re-Pair (dir+rev)`). The same is done with Boldi and Vigna's technique [BV04] (*WebGraph*), as it also allows for direct neighbors retrieval only. `WebGraph (dir+rev)` denotes the alternative using version 2.4.2, variant strictHostByHostGray, adding up the space for the direct and the transposed graph. Only the required space on disk of the structure is reported, even if the process requires much more memory to run. For this comparison, parameters are set in order to favor compression over speed (window size 10, maximum reference unlimited). With this compression they retrieve direct neighbors in about 100 microseconds [CN10b].

Finally, last column shows the space achieved by Asano *et al.* [AMN08] for graph `EU` (which is the largest graph they report). As, again, their representation cannot retrieve reverse neighbors, `Asano`×2 is an estimation, obtained by multiplying their space by 2, of the space they would need to represent both the normal and transposed graphs. This is probably slightly overestimated, as transposed Web graphs compress slightly better than the original ones. Indeed it could be that their method can be extended to retrieve reverse neighbors using much less than twice the space. The reason is that, as it is explained in Section 11.3.3, they store the intra-domain links (which are the major part) in a way that they have to uncompress a full domain to answer direct neighbor queries, and answering reverse neighbors is probably possible with the same amount of work. They would have to duplicate only the inter-domain links, which account for a minor part of the total space. Yet, this is speculative. Besides, as we see later, the times using this representation are non-competitive by orders of magnitude anyway.

As we can see in the comparison, our proposal obtains the best space from all the alternatives of the literature. `Re-Pair WT` was proven to achieve the smallest space reported in the literature while supporting direct and reverse neighbors in reasonable time: around 35 microseconds/edge for direct and 55 for reverse neighbors [CN10b]. With the spaces obtained by the k^2-tree using DACs this statement does not longer hold. Our k^2-tree representation of Web graphs becomes the most attractive alternative when minimum space usage is sought.

13.3.2 Retrieval times

In this section we focus on studying the efficiency of the navigation of the k^2-tree technique. We first compare graph representations that allow one retrieving both direct and reverse neighbors, i.e., those included in the previous table, and then we will compare the k^2-tree technique with some of the techniques that only support direct navigation.

The technique `Re-Pair WT` ([CN10b]) obtained the smallest space previously reported in the literature. In addition, they can navigate the graph in reasonable time: around 35 microseconds/edge for direct and 55 for reverse neighbors. As we have seen in Table 13.5, there exist some configurations of the k^2-tree representation that obtain better compression ratio and navigate the Web graph faster than `Re-Pair WT`: about 2-15 microseconds/edge depending on the graph (as we have seen in Figures 13.2 and 13.3 in Section 13.2). Hence, these k^2-tree representations outperform the `Re-Pair WT` technique both in space and time efficiency. Yet, `Re-Pair WT` is no longer the most attractive alternative to represent a Web graph when very little space and forward and reverse navigation are required.

We also compare our technique with the other methods of the literature, which are not as succinct as `Re-Pair WT`, but they achieve more efficient time results. Figures 13.4, 13.5, 13.6 and 13.7 show the space/time tradeoff for retrieving direct (left) and reverse (right) neighbors over different graphs. We measure the average time efficiency in μs/e as before. Representations providing space/time tuning parameters appear as a line, whereas the others appear as a point.

We compare our compact representations with the fastest proposal in [CN10b] that computes both direct and reverse neighbors (`Re-Pair GMR`), as well as the original representation in [CN10c] (`Re-Pair (dir+rev)`). A variant of Re-Pair GMR labeled `Re-Pair GMR (2)` is also included, where *access* operation is solved in constant time and *select* in time $O(\log\log n)$. Thus, `Re-Pair GMR` is faster for reverse neighbors (using constant-time select), and `Re-Pair GMR (2)` is faster on direct neighbors (using constant-time access). We also include the `WebGraph (dir+rev)` alternative from Boldi and Vigna.[2]

We also include the proposal of Apostolico and Drovandi [AD09], presented in Section 11.3.5. `AD(dir+rev)` denotes the alternative using the version 0.2.1 of

[2] I would like to thank Francisco Claude for providing the numeric results obtained by the techniques `Re-Pair`, `Re-Pair GMR`, `Re-Pair GMR (2)`, and `WebGraph(dir+rev)`.

their software[3], where both the graph and its transpose are represented in order to achieve reverse navigation as well. We vary the compression level ℓ of AD technique to obtain a space-time tradeoff, using $\ell = 4, 8, 16, 100, 200, 500, 1000, 1500$. We add the space for the offsets and indexes of the first node of each chunk (64 bits and 32 bits per chunk respectively) to support random access to the graph. Notice that AD technique performs a reordering of the node identifiers based on the Breadth First Search (BFS) of the graph instead of the lexicographic order. This permutation of identifiers is not accounted for in the space results reported next. However, this mapping should be stored if we want to recover the graph with the original node identifiers.

We study their navigation efficiency compared to our alternatives $'2 \times 2'$, $'3 \times 3'$, $'4 \times 4'$, Hybrid5, Hybrid37 and DAC, described in Section 13.2.

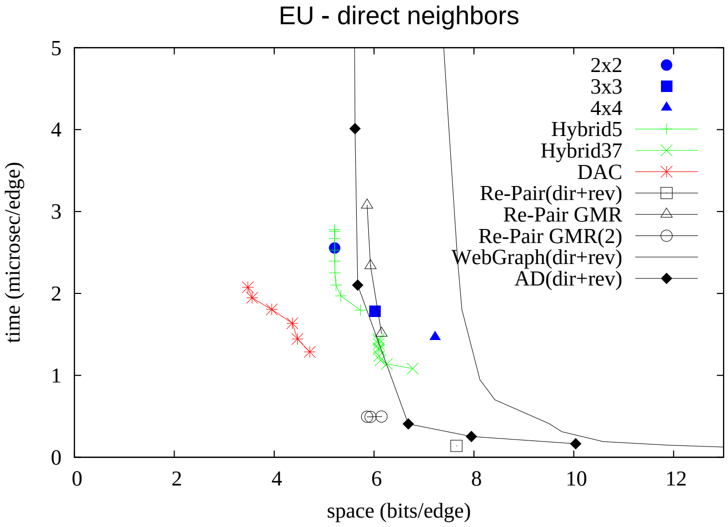
As we can see, our representations (particularly DAC, and also Hybrid5 and $'2 \times 2'$ over EU) achieve the best compression (1.8 to 5.3 bpe, depending on the graph) among all the techniques that provide direct and reverse neighbor queries. The alternative that gets closer is AD(dir+rev), which achieves very fast navigation when occupying more space, but it gets considerable slower when the compression ratio gets closer to the smallest configuration of DAC. Re-Pair GMR also obtains attractive space results, but it is much slower to retrieve direct neighbors, or Re-Pair GMR (2), which is much slower to retrieve reverse neighbors. Finally, WebGraph (dir+rev) and Re-Pair (dir+rev) offer very attractive time performance, similar to AD(dir+rev), but they need significantly more space. As explained, using less space may make the difference between being able of fitting a large Web graph in main memory or not.

If, instead, we wished only to carry out forward navigation, alternatives RePair, WebGraph and particularly AD become preferable (smaller and faster than ours) in most cases. Figure 13.8, shows graph EU, where we still achieve significantly less space than WebGraph, but not than AD.

We now present a comparison of the performance of our proposal with Buehrer and Chellapilla's technique [BC08], described in Section 11.3.4, which will be denoted by VNM. As we do not have their code and we do not have a comparable experimental evaluation on time performance neither, we will estimate the space and time results obtained by their compression method.

Table 13.6 shows in the first column VNM(∞) the space consumption they report for the EU, Indochina, UK and Arabic graphs. This space does not include the space required for the storage of the offset per node to provide random navigation over the graph, hence this variant does not provide direct access. In the second column VNM×2 we estimate the space required to represent both the normal and

[3] The implementation (in Java) for Apostolico and Drovandi's technique is publicly available in http://www.dia.uniroma3.it/ drovandi/software.php. I would like to thank Guido Drovandi for solving some doubts I had about their implementation.



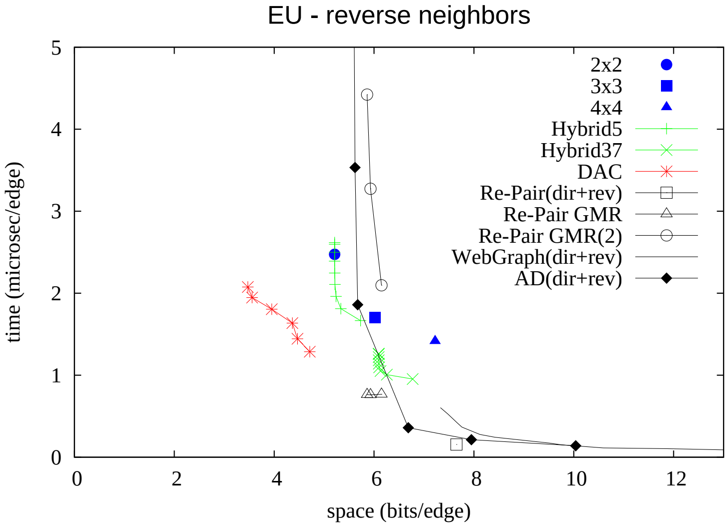
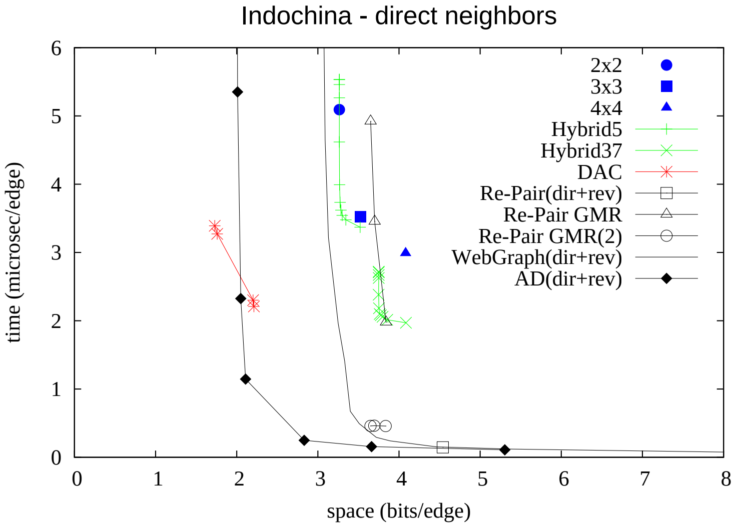


Figure 13.4: Space/time tradeoff to retrieve direct neighbors (top) and reverse neighbors (bottom) for EU graph.



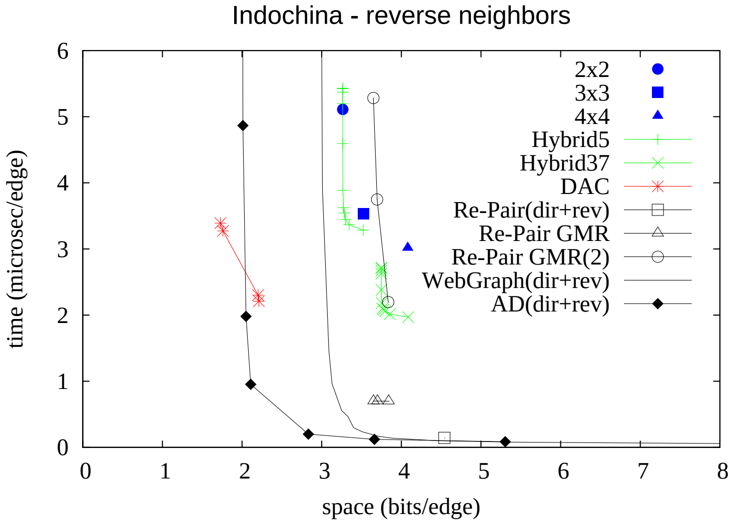
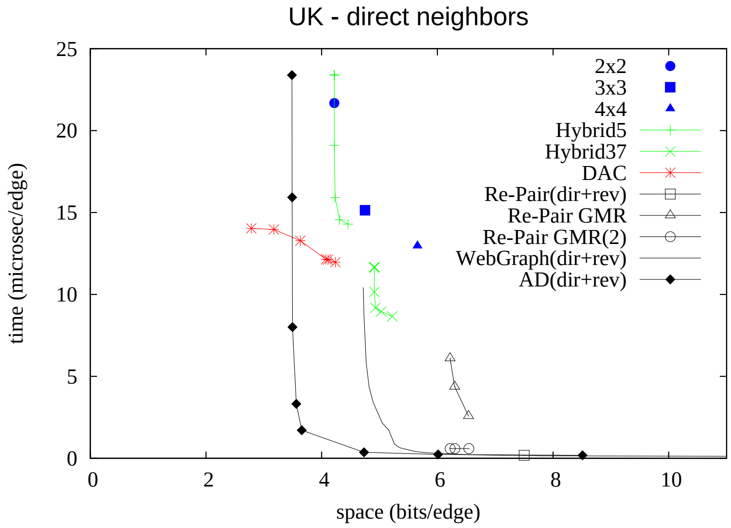


Figure 13.5: Space/time tradeoff to retrieve direct neighbors (top) and reverse neighbors (bottom) for `Indochina` graph.



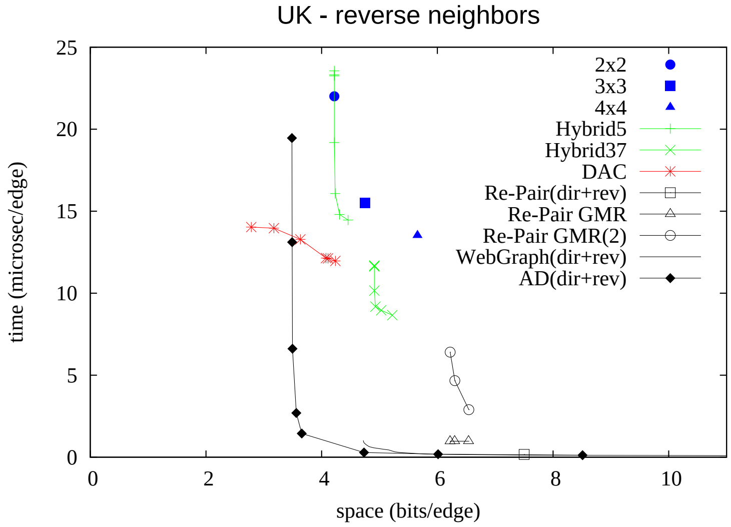
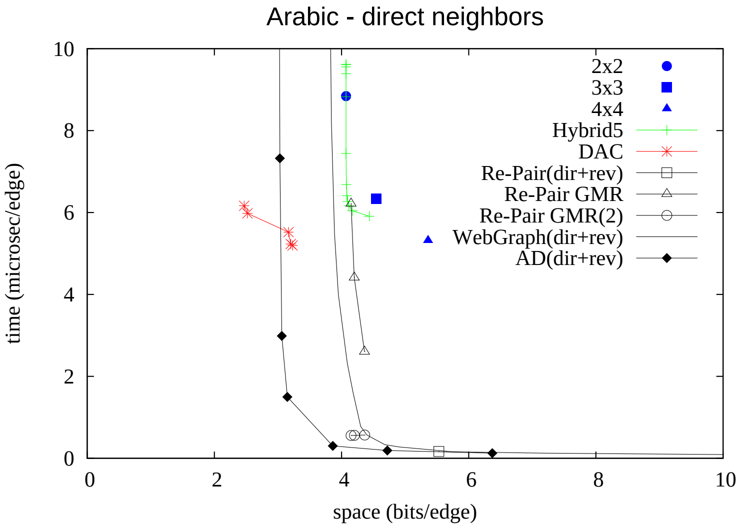


Figure 13.6: Space/time tradeoff to retrieve direct neighbors (top) and reverse neighbors (bottom) for `UK` graph.



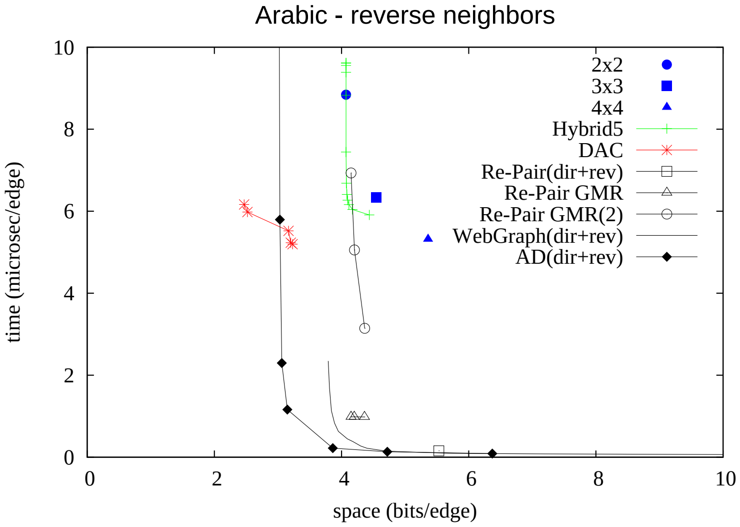


Figure 13.7: Space/time tradeoff to retrieve direct neighbors (top) and reverse neighbors (bottom) for `Arabic` graph.

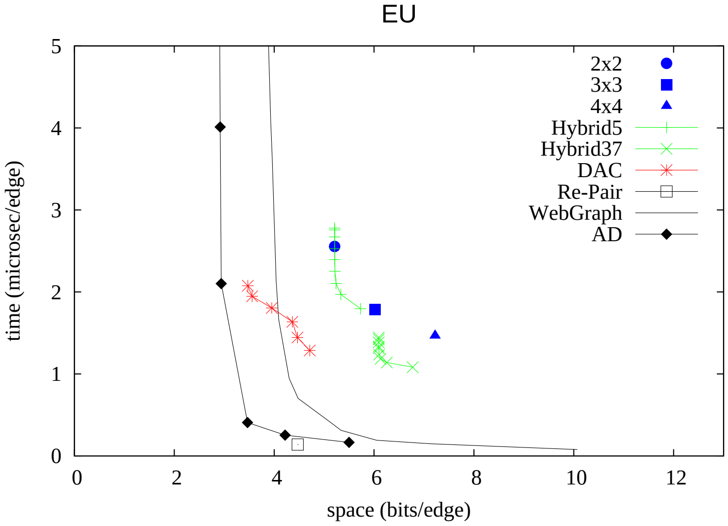


Figure 13.8: Space/time tradeoff for graph representations that retrieve only direct neighbors (and ours) over graph EU.

transposed graphs in order to support direct and reverse navigation by multiplying their space by 2. Again, this might be overestimated, as transposed Web graphs compress slightly better than the original ones. In the next two columns, `VNM` and `VNM`×2, we add the space of their pointers array, such that the compressed graph can be randomly navigated. In the last column of the table we include the results for our smallest representation (described in Section 13.3.1). As we can observe, `VNM` obtains worse compression ration than our k^2-tree representation when compressing the normal and the transposed graph, even when no random access is supported. No traversal times are reported by Buehrer and Chellapilla [BC08] for their method. However, Claude and Navarro [CN10c] made an estimation, comparing both algorithms, and stated that the two techniques have a similar time performance. Hence, compared with the k^2-tree technique `VNM` would be faster, but requiring a significantly higher amount of space.

We also compare our proposal with the method in [AMN08] (`Asano`). As we do not have their code, we ran new experiments on a Pentium IV of 3.0 GHz with 4 GB of RAM, which resembles better the machine used in their experiments. We used the smallest graphs, on which they have reported experiments. Table 13.7 shows the space and average time needed to retrieve the whole adjacency list of a page, in milliseconds per page. We also include the space of the estimation `Asano`×2

Space (bpe)	VNM(∞)	VNM(∞)×2	VNM	VNM×2	k^2-tree
EU	2.90	5.80	4.07	8.14	3.47
Indochina	-	-	-	-	1.73
UK	1.95	3.90	3.75	7.50	2.78
Arabic	1.81	3.62	2.91	5.82	2.47

Table 13.6: Space comparison between k^2-tree and Buehrer and Chellapilla's technique for several graphs. Columns VNM(∞)×2 and VNM×2 are estimations.

obtained by multiplying their space by 2, of the space they would need to represent both the normal and transposed graphs.

For the comparison shown in Table 13.7, we represent CNR with the Hybrid5 alternative where we use $k = 4$ only in the first level of the tree, and then we use $k = 2$ for the rest of the levels. In the same way, we represent EU using the Hybrid5 alternative where we change from $k = 4$ to $k = 2$ in the third level of the tree. These are the most space-efficient configurations of the alternative Hybrid5 for those graphs. Compression ratio and time results are shown in the third column of the table.

We observe that our method is orders of magnitude faster to retrieve an adjacency list, while the space is similar to Asano×2. The time difference is so large that it is also possible to be competitive even if part of our structure (e.g. L) is on secondary memory. Our main memory space in this case, omitting bitmap L, is reduced to about half the space of the Hybrid5 alternative. The exact value (in bpe) is shown in the last column of Table 13.7, denoted by Hybrid5 no-L. Time results are slightly worse than the Hybrid5 alternative, since frequent accesses to disk are required. However, it is still orders of magnitude faster compared to Asano *et al.* technique.

13.4 Extended functionality performance

As we have said in Section 12.4, our representation supports extra navigability, in addition to the most common functionality such as extracting direct and reverse neighbors. This extended navigation includes single links or range queries.

13.4.1 Single link retrieval

If we want to know whether a Web page p links to another Web page q, the k^2-tree offers a more efficient procedure than extracting the whole adjacency list of Web

Space (bpe)	Asano	Asano×2	Hybrid5	Hybrid5 no-L
CNR	1.99	3.98	4.46	2.52
EU	2.78	5.56	5.21	2.89
Time (msec/page)				
CNR	2.34		0.048	0.053
EU	28.72		0.099	0.110

Table 13.7: Comparison with approach Asano on small graphs. The second column is an estimation.

Time (μs)	Whole adjacency list	Average time per link	Single Link
EU	44.617	2.001	0.123
Indochina	88.771	3.391	0.182

Table 13.8: Checking individual links over Web graphs with the extended functionality of the k^2-tree representation.

page p to check if Web page q appears, which is the unique way to answer this query for most Web graph compression methods.

There are some other proposals supporting only direct navigation that also efficiently determine whether two nodes are connected. Apostolico and Drovandi's [AD09] technique, explained in Section 11.3.5, achieves an average time which is less than 60% of the retrieval time of the whole adjacency list. As we can observe in Table 13.8, our technique can answer a single link query order of magnitudes faster than retrieving the whole list. This comparative has been done using the smallest alternatives of k^2-tree over EU and Indochina graphs, which have been described in Section 13.3.1. The first column of Table 13.8 shows the time (in μs) required to retrieve the whole adjacency list for all the nodes in random order. The second column shows the average time per link when the whole list is computed, that is, we divide the value at the first column of the table by the number of retrieved links. The last column of the table shows the average time needed for checking all the links of the Web graph in random order. We can notice that checking individual links requires less time than the average time per retrieved link when the whole list is obtained. This is due to the fact that retrieving a whole list of a Web page may cause several unsuccessful top-down traversals over the tree, some of them complete traversals from the root node to a leaf of the k^2-tree if there is any 1 in the same leaf submatrix than the Web page of the query, due to direct neighbors of close Web pages, but the Web page has no links in that submatrix.

Time (μs)	Ours	AD ($\ell = 4$)
`EU`	0.123	1.192
`Indochina`	0.182	1.055

Table 13.9: Comparison between our proposal and Apostolico and Drovandi's technique when checking individual links.

Table 13.9 compares our proposal with Apostolico and Drovandi's technique (AD) by showing the average time to test the adjacency between pairs of random nodes, computed by checking all the pairs of nodes in random order. For our proposal we use the smallest alternatives of k^2-tree over `EU` and `Indochina` graphs, which have been described in Section 13.3.1 and occupy 3.47 bpe and 1.73 bpe respectively. The times reported for AD technique correspond to their fastest configurations (setting $\ell = 4$), which occupy 10.04 bpe and 5.30 bpe respectively when representing simultaneously the direct and the transposed graph in order to support direct and reverse neighbors, as our technique, and occupy 5.50 bpe and 3.05 bpe when supporting only direct neighbors retrieval. As we can observe from the results, our proposal tests the connectivity of pairs of random nodes around 5-10 times faster than AD technique, and it also requires significantly less space.

13.4.2 Range searches

We now show the performance of the range operation over the compact representation of Web graphs. We compare the time needed for retrieving $r > 0$ consecutive lists of direct neighbors starting with the direct neighbors list of Web page p up to the list of Web page $p + r - 1$, and the time spent to obtain all the hyperlinks in a range $[p, p + r - 1] \times [0, n]$ for all $p \in V$ in a random order, being both equivalent operations with the same result set of connected Web pages. As we can see in Figure 13.9, only for $r = 1$ (that is, when the range includes just 1 node, and it is equivalent to an adjacency list query) retrieving consecutive lists of direct neighbors obtains better time results than the range query, as the range query adds some overhead that is not compensated when only one list of neighbors is extracted. However, when $r > 1$ the range query obtains almost constant time results, while retrieving r consecutive list of neighbors increases linearly with r, as expected. These experiments were performed over the compressed k^2-representation of graph `Indochina`, using the alternative 2×2.

Therefore, whenever this type of range queries is needed, the k^2-tree representation of the Web graph is a suitable representation, since it can obtain the result more efficiently than retrieving several adjacency lists one by one as we have just showed. Moreover, a biggest benefit is obtained if our goal is to check if there exists

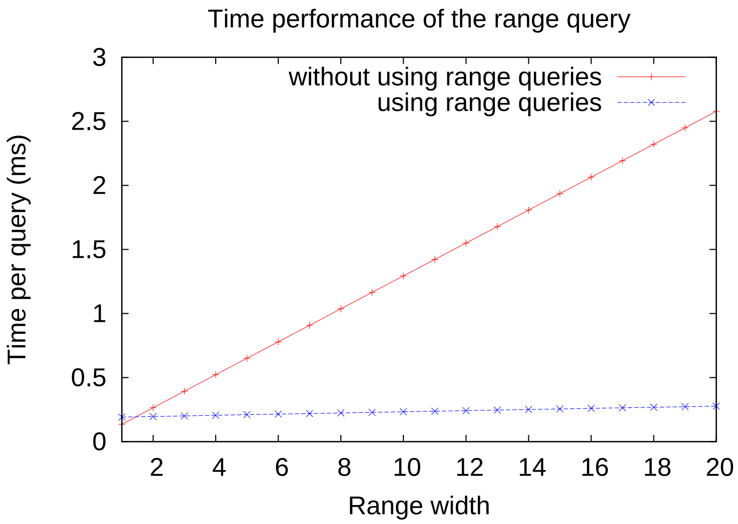


Figure 13.9: Range query performance compared to simple list retrieval query for different width of ranges.

a link from a range of pages $[p_1, p_2]$ to another $[q_1, q_2]$. In this case, as we have explained in Section 12.4.2, the k^2-tree can answer this query without the need of extracting completely any list of adjacency, and what is more, in case of a positive answer it can solve it before reaching the leaf node where the link is represented, which saves navigational time.

Figure 13.10 shows the average query time in milliseconds that is required to check if there is any link in the range $[p, p + r - 1] \times [0, n]$, with $1 \leq r \leq 20$ and $0 \leq p \leq n$, performed in random order. We compare this time with the time required by the range query that reports all the links existing in the same range, which was already shown in Figure 13.9. As we can see, the time required to find all the links inside a range increases with the range width, but moderately compared to the time to compute the neighbor lists individually, as we have seen in Figure 13.9. However, checking the existence of a link inside the same range can be performed significantly faster and the time required decreases as the range width increases. This is due to the fact that the existence of a link in a bigger range can be detected in a higher level of the tree and this avoids the navigation to lower levels of the tree. Hence, this operation becomes extremely faster over the k^2-tree technique, especially if we take into account that checking the existence of a link in a range must be performed by extracting the neighbors lists and checking if there is a link in the sought range.

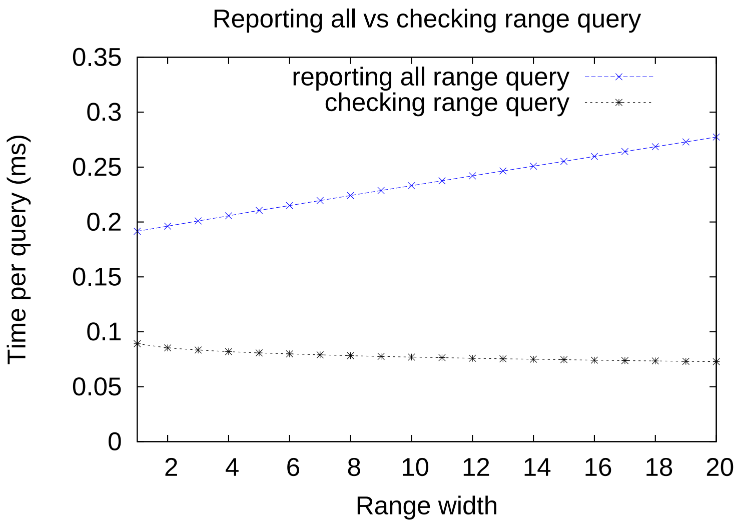


Figure 13.10: Checking the existence of a link in a range compared to finding all the links in the same range.

13.5 Comparison of the behavior between random graphs and Web graphs

The k^2-tree technique is especially designed to take advantage of the properties of Web graphs, that is, the similarity of the adjacency lists, the skewed distribution and the locality of reference. All these properties of Web graphs cause the sparseness and clustering of their adjacency matrix, which are exploited in order to obtain very compact representations of the Web graphs.

In this section we will show that the space and time analyses detailed in Sections 12.2.1.1 and 12.2.3.1 (calculated for uniformly distributed graphs) are too pessimistic for Web graphs. In order to prove this hypothesis, we create two graphs called `RandReord` and `DomainReord` that are equal to the original graph `EU` (denoted in this section as `Original`), but with a reordering of the identifiers of the Web pages. This reordering of the identifiers causes a reordering of the rows and columns in the adjacency matrix, since each row and each column of the adjacency matrix represents one Web page according to its identifier. Web pages are alphabetically sorted by URL in graph `Original` (graph `EU`), such that locality of reference is translated into closeness of the ones in the adjacency matrix. `RandReord` and `DomainReord` synthetic graphs are created as follows:

- Graph `DomainReord` tries to improve the locality of reference by the folklore idea of sorting the domains in reverse order, as then `aaa.bbb.com` will stay close to `zzz.bbb.com`. Hence, graph `DomainReord` is created from graph `EU` such that Web page identifiers are sorted according to this.

- Graph `RandReord` is obtained after a random permutation of the Web page identifiers. This reordering eliminates the locality of reference, such that ones in a row are no longer together, but distributed along all the row.

In addition, we also create a uniformly distributed graph with the same number of nodes and the same number of edges than graph `EU`. We denote this graph by `Uniform`. This graph does not preserve any of the properties of Web graphs: outdegrees of pages do not longer follow a skewed distribution and ones are spread along all the adjacency matrix, so there is no locality of reference nor similarity of adjacency lists.

Table 13.10 compares the behavior of the proposal for these four different graphs. First column indicates the graph and the representation configuration used to compress it. As we have detailed, we compare four graphs: `Original`, `DomainReord`, `RandReord` and `Uniform`, and we compress each of those four graphs with the k^2-tree technique using DACs in the leaf level with a fixed parameter $b = 4$. For all of them, we first partition the adjacency matrix in submatrices of size $2^{18} \times 2^{18}$ and then create one hybrid k^2-tree for each submatrix, where $k = 4$ for the first 5 levels of the tree and $k = 2$ for the rest. The subdivision is not continued until the last level, but the leaves of the last x levels are represented all together. For each graph, we create 3 representations where $x = 2, 3, 4$, that is, obtaining leaf submatrices of size 4×4, 8×8 and 16×16 respectively. Hence, if we denote k_L the value of k for the last level of the hybrid k^2-tree, then $k_L = 4$, $k_L = 8$ and $k_L = 16$ respectively for the 3 representations created.

The second column of the table shows the space (in bpe) obtained by each representation. According to the space analysis in Section 12.2.1.1, for $k = 2$ the space of the representation for uniformly distributed graphs is asymptotically twice the information-theoretic minimum necessary to represent all the matrices of $n \times n$ with m 1s, that is $\log \binom{n^2}{m}$, which is 15.25 for graph `EU` [CN10b]. As we can see in the table, the representation of the uniformly distributed graph occupies around 24-27 bpe, close to twice that value. In fact, using a 2×2 subdivision of the adjacency matrix, instead of using that hybrid approach, the space obtained would be 31.02. For `RandReord`, the random reordering of the identifiers of the Web pages eliminates some of the most important properties of Web graphs, so the space is also high. However, we can observe that on graphs `Original` and `DomainReord` the space is much better, as Web graphs are far from uniformly distributed and the k^2-tree technique takes advantage of this fact. The domain reordering slightly improves the compression ratio, since it is common that pages point to other pages inside the same domain, even if they do not share the same subdomain. These pages are distant in graph `Original` if their subdomains are not alphabetically close, but they are near to each other in graph `DomainReord`.

Third column indicates the size of the vocabulary of leaf submatrices, that is, the number of different non-zero $k_L \times k_L$ submatrices that appear in the adjacency matrix. Fourth column shows the length of the sequence of submatrices of the last level, that is, the total number of submatrices that are represented with DACs. We can observe that the vocabularies of leaves for `Original` and `DomainReord` are larger than for `RandReord` and `Uniform`. This happens since the adjacency matrices of these last two graphs have their ones spread all along the matrix, such that it is rare that several ones coincide in the same leaf submatrix. Hence, since there are very few possible submatrices in the leaf level, the vocabulary of submatrices is small, but the sequence of submatrices is larger. Moreover, due to the uniform distribution of the ones in the adjacency matrix, the distribution of frequency of the submatrices of the vocabulary is also uniform. Consequently, DACs cannot obtain a very compact representation of the sequence. On the contrary, the vocabulary of leaves for a Web graph, such as `Original` or `DomainReord`, is larger, but it follows a skewed distribution. Typical patterns such as horizontal, vertical, and diagonal runs are captured inside those leaf submatrices. Some of them appear more frequently than others, so they are encoded using fewer bits than less frequent submatrices. Hence, the space required for the leaves representation is lower than for random graphs.

We can also notice from the results of the fourth column that the total number of non-zero submatrices at the last level remains almost constant in case of a random graph, since ones are spread all along the matrix and they do not coincide in the same $k_L \times k_L$ submatrix, no matter if $k_L = 4$ or $k_L = 16$. In fact, the length of the sequence of leaf submatrices is close to the number of edges of the graph, that is, the total number of ones in the adjacency matrix, which is 19,235,140. However, the length of the sequence of leaf submatrices for the Web graphs `Original` and `DomainReord` is lower, and far from that total number of edges. Hence, ones are located together in the adjacency matrix. Yet, the number of total non-zero 16×16 submatrices when $k_L = 16$ is lower than the number of total non-zero 4×4 submatrices, since close ones coincide in the same big submatrix.

The last column of the table shows the efficiency of direct neighbors retrieval by measuring the average time per neighbor retrieved in μs/e. During the time analysis of the proposal in Section 12.2.3.1, we have already anticipated that navigation over Web graphs would be more efficient than over random graphs. This is due to the fact that the k^2-tree has numerous leaves in the case of `RandReord` and `Uniform` graphs, as we can see in the table, so this implies that the retrieval of all the neighbors of the graph must traverse all those leaves to return usually just one neighbor per submatrix, whereas in the case of a Web graph, a leaf submatrix can be visited once to answer several neighbors of the same page, which reduces the navigation time. Hence, the total navigation time to retrieve all the neighbors of all the pages in a

	Space (bpe)	Leaves Voc.	# Leaves	Time (μs/e)
`Original` $k_L = 4$	4.04	12,913	6,273,036	2.057
`DomainReord` $k_L = 4$	4.03	13,054	6,276,012	1.994
`RandReord` $k_L = 4$	25.79	226	18,800,628	48.774
`Uniform` $k_L = 4$	27.71	136	19,231,353	53.315
`Original` $k_L = 8$	3.56	209,901	3,344,592	2.037
`DomainReord` $k_L = 8$	**3.55**	210,385	3,341,643	**1.964**
`RandReord` $k_L = 8$	25.24	3,585	18,514,330	49.300
`Uniform` $k_L = 8$	27.30	2,085	19,219,329	53.834
`Original` $k_L = 16$	7.88	455,955	1,716,719	2.044
`DomainReord` $k_L = 16$	7.86	454,691	1,704,056	1.982
`RandReord` $k_L = 16$	23.23	78,294	18,109,891	50.707
`Uniform` $k_L = 16$	24.43	28,345	19,171,732	54.498

Table 13.10: Effect of the reordering of the nodes and behavior of uniformly distributed graphs.

random order is considerably lower in the case of Web graphs than in the case of random graphs.

Moreover, as we have seen in Section 12.2.3.1, the navigation time to retrieve a list of direct or reverse neighbors has no worst-case guarantees better than $O(n)$. For a random uniformly distributed matrix, this analysis was refined to $O(\sqrt{m})$, but we guessed (and confirmed in Table 13.10) that the average performance for a Web graph would be better, due to the clustering and sparseness of its adjacency matrix. In addition, we also expected that the time to retrieve the neighbors of a Web page would depend on the length of the output list, whereas the time to retrieve the adjacency list of a node in a random graph would be slower and independent of the length of the output list. This is due to the fact that the tree is close to complete in all its levels, as ones are spread all along the adjacency matrix. Consequently, when extracting the adjacency list of a node of a uniformly distributed graph, numerous branches and leaves are traversed worthlessly, since they are the result of a subdivision prompted by a near 1, but not by a neighbor of the node. Thus, an almost stable time of $O(\sqrt{m})$ is needed to answer the query. On the contrary, the k^2-tree of a Web graph is not very branchy, since large areas of zeroes are represented in a leaf node at an early level of the tree. Hence, when looking for neighbors of a Web page, the navigation is guided towards the leaves where the neighbors of the Web page are.

In order to confirm these hypotheses, we ran some experiments over the graphs `Original`, `RandReord` and `Uniform`. We compute the time to retrieve the adjacency

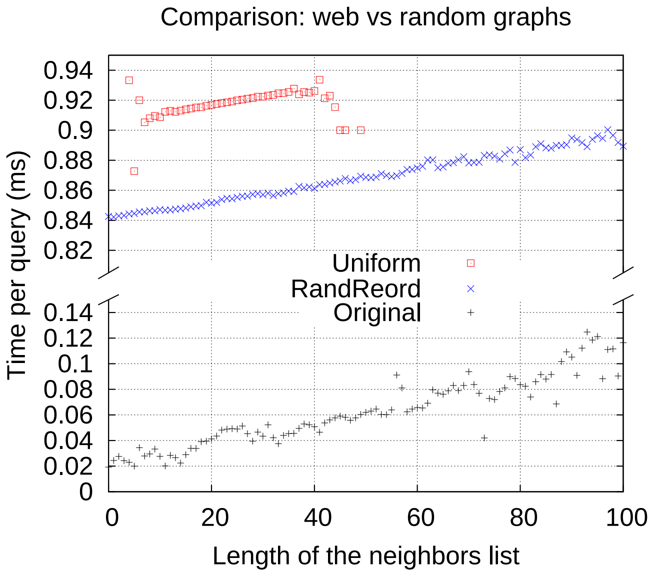


Figure 13.11: Adjacency list retrieval time (in ms) for Web graphs and random graphs.

list of each node, for all the nodes of the graphs, in a random order. We illustrate those results depending on the length of the list in Figure 13.11. The y axis has been cut in order to properly visualize the curves, since results are considerably faster for graph `Original` than for `RandReord` and `Uniform`, as we have already seen in Table 13.10. In addition, we can observe that the time to retrieve an adjacency list of graph `Original` depends linearly on the list length, while the time for random graphs `RandReord` and `Uniform` increases with the list length but in a very moderate way, in relative terms. This dependency on the list length can be better seen in Figure 13.12, where the average query time to retrieve an adjacency list is divided by the length of the list, that is, we measure the average time to retrieve a neighbor and visualize it depending on the length of the adjacency list.

13.6 Discussion

In this chapter we have tested our proposal, the k^2-tree technique, over different Web graphs. We have studied different variants presented in the previous chapter and compared them with the compression methods of the literature that support both direct and reverse navigation over the graph.

We have concluded that the best space and time results for the k^2-tree technique are obtained when large k values are used at the top levels of the tree and also for

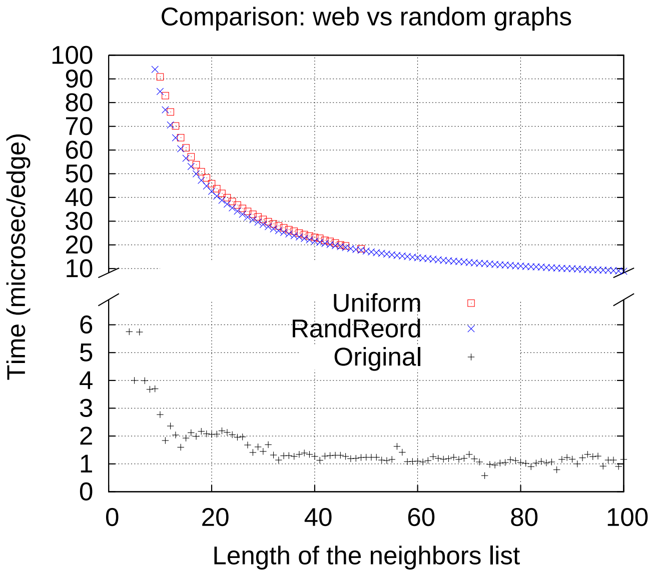


Figure 13.12: Direct Neighbor retrieval time (in μ/e) for Web graphs and random graphs.

the leaf level, where DACs are used to represent the sequence of leaf submatrices in a compact way. Compared to the state-of-the-art methods, the k^2-tree technique achieves some interesting results both in time and space. We achieve the smallest space reported in the literature, while retrieving direct and reverse neighbors in an efficient time.

Even though other methods, such as those presented by Claude and Navarro, Boldi and Vigna or Apostolico and Drovandi (representing both graphs, original and transpose), can be faster at retrieving direct and reverse neighbors, our method always needs fewer bits per edge. Saving space is crucial in order to apply a representation to real Web graphs. If we need less space to represent a small Web graph, we will be able to operate faster in main memory with a larger one. We save I/O accesses that spend much more time than the difference between our proposal and the ones by Claude and Navarro, Boldi and Vigna or Apostolico and Drovandi. In addition, we can perform other operations such as range searches or checking whether two Web pages are connected or not without extracting the whole adjacency list.

Chapter 14

Discussion

14.1 Main contributions

Compressed graph representations allow running graph algorithms in main memory on much larger subsets than classical graph representations. Since the Web can be seen as a huge graph, compressed representation of Web graphs are essential to run algorithms that extract information from the Web structure in an efficient way.

We have introduced a compact representation for Web graphs that takes advantage of the sparseness and clustering of their adjacency matrix. Our representation is a particular type of tree, which we call the k^2-tree, that enables efficient forward and backward navigation in the graph (a few microseconds per neighbor found) within compact space (about 2 to 5 bits per link). We have presented several variants of the method with different space requirements and time results, and have shown the appropriate parameter tuning to obtain the smallest representation of the Web graph, and also how the navigation can be improved.

Our experimental results show that our technique offers an attractive space/time tradeoff compared to the state of the art. We achieve the smallest graph representation reported in the literature that supports direct and reverse navigation in efficient time. Moreover, we support queries on the graph that extend the basic forward and reverse navigation. For instance, it is possible to check if one Web page has a link to another Web page without retrieving the whole list of direct neighbors. It is also possible to recover all the connected pairs of Web pages inside a range in a very efficient way.

14.2 Other Applications

The k^2-tree technique was originally designed to represent Web graphs in a very compact way. We have already shown in the experimental evaluation that its efficiency decreases for uniformly distributed graphs. However, the k^2-tree can be employed to represent other kind of graphs whose adjacency matrix also exhibits sparseness and clustering properties, apart from Web graphs. For example, we could use our proposal to compress social networks and compare its performance with other solutions of the literature for this scenario [CKL$^+$09]. Moreover, it can be generalized to represent any binary relation.

Binary relations are an abstraction to represent the relation between the objects of two collections of different nature. They can be used in several low-level structures within a more complex information retrieval system, or even replace one of the most used ones: an inverted index can be regarded as a binary relation between the vocabulary of terms and the documents where they appear. The k^2-tree technique can be directly applied over the relation matrix of the binary relation, achieving a navigable representation in a compact space. Our proposal may implement several operations among those included in the extended set of primitives of interest in applications of binary relation data structures proposed by Barbay *et al.* [BCN10].

In addition, the k^2-tree representation can be the basis for a new method to represent a graph database, where graphs are not as simple as Web graphs, but rather have types, attributes, and multiedges. We present a preliminary proposal of this application in the next section.

14.2.1 A Compact Representation of Graph Databases

Graph databases have emerged as an alternative data model with applications in many complex domains. Typically, the problems to be solved in such domains involve managing and mining huge graphs. The need for efficient processing in such applications has motivated the development of methods for graph compression and indexing. However, most methods aim at an efficient representation and processing of simple graphs (without attributes in nodes or edges, or multiple edges for a given pair of nodes). A recent proposal [ÁBLP10] presents a model for compact representation of general graph databases. The goal is to represent any labeled, directed, attributed multigraph.

The proposal consists in a new representation of a graph database based on the k^2-tree technique, which obtains very compact space enabling any kind of navigability over the graph. The k^2-tree method, which is designed for simple directed graphs, or more generally, for any binary relation between two sets, cannot be directly applied to represent any labeled, directed, attributed, multigraph G. Then, a complex data structure is proposed, called Compact Graph Database (CGD), which represents any graph G as a combination of three k^2-trees and some extra information. The relations that are represented using the k^2-tree technique are the following:

- The binary relation between the nodes and their attribute values. Let the nodes be the rows of a matrix, and all the possible values of all the attributes be the columns of that matrix. Then, a cell of the matrix will contain a 1 if the node of the row has the attribute value of the column.
- Analogously to the relation between nodes and attribute values, the relation between the edges and their attribute values is also represented using the k^2-tree technique.
- More intuitive is the k^2-tree representation of the relation between the nodes of the graph, that is, the edges of the graph. Since the graph is a multigraph, some extension of the original method of k^2-tree is needed in order to store the multiple edges between the same pair of source and target nodes.

The algorithms to answer typical queries over graph databases (e.g. select, getValue, etc) are detailed in this paper [ÁBLP10] and space and time performance is measured over two datasets taken from real domains: Wikipedia and Youtube. The difference in space requirements is significant compared to other graph database systems. The proposal achieves compression rates around 50% between the compact representation and the raw text representation of the database, while others needs more than twice the size of the raw representation. The compression ratio achieved affects the navigation performance, yet time results are still competitive. Hence, this proposal represents an attractive alternative due to the compression rates it achieves and the efficiency on query resolution.

14.3 Future work

Our proposal exploits the properties of the adjacency matrix, yet with a general technique to take advantage of clustering rather than a technique tailored to particular Web graphs. We introduce a compact tree representation of the matrix that not only is very efficient to represent large empty areas of the matrix, but at the same time allows efficient forward and backward navigation. An elegant feature of our solution is that it is symmetric, in the sense that forward and backward navigation are carried out by similar means and achieve similar times. Due to the properties of this general technique, we believe that it can be applied to several domains where general binary relations can express the relations between the objects involved. For instance, we can consider the relation between documents and terms (keywords) in those documents, so that we can represent an index of the text collection with our proposal. One interesting example could be the representation of discrete grids of points, for computational geometry applications or geographic information systems.

Following this idea, we will study the application of our proposal to construct new index structures or retrieval algorithms that take into account the spatial nature of geographic references embedded within documents. These scenarios may not present the same distribution as Web graphs, such as the locality of references and clustering exhibited by the adjacency matrix where Web pages are sorted according to the URL ordering, which has been probed to be the most efficient technique for assigning identifiers in the case of Web Search Engines [Sil07]. Yet, several sparse matrix reordering schemes, such as Reverse Cuthill- McKee and King's algorithms [CM69, Kin70], can be studied in order to improve the compression and navigation times.

We also plan to extend our work by considering more complex navigation algorithms over the graph. We have presented some basic and extended functionality, such as retrieving the direct and reverse neighbors of a node, checking whether there exists a link from one Web page to another, or retrieving all the connected pairs in a range of node identifiers. More complex algorithms can be run over the graphs using these basic operations. These algorithms might be natively implemented using the k^2-tree data structure, outperforming the behavior of a naive implementation of the algorithm using the basic operations. Several algorithms to solve classical graph problems, such as obtaining the shortest path or minimum cuts in a graph, can be considered and implemented.

We have proposed a static data structure, the k^2-tree, to represent any Web graph in very compact space. The tree is stored levelwise using static bitmaps. Deleting a link between two Web pages can be performed by just changing the bit to zero in the cell of the last level of the tree and also in upper levels of the tree if the subtree represented with that bit represented the only link that is being deleted. Hence, the cell is marked as deleted but no structural modifications of the tree are performed. This procedure might not return an optimal k^2-tree, since the space of the data structure is maintained and it could be reduced. However, it is an accurate representation of the Web graph and it can be navigated with the described algorithms. Deleting Web pages is done in a similar way, by deleting the links that are pointed by or point to that Web page. The problem arises if new links or Web pages are added. If a link is added such that one new 1 is placed in the adjacency matrix, and that 1 is surrounded by others 1s in the same $k \times k$ matrix, then the link can be easily added by just changing the 0 to a 1 in the leaf matrix. However, if it becomes the only 1 in the $k \times k$ matrix at the last level of the tree, a new leaf must be created in the last level of the tree, and also its corresponding path from an upper level of the tree. This would require the insertion of some nodes in the tree and hence, the insertion of the representation of those nodes in the compact representation of the tree. Therefore, some bits would be inserted in the middle of the bitmaps that represent each level of the tree, which is not supported by the data structures used for the bitmaps (we use a static representation). Even though dynamism is not a vital characteristic for compression methods focused on Web graphs, it may be an interesting feature, especially if we use our technique in other scenarios. Hence, we plan to study how to modified our data structure in order to support dynamism.

The improvement of the time efficiency of our proposal is also a goal for continuing our research. A recent joint work with Claude [CL11] consisted in combining our k^2-trees and the RePair-Graph [CN10c]. The new proposal takes advantage of the fact that most links are intra-domain, and represents the intra-domain links using separate k^2-trees (with a common leaf submatrices vocabulary), and the inter-domain links using a RePair-based strategy. This new representation, called k^2-partitioned, significantly improves the time performance of k^2-tree while almost retaining the compression ratio. Hence, it achieves very compact spaces, smaller than the rest of the techniques except for the k^2-tree, obtaining very competitive time results.

Part IV

Thesis Summary

Chapter 15

Conclusions and Future Work

15.1 Summary of contributions

The amount of digital data has been constantly growing since the birth of the first computer. As the storage capacity and processing speed increase, larger volumes of data must be manipulated. This data usually contains text, images or even multimedia information such as music and video. Therefore, processing massive datasets and extracting relevant information from them have become attractive challenges in the field of computer science.

Some research has focused its efforts on studying new approaches to effectively store information and support efficient query and modification, using the minimum amount of space as possible. We are interested in compressed representations of the data, where we can perform complex operations directly on the compact representation. These representations can even obtain enhanced functionality which is not offered by the plain representation of the data.

In this thesis we have addressed the problem of the efficiency in Information Retrieval by presenting some new general low-level data structures and algorithms that can be used in several applications. We experimentally showed that these structures obtain interesting space/time tradeoffs compared to other techniques commonly used in those domains. The methods we presented, all of them conceived to operate in main memory, were developed upon one base idea: since they are compact data structures, they allow to represent large volumes of data in higher and faster levels in the memory hierarchy.

This section summarizes the main contributions of this thesis:

- We have presented *Directly Addressable Codes* (DACs), a new variable-length encoding scheme for sequences of integers that, in addition to represent the sequence in compact space, enables fast direct access to any position of the encoded sequence. We have also proposed an optimization algorithm that computes the most compact configuration of our codes given the frequencies distribution of the integers of the sequence to encode. Moreover, we have presented a rearrangement strategy that can be applied over the encoded sequences obtained by any variable-length encoding and provides direct access to any element of the sequence by just adding some bitmaps over the sequence.

We have shown that the technique is simple and competitive in time and space with existing solutions in several applications, such as the representation of LCP arrays or high-order entropy-compressed sequences. It becomes a very attractive solution when just direct access to the encoded sequence is required, comparing this technique with classical solutions to provide direct access, such as the use of sparse or dense samplings over the sequence.

Several recent implementations of classical data structures, such as compressed suffix trees or PATRICIA trees, can benefit from the efficiency of our data structure. When direct access is required over a sequence of non-uniformly distributed integers, especially if most of them are small, but some of them are larger, hence, our variable-length encoding scheme becomes the preferred choice to obtain a very fast access to a very compact representation of the integers.

- We have proposed the *Byte-Oriented Codes Wavelet Tree* (BOC-WT), a new data structure that permits the compact representation and efficient manipulation of natural language text. This tree-shaped structure maintains the properties of the compressed text obtained by any word-based, byte-oriented prefix-free encoding technique, that is, it maintains the same compression ratio and comparable compression and decompression times, and in addition it drastically improves searches.

  The proposed data structure can be considered as a word-based self-indexed representation of the text, which occupies a space proportional to the compressed text (31%-35% of the size of the original text) and searches are performed in time independent of the text length. BOC-WT obtains efficient time results for counting, locating and extracting snippets when searching for a pattern in a text. Compared to classical inverted indexes, it obtains interesting results when the space usage is not high. By adding a small extra structure to BOC-WT, searching is considerably improved and it competes successfully with block-addressing inverted indexes that take the same space on top of the compressed text. Compared to other word-based self-indexes, our data structure obtains better times when searching for individual words or extracting portions of text. Searching long phrase patterns is performed more efficiently by other self-indexes, however, BOC-WT is still the preferred choice for locating and displaying the occurrences of short phrases composed of two words.

- Finally, we have proposed k^2-*tree*, a new compact tree-shaped representation for Web graphs which supports basic navigation over the Web graph, that is, retrieving the direct and reverse list of neighbors of a page, in addition to some interesting extra functionality. For instance, it is possible to check if one Web page has a link to another Web page without retrieving the whole list of direct neighbors. It is also possible to recover all the connected pairs of Web pages inside a range in a very efficient way.

  We present several variants of our technique. One of them includes a compact representation of the leaves of the tree encoded using our first contribution, the Directly Addressable Codes, which improves simultaneously both time and space results. The experimental evaluation of our technique shows that this variant achieves the smallest graph representation reported in the literature that supports direct and reverse navigation in efficient time, and our proposal offers an interesting space/time tradeoff when varying the configuration of parameters. Our representation enables efficient forward and backward navigation in the graph (a few microseconds per neighbor found) within compact space (about 2 to 5 bits per link).

15.2 Future work

In this section we detail some future plans after this thesis. We will describe the most interesting ones for each contribution.

- The Directly Addressable Codes can be applied in many different domains. They are especially designed to enable direct access to any element of a compressed sequence, so it can be used in lots of data structures. We plan to study the feasibility and suitability of our proposal to other well-developed scenarios, such as the compression of inverted lists and natural language texts.

  Moreover, the rearrangement strategy used has been described as a contribution by itself. Hence, we will compare this rearrangement with the classical solutions when providing direct access to non-statistical variable-length encodings.

- The second data structure presented in this thesis, the Byte-Oriented Codes Wavelet Tree, presents a most consolidated status. We have shown the behavior of our proposal built over compressed natural language text with Plain Huffman. We can extend our proposal to allow for more flexible searching. For instance, we might want to find phrases regardless of whether the words are separated by a space, two spaces, a tab, a newline, etc. Moreover, we can consider only stemmed words for those searches or we can omit stopwords such as articles or prepositions. Even if all this functionality is supported, our data structure must reproduce the original text, returning the variants of the stemmed words or the stopwords when needed. Hence, different vocabularies can be distinguish to encode differently words from separators, similarly to the strategy used to represent XML documents.

In addition, we plan to study the convenience of our data structure to represented in a self-indexed form other kinds of documents, not only natural language text or XML documents, and study ways to provide dynamism to the structure.

- We have already stated that our k^2-tree data structure can be apply to several domains, considering any binary relation as a graph, such that the k^2-tree can represent the associated relation matrix. Since general binary relations may not present the same distribution as Web graph, such as the locality of references and clustering presented in the adjacency matrix, we plan to study several sparse matrix reordering schemes to improve the overall performance of our technique.

  We also plan to extend our work by considering more complex navigation algorithms over the graph. We will natively implement several algorithm to solve classical graph problems, such as obtaining the shortest path or minimum cuts in a graph. Another interesting line of research can be to study a dynamic version of our proposal. k^2-tree is a static data structure which allows any deletion of a link or a Web page, but only some insertions can be performed. Hence, we will study if the k^2-tree data structure can be modified in order to support dynamism.

Appendix A

Publications and other research results

This chapter summarizes the publications and research stays of the author directly related with this thesis. For each publication, we include references to relevant works in which it has been cited (these citations were updated by March 2011).

Publications

Journals

- Välimäki, N., Ladra, S., Mäkinen, V. Approximate All-Pairs Suffix/Prefix Overlaps. In *Information and Computation* (To appear).

- Fariña, A., Ladra, S., Pedreira, O., Places, A. S. Rank and select for succinct data structures. *Electronic Notes in Theoretical Computer Science, 236, pp. 131-145, 2009.*

International conferences

- Välimäki, N., Ladra, S., Mäkinen, V. Approximate All-Pairs Suffix/Prefix Overlaps. In *Proc. of the 31st Annual Symposium on Combinational Pattern Matching (CPM)* - LNCS 6129, pp.76–87. New York, USA, 2010.

- Álvarez, S., Brisaboa, N. R., Ladra, S., Pedreira, O. A Compact Representation of Graph Databases. In *Proc. of the Eighth Workshop on Mining and Learning with Graphs (MLG)*, pp. 18–25. Washington D.C., USA, 2010.

- Brisaboa, N. R., Ladra, S., Navarro, G. k2-trees for Compact Web Graph Representation. In *Proc. of the 16th International Symposium on String Processing and Information Retrieval (SPIRE)* - LNCS 5721, pp. 18–30. Saariselkä, Finland, 2009.

  This paper has been cited by:

  - Claude, F., Navarro, G. (2010b). Fast and Compact Web Graph Representations. In *ACM Transactions on the Web* 4(4):article 16, 2010.
  - Claude, F., Navarro, G. (2010a). Extended Compact Web Graph Representations. In *Algorithms and Applications (Ukkonen Festschrift)*, LNCS 6060, pp. 77–91, 2010.
  - Grabowski, S., Bieniecki, W. (2010). Tight and Simple Web Graph Compression. In *Proc. Prague Stringology Conference 2010 (PSC)*, pp. 127–137, 2010.
  - Boldi, P., Santini, M., Vigna, S. (2010). Permuting web and social graphs. In *Internet Mathematics*, 6(3):257–283, 2010.

- Brisaboa, N. R., Ladra, S., Navarro, G. Directly Addressable Variable-Length Codes. In *Proc. of the 16th International Symposium on String Processing and Information Retrieval (SPIRE)* - LNCS 5721, pp. 122–130. Saariselkä, Finland, 2009.

  This paper has been cited by:

  - Kreft, S., Navarro. G. (2011). Self-Indexing based on LZ77. In *Proc. of the 32nd Annual Symposium on Combinational Pattern Matching (CPM)*, 2011. (To appear)
  - Brisaboa, N. R., Cánovas, R., Claude, F., Martínez-Prieto, M. A., Navarro, G. (2011). Compressed String Dictionaries. In *Proc. of the 10th International Symposium on Experimental Algorithms (SEA)*, 2011. (To appear)
  - Teuhola, J. (2010). Interpolative Coding of Integer Sequences Supporting Log-Time Random Access. In *Information Processing and Management.*
  - Cánovas, R., Navarro, G. (2010). Practical Compressed Suffix Trees. In *Proc. of the 9th International Symposium on Experimental Algorithms (SEA)* - LNCS 6049, pp. 94–105, 2010.
  - Sirén, J. (2010). Sampled Longest Common Prefix Array . In *Proc. of the 31st Annual Symposium on Combinational Pattern Matching (CPM)* - LNCS 6129, pp.227–237, 2010.
  - Conway, T. C., Bromage, A. J. (2010). Succinct Data Structures for Assembling Large Genomes. In *Proc. of the 9th International Symposium on Experimental Algorithms (SEA)* - LNCS 6049, pp. 94–105, 2010.

- Brisaboa, N. R., Fariña, A., Ladra, S., Navarro, G. Reorganizing compressed text. In *Proc. of the 31th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'08)*, pp. 139–146. Singapore, 2008.

  This paper has been cited by:

  - Ferragina, P., Manzini, G. (2010). On compressing the textual web. In *Proc. of the 3rd ACM International Conference on Web Search and Data Mining (WSDM)*, pp. 391–400, 2010.
  - Arroyuelo, D., González, S., Oyarzún, M. (2010). Compressed Self-Indices Supporting Conjunctive Queries on Document Collections. In *Proc. of the 17th International Symposium on String Processing and Information Retrieval (SPIRE)* - LNCS 6393, pp. 43–54, 2010.
  - Brisaboa, N. R., Cerdeira, Navarro, G., Pasi, G. (2010). An Efficient Implementation of a Flexible XPath Extension. In *Proc. of the 9th International Conference on Adaptivity, Personalization and Fusion of Heterogeneous Information (RIAO)*, pp. 140–147, 2010.
  - Ferragina, P., González, R., Navarro, G., Venturini, R. (2009). Compressed Text Indexes: From Theory to Practice. In *ACM Journal of Experimental Algorithmics (JEA)* 13:article 12, 2009.
  - Barbay, J., Navarro, G. (2009). Compressed Representations of Permutations, and Applications. In *Proc. of the 26th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pp. 111–122, 2009.
  - Brisaboa, N. R., Cerdeira, Navarro, G., Pasi, G. (2009). A Compressed Self-Indexed Representation of XML Documents. In *Proc. of the European Conference on Digital Libraries (ECDL)* - LNCS 5714, pp. 273–284, 2009.
  - Claude, F., Navarro, G. (2008) Practical Rank/Select Queries over Arbitrary Sequences. In In *Proc. of the 15th International Symposium on String Processing and Information Retrieval (SPIRE)* - LNCS 5280, pp. 176–187, 2008.
  - Brisaboa, N. R., Fariña, A., Navarro, G., Places, A. S., Rodríguez, E. (2008) Self-Indexing Natural Language. In In *Proc. of the 15th International Symposium on String Processing and Information Retrieval (SPIRE)* - LNCS 5280, pp. 121–132, 2008.

- Brisaboa, N. R., Cillero, Y., Fariña, A., Ladra, S., Pedreira, O. A New Approach for Document Indexing UsingWavelet Trees. In *Proc. of the 18th International Workshop on Database and Expert Systems Applications (DEXA)*, pp. 69–73. Regensburg, Germany, 2007.

This paper has been cited by:

- Brisaboa, N. R., Luaces, M. R., Navarro, G., Seco. D. (2009). A New Point Access Method based on Wavelet Trees. In *Proc. of the 3rd International Workshop on Semantic and Conceptual Issues in Geographic Information System (SeCoGIS)* - LNCS 5833, pp. 297–306, 2009.

- Cillero, Y., Ladra, S., Brisaboa, N. R., Fariña, A., Pedreira, O. Implementing byte-oriented rank and select operations. In *Proc. of SOFSEM SRF: Current Trends in Theory and Practice of Computer Science (SOFSEM) Student Research Forum*, pp. 34–45. High Tatras, Slovakia, 2008.

National conferences

- Álvarez, S, Brisaboa, N. R., Ladra, S., Pedreira, O. Almacenamiento y explotación de grandes bases de datos orientadas a grafos. In *Actas de las XV Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, pp. 187–197. Valencia, 2010.

- Brisaboa, N. R., Fariña, A., Ladra, S., Places, A. S., Rodríguez, E. Indexación y autoindexación comprimida de documentos como base de su procesado. In *Actas del I Congreso Español de Recuperación de Información (CERI)*, pp. 137–148. Madrid, 2010.

- Fariña, A.; Ladra, S., Paramá, J. R., Places, A. S., Yáñez-Miragaya, A.: Mejorando la búsqueda directa en texto comprimido. In *Actas del I Congreso Español de Recuperación de Información (CERI)*, pp. 283–290. Madrid, 2010.

- Álvarez, S., Cerdeira-Pena, A., Fariña, A., Ladra, S. Desarrollo de un compresor PPM orientado a palabra. In *Actas de las XIV Jornadas de Ingeniería del Software y Bases de Datos (JISBD)*, pp. 225–236. San Sebastián, 2009.

- Brisaboa, N. R., Cillero, Y., Fariña, A., Ladra, S., Pedreira, O. Indexación de textos utilizando Wavelet Trees. In *Actas de los Talleres de las Jornadas de Ingeniería del Software y Bases de Datos (TJISBD)*, 1(7), pp. 37–46. Zaragoza, 2007.

Research stays

- *February 15th, 2008 – July 31st, 2008.* Research stay at Universidad de Chile (Santiago, Chile), under the supervision of Prof. Gonzalo Navarro.

- *August 7th, 2009 – November 7th, 2009.* Research stay at University of Helsinki (Finland), under the supervision of Prof. Veli Mäkinen.

Bibliography

[ÁBLP10] S. Álvarez, N. R. Brisaboa, S. Ladra, and O. Pedreira. A compact representation of graph databases. In *Proc. of the 8th Workshop on Mining and Learning with Graphs (MLG)*, pages 18–25, 2010.

[Abr63] N. Abramson. *Information Theory and Coding.* McGraw-Hill, 1963.

[AD09] A. Apostolico and G. Drovandi. Graph compression by BFS. *Algorithms*, 2(3):1031–1044, 2009.

[AGO10] D. Arroyuelo, S. González, and M. Oyarzún. Compressed self-indices supporting conjunctive queries on document collections. In *Proc. of the 17th Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 6393, pages 43–54, 2010.

[AIS93] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 207–216, 1993.

[AM01] M. Adler and M. Mitzenmacher. Towards compressing Web graphs. In *Proc. of the 11th Data Compression Conference (DCC)*, pages 203–212, 2001.

[AMN08] Y. Asano, Y. Miyawaki, and T. Nishizeki. Efficient compression of web graphs. In *Proc. 14th Annual International Conference on Computing and Combinatorics (COCOON)*, LNCS 5092, pages 1–11, 2008.

[AN10] D. Arroyuelo and G. Navarro. Practical approaches to reduce the space requirement of lempel-ziv-based compressed text indices. *ACM Journal of Experimental Algorithmics (JEA)*, 15(1.5), 2010.

[BBH$^+$98] K. Bharat, A. Broder, M. Henzinger, P. Kumar, and S. Venkatasubramanian. The Connectivity Server: Fast access to linkage information on the Web. In *Proc. of the 7th World Wide Web Conference (WWW)*, pages 469–477, 1998.

[BC08] G. Buehrer and K. Chellapilla. A scalable pattern mining approach to web graph compression with communities. In *Proc. 1st ACM International Conference on Web Search and Data Mining (WSDM)*, pages 95–106, 2008.

[BCF$^+$11] N. R. Brisaboa, F. Claude, A. Fariña, G. Navarro, A. Places, and E. Rodríguez. Word-based self-indexes for natural language text. *ACM Transactions on Information Systems (TOIS), Submitted*, 2011.

[BCN09] N. R. Brisaboa, A. Cerdeira, and G. Navarro. A compressed self-indexed representation of XML documents. In *Proc.of the 13th European Conference on Digital Libraries (ECDL)*, LNCS 5714, pages 273–284, 2009.

[BCN10] J. Barbay, F. Claude, and G. Navarro. Compact rich-functional binary relation representations. In *Proc. of the 9th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 6034, pages 170–183, 2010. To appear.

[BCSV04] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. *Software: Practice and Experience (SPE)*, 34(8):711–726, 2004.

[BCW84] T. Bell, J. Cleary, and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[BCW90] T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice Hall, New Jersey, 1990.

[BDM$^+$05] D. Benoit, E. Demaine, I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

[BFLN08] N. R. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. of the 31th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 139–146, 2008.

[BFN$^+$08] N. R. Brisaboa, A. Fariña, G. Navarro, A. Places, and E. Rodríguez. Self-indexing natural language. In *Proc. of the 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 121–132, 2008.

[BFNE03] N. R. Brisaboa, A. Fariña, G. Navarro, and M. Esteller. (s,c)-dense coding: An optimized compression code for natural language text databases. In *Proc. of the 10th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 2857, pages 122–136, 2003.

[BFNP07] N. R. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10:1–33, 2007.

[BGMR06] J. Barbay, A. Golysnki, I. Munro, and S. Srinivasa Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. In *Proc. of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, number 4009 in LNCS, pages 24–35, 2006.

[BHMR07] J. Barbay, M. He, I. Munro, and S. Srinivasa Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In *Proc. of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689, 2007.

[BINP03] N. R. Brisaboa, E. Iglesias, G. Navarro, and J. Paramá. An efficient compression code for text databases. In *Proc. of the 25th European Conference on Information Retrieval Research (ECIR)*, LNCS 2633, pages 468–481, 2003.

[BKM$^+$00] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the Web. *Computer Networks*, 33(1–6):309–320, 2000. Also in *Proc. 9th World Wide Web Conference (WWW).*

[BLN09a] N. R. Brisaboa, S. Ladra, and G. Navarro. Directly addressable variable-length codes. In *Proc. of the 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5721, pages 122–130, 2009.

[BLN09b] N. R. Brisaboa, S. Ladra, and G. Navarro. K2-trees for compact web graph representation. In *Proc. of the 16th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5721, pages 18–30, 2009.

[BM77] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM (CACM)*, 20(10):762–772, October 1977.

[BRSV11] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proc. of the 20th international conference on World Wide Web (WWW)*, 2011.

[BSTW86] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM (CACM)*, 29(4), 1986.

[BV04] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. of the 13th International World Wide Web Conference (WWW)*, pages 595–601, 2004.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Systems Research Center, 1994. http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/.

[BYRN99] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman, May 1999.

[CFMPN10] F. Claude, A. Fariña, M. Martínez-Prieto, and G. Navarro. Compressed q-gram indexing for highly repetitive biological sequences. In *Proc. of the 10th IEEE Conference on Bioinformatics and Bioengineering (BIBE)*, pages 86–91, 2010.

[CHLS07] H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms (TALG)*, 3(2):article 21, 2007.

[CKL$^+$09] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proc. of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 219–228, 2009.

[CL11] F. Claude and S. Ladra. Practical representations for web and social graphs. In *Proc. of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), submitted*, 2011.

[Cla96] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996.

[CM69] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. of the 24th ACM National Conference*, pages 157–172, 1969.

[CM05] J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In *Proc of the 12th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 3772 of *LNCS*, pages 1–12, 2005.

[CM06] J. S. Culpepper and A. Moffat. Phrase-based pattern matching in compressed text. In *Proc. of the 13th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 4209 of *LNCS*, pages 337–345, 2006.

[CM07] J. S. Culpepper and A. Moffat. Compact set representation for information retrieval. In *Proc. of the 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 137–148, 2007.

[CN08] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. of the 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187, 2008.

[CN09] F. Claude and G. Navarro. Self-indexed text compression using straight-line programs. In *Proc. of the 34th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, LNCS 5734, pages 235–246, 2009.

[CN10a] R. Cánovas and G. Navarro. Practical compressed suffix trees. In *Proc. of the 9th International Symposium on Experimental Algorithms (SEA)*, LNCS 6049, pages 94–105, 2010.

[CN10b] F. Claude and G. Navarro. Extended compact web graph representations. In T. Elomaa, H. Mannila, and P. Orponen, editors, *Algorithms and Applications (Ukkonen Festschrift)*, LNCS 6060, pages 77–91, 2010.

[CN10c] F. Claude and G. Navarro. Fast and compact web graph representations. *ACM Transactions on the Web (TWEB)*, 4(4):article 16, 2010.

[CW84] John G. Cleary and Ian H. Witten. Data compression using Adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[DMLT05] D. Donato, S. Millozzi, S. Leonardi, and P. Tsaparas. Mining the inner structure of the Web graph. In *Proc. of the 8th Workshop on the Web and Databases (WebDB)*, pages 145–150, 2005.

[DRR06] O. Delpratt, N. Rahman, and R. Raman. Engineering the louds succinct tree representation. In *Proc. of the 5th International Workshop on Experimental Algorithms (WEA)*, LNCS 4007, pages 134–145, 2006.

[Eli74] Peter Elias. Efficient storage and retrieval by content and address of static files. *Journal of the ACM (JACM)*, 21:246–260, April 1974.

[Fan71] R. Fano. On the number of bits required to implement an associative memory. Memorandum 61, Computer Structures Group, Project MAC, MIT, Cambridge, Mass., 1971.

[FGNV09] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *ACM Journal of Experimental Algorithmics (JEA)*, 13:article 12, 2009. 30 pages.

[FM05] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM (JACM)*, 52:552–581, July 2005.

[FMMN07] P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 3(2):article 20, 24 pages, 2007.

[FMN09] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science (TCS)*, 410(51):5354–5364, 2009.

[FNP08] A. Fariña, G. Navarro, and J. Paramá. Word-based statistical compressors as natural language compression boosters. In *Proc. of the 18th Data Compression Conference (DCC)*, pages 162–171, 2008.

[FV07] P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. In *Proc. of the 18th Annual Symposium on Discrete Algorithms (SODA)*, pages 690–696, 2007.

[GGMN05] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Poster Proc. Volume of 4th Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005.

[GGV03] R. Grossi, A. Gupta, and J.S. Vitter. High-order entropy-compressed text indexes. In *Proc. of 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[GHSV06] A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed data structures: Dictionaries and data-aware measures. In *Proc. of the 2006 IEEE Data Compression Conference (DCC)*, 2006.

[GMR06] A. Golynski, I. Munro, and S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. of the 17th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[GN07] R. González and G. Navarro. Compressed text indexes with fast locate. In *Proc. of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 216–227, 2007.

[Gon09] S. González. Búsquedas en paralelo sobre texto comprimido autoindexado. Master's thesis, Department of Computer Science, University of Chile, October 2009.

[GRRR06] R. Geary, N. Rahman, R. Raman, and V. Raman. A simple optimal representation for balanced parentheses. *Theoretical Computer Science (TCS)*, 368(3):231–246, 2006.

[GV00] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. of the 32nd Symposium on Theory of Computing (STOC)*, pages 397–406, 2000.

[Hea78] H. Heaps. *Information Retrieval - Computational and Theoretical Aspects.* Academic Press, NY, 1978.

[Hor80] R. N. Horspool. Practical fast searching in strings. *Software: Practice and Experience (SPE)*, 10(6):501–506, 1980.

[Huf52] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers (IRE)*, 40(9):1098–1101, 1952.

[Jac89a] G. Jacobson. Space-efficient static trees and graphs. In *Proc. of the 30th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 549–554, 1989.

[Jac89b] G. Jacobson. *Succinct static data structures.* PhD thesis, Carnegie Mellon University, 1989.

[JSS07] J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In *Proc. of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 575–584, 2007.

[Kin70] I. P. King. An automatic reordering scheme for simultaneous equations derived from network systems. *International Journal for Numerical Methods in Engineering*, 2:523–533, 1970.

[KKR$^+$99] J. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. The Web as a graph: Measurements, models, and methods. In *Proc. of the 5th Annual International Conference on Computing and Combinatorics (COCOON)*, LNCS 1627, pages 1–17, 1999.

[KN11] S. Kreft and G. Navarro. Self-indexing based on LZ77. In *Proc. of the 22th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS, 2011. To appear.

[Kre10] S. Kreft. Self-index based on lz77. Master's thesis, University of Chile, 2010.

[LM00] J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proceedings of the IEEE*, 88(11):1722–1732, 2000.

[Mal76] K. Maly. Compressed tries. *Communications of the ACM (CACM)*, 19:409–415, July 1976.

[MC07] A. Moffat and S. Culppeper. Hybrid bitvector index compression. In *Proc. of the 12th Australasian Document Computing Symposium (ADCS)*, pages 25–31, 2007.

[MK95] A. Moffat and J. Katajainen. In-place calculation of minimum-redundancy codes. In S.G. Akl, F. Dehne, and J.-R. Sack, editors, *Proc. of the Workshop on Algorithms and Data Structures (WADS)*, LNCS 955, pages 393–402, 1995.

[MM93] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing (SICOMP)*, 22(5):935–948, 1993.

[MN05] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

[MN07] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science (TCS)*, 387(3):332–347, 2007.

[MN08] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms (TALG)*, 4(3):article 32, 2008. 38 pages.

[MNZBY98] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast searching on compressed text allowing errors. In *Proc. of the 21th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 298–306, 1998.

[MNZBY00] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.

[Mof89] A. Moffat. Word-based text compression. *Software: Practice and Experience (SPE)*, 19(2):185–198, 1989.

[Mof90] A. Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 38, 1990.

[Mor68] D. R. Morrison. PATRICIA—practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM (JACM)*, 15(4):514–534, 1968.

[MR01] I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing (SICOMP)*, 31(3):762–776, 2001.

[MT96] A. Moffat and A. Turpin. On the implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications*, 45:170–179, 1996.

[MT02] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer Academic Publishers, 2002.

[Mun96] I. Munro. Tables. In *Proc. of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, LNCS v. 1180, pages 37–42, 1996.

[MW94] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. In *Proc. of the Winter 1994 USENIX Technical Conference*, pages 23–32, 1994.

[Nav04] G. Navarro. Indexing text using the ziv-lempel trie. *Journal of Discrete Algorithms (JDA)*, 2(1):87–114, 2004.

[NM07] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys (CSUR)*, 39(1):article 2, 2007.

[NMN$^+$00] G. Navarro, E. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.

[NT00] G. Navarro and J. Tarhio. Boyer-moore string matching over ziv-lempel compressed text. In *Proc. of the 11st Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 1848, pages 166–180, 2000.

[OS07] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc.of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 60–70, 2007.

[Pag99] R. Pagh. Low redundancy in static dictionaries with o(1) worst case lookup time. In *Proc. of the 26th International Colloquium on Automata, Languages, and Programming (ICALP)*, number 1644 in LNCS, 1999.

[PBMW99] L. Page, S. Brin, R. Motwani, and T. Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[RGM03] S. Raghavan and H. Garcia-Molina. Representing Web graphs. In *Proc. of the 19th International Conference on Data Engineering (ICDE)*, page 405, 2003.

[RKT99] S. Rajagopalan R. Kumar, P. Raghavan and A. Tomkins. Trawling the web for emerging cyber-communities. *Computer Networks*, 31(11-16):1481–1493, 1999.

[RRR02] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. of the 13th Annual Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[RSWW01] K. Randall, R. Stata, R. Wickremesinghe, and J. Wiener. The LINK database: Fast access to graphs of the Web. Technical Report 175, Compaq Systems Research Center, Palo Alto, CA, 2001.

[RTT02] J. Rautio, J. Tanninen, and J. Tarhio. String matching with stopper encoding and code splitting. In *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 42–52, 2002.

[Sad03] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[Sam06] H. Samet. *Foundations of Multidimensional and Metric Data Structures.* Morgan Kaufmann Publishers Inc., 2006.

[SC07] T. Strohman and B. Croft. Efficient document retrieval in main memory. In *Proc. of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 175–182, 2007.

[Sil07] F. Silvestri. Sorting out the document identifier assignment problem. In *Proc. of the 29th European Conference on IR Research (ECIR*, pages 101–112, 2007.

[Sir10] J. Sirén. Sampled longest common prefix array. In *Proc. of the 21th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 6129, pages 227–237, 2010.

[SK64] E. S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM (CACM)*, 7(3):166–169, 1964.

[Sol07] D. Solomon. *Variable-length codes for data compression.* Springer-Verlag, 2007.

[ST07] P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proc. of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 71–83, 2007.

[STKA07] H. Saito, M. Toyoda, M. Kitsuregawa, and K. Aihara. A large-scale study of link spam detection by graph algorithms. In *Proc. of the 3rd International Workshop on Adversarial Information Retrieval on the Web (AIRWeb)*, page 48, 2007.

[Sto88] J. Storer. *Data Compression: Methods and Theory.* Addison Wesley, Rockville, Md., 1988.

[SW49] C. E. Shannon and W. Weaver. *A Mathematical Theory of Communication.* University of Illinois Press, Urbana, Illinois, 1949.

[SWYZ02] F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proc. of the 25th Annual International ACM SIGIR conference on Research and development in information retrieval (SIGIR)*, pages 222–229, 2002.

[SY01] T. Suel and J. Yuan. Compressing the graph structure of the Web. In *Proc. of the 11th Data Compression Conference (DCC)*, pages 213–222, 2001.

[TM97] A. Turpin and A. Moffat. Fast file search using text compression. In *Proc. of the 20th Australasian Computer Science Conference (ACSC)*, pages 1–8, 1997.

[Vig08] S. Vigna. Broadword implementation of rank/select queries. In *Proc. of the 5th Workshop on Experimental Algorithms (WEA)*, pages 154–168, 2008.

[Vit87] J. S. Vitter. Design and analysis of dynamic huffman codes. *Journal of the ACM (JACM)*, 34:825–845, October 1987.

[Vit01] J. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys (CSUR)*, 33(2):209–271, 2001. Version revised at 2007 from `http://www.cs.duke.edu/~jsv/Papers/Vit.IO_survey.pdf`.

[Wel84] T. A. Welch. A technique for high performance data compression. *Computer*, 17(6):8–20, June 1984.

[WMB99] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes.* Morgan Kaufmann Publishers, New York, 2nd edition, 1999.

[WNC87] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM (CACM)*, 30(6):520–540, 1987.

[WZ99] H. E. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal (COMPJ)*, 42(3):193–201, 1999.

[Zip49] G. K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley (Reading MA), 1949.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

[ZMR98] J. Zobel, A. Moffat, and K. Ramamohanarao. Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems (TODS)*, 23(4):453–490, 1998.