UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

ESTRUCTURAS COMPRIMIDAS PARA GRAFOS DE LA WEB

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS,
MENCIÓN COMPUTACIÓN

FRANCISCO JOSÉ CLAUDE FAUST

PROFESOR GUÍA:
GONZALO NAVARRO BADINO

MIEMBROS DE LA COMISIÓN:
BENJAMIN BUSTOS CÁRDENAS
MAURICIO MARÍN CAIHUÁN
JÉRÉMY BARBAY

Esta tesis ha recibido el apoyo de Yahoo! Research Latin America, del Núcleo Milenio Centro de Investigación de la Web Proyecto P04-067-F, Mideplan, Chile, y del Proyecto Fondecyt 1-050493, Conicyt, Chile.

SANTIAGO DE CHILE
AGOSTO 2008

Resumen

La estructura de la Web se puede modelar como un grafo, donde las páginas son los nodos y los hipervínculos las aristas. Estos *grafos Web* son ampliamente utilizados para diversas tareas de análisis de la Web, tales como el cálculo de *PageRank* o la detección de *spam* en la Web, entre otras. Una de las limitantes que se presentan al trabajar con estos grafos es su tamaño, por ejemplo, el 2005 se calculó que la Web pública y estática tenía 11.5 mil millones de nodos, y unas 15 aristas por nodo, lo que requiere más de 600 GB para su representación plana. De aquí surge la motivación de este trabajo, que consiste en la creación de estructuras de datos comprimidas para representar grafos de la Web.

Una estructura comprimida busca almacenar la mayor cantidad de datos en el menor espacio posible, ya sea en memoria principal o en disco, soportando las consultas de interés sin la necesidad de descomprimir la estructura en su totalidad. La principal ventaja de estas estructuras es que se puede evitar mantener la información en disco o se disminuye la cantidad de transferencias necesarias. Esto es de vital importancia dado que el disco puede llegar a ser un millón de veces más lento que la memoria principal.

Entre los resultados más importantes de este trabajo se presenta una estructura comprimida para grafos de la Web que mejora el estado del arte, ofreciendo el mejor compromiso espacio-tiempo conocido para recuperar listas de adyacencia. Además se muestra cómo extender esta estructura para soportar consultas más complejas, como vecinos reversos, manteniendo los requerimientos de espacio.

Como productos agregados se incluyen resultados experimentales y propuestas para el problema de *rank* y *select* sobre secuencias generales, incluyendo estructuras no implementadas antes. Los resultados derivan en mejoras inmediatas para índices comprimidos para texto, en los cuales se reduce el espacio utilizado por los mejores índices existentes, a veces incluso sin penalización en el tiempo de búsqueda. Además se presenta un algoritmo aproximado para comprimir utilizando el método Re-Pair cuando la memoria principal es limitada. También se obtienen resultados en estructuras comprimidas para relaciones binarias, presentándose una nueva propuesta que, además de utilizar espacio proporcional a la entropía de la relación binaria, permite dinamizar la estructura, vale decir, aceptar inserciones y borrados de pares en la relación.

Agradecimientos

Quiero comenzar agradeciendo a mi padres, Ingrid Faust y Francisco Claude, a mi hermana Carolina Claude y a mis abuelas Sonja Kühne y Nicole Bourdel. Todos ellos han sido un apoyo incondicional durante toda mi vida y no existen palabras adecuadas para agradecer todo lo que han hecho por mí, ni lo importantes que son en mi vida. Además me gustaría incluir a mi polola María Jesús Chau, quien ha comprendido mi pasión por lo que hago y me ha apoyado de corazón, junto con alegrar aún más estos últimos meses de trabajo.

También es importante incluir a los amigos, que siempre estuvieron ahí para compartir buenos momentos. Entre ellos, sin un orden particular, están: Roberto Konow, Francisco Gutiérrez, Ismael Vergara, Francisca Varela, María José Paredes, Francisco Uribe, Cristina Melo, Marcela Huerta, Gonzalo Dávila, Philippe Pavez, Cristián Serpell, Sebastián Kreft, Victor Ramiro, Julio Villane, Patricia Nuñez, Gustavo García, Mauricio Farah, Ignacio Fantini y muchos otros.

Otro grupo, muy importante para mí durante este trabajo, es el Grupo Miércoles de Algoritmos. Todos y cada uno de los integrantes de este grupo aportó a este trabajo con ideas y estimulantes discusiones, todo esto en un ambiente familiar y de amistad entre los miembros que hizo el trabajo aún más entretenido y ameno. Quiero agradecer especialmente a Rodrigo González, Diego Arroyuelo, Rodrigo Paredes, Hernán Arroyo, Rodrigo Cánovas, Susana Ladra, Ana Cerdeira, Daniel Valenzuela, Nora Reyes, Gilberto Gutiérrez y Felipe Sologuren.

No puedo dejar fuera a los miembros de mi comisión, Jérémy Barbay, Benjamin Bustos y Mauricio Marín, quienes revisaron el borrador de la tesis con una dedicación admirable y enriquecieron el contenido de ésta con comentarios muy acertados.

Por último me gustaría agradecer a mi profesor guía, Gonzalo Navarro, con quien tuve el honor y la suerte de trabajar en esta tesis. No sólo guió mi trabajo y estuvo siempre ahí para ayudarme durante todo el desarrollo de la tesis, sino que además estuvo presente como un gran amigo, siempre dispuesto a dar buenos consejos y organizar una que otra parrillada.

A todos los mencionados en esta página, y a todos ellos que me acompañaron durante este proceso, mis más sinceros agradecimientos.

Francisco José Claude Faust
13 de Agosto de 2008

University of Chile
Faculty of Physics and Mathematics
Graduate School

Compressed Data Structures for Web Graphs

by

Francisco Claude

Submitted to the University of Chile in fulfillment
of the thesis requirement to obtain the degree of
MSc. in Computer Science

Advisor : **Gonzalo Navarro**

Committee : Benjamin Bustos
: Mauricio Marín
: Jérémy Barbay

This work has been supported in part by Yahoo! Research Latin America, by Millennium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile, and by Fondecyt Project 1-050493, Conicyt, Chile.

Departament of Computer Science - University of Chile
Santiago - Chile
August 2008

Abstract

The Web structure can be modeled as a directed graph, where the pages are nodes and the links between pages correspond to edges. These *Web graphs* are widely used for analyzing the Web, for example, for the calculation of *Page-Rank* and for *spam detection*, among others. The size of these graphs makes their manipulation in main memory unfeasible. In 2005, the graph representing the public static Web was estimated to have 11.5 billion nodes, and 15 edges per node on average; this requires around 600 GB in its plain representation. Hence the motivation for this work, which consists in building compressed data structures for Web graphs.

A compressed data structure aims to represent data using little space, in main memory or on disk, while supporting queries without decompressing the whole data. The main advantage of these structures is that more data can be kept in main memory, or fewer disk transfers are required. This is of utmost importance, since an access to disk can be a million times slower than an access to main memory.

The most important contribution of this thesis is a new compressed data structure for Web graphs, which offers the best known space/time trade-off for retrieving the adjacency list of a node. We also show how to extend the structure in order to support more complex queries, such as reverse neighbors, within the same space.

As byproducts of this work we present experimental results for different data structures for the *rank* and *select* problem on general sequences, including structures never implemented before. These results have direct implications in *full-text self-indexes*, where we obtain space usages never achieved before, sometimes even without time penalty. We also present a new approximate version of the Re-Pair compression algorithm, which allows compressing sequences using little memory on top of the sequence itself. Finally, we include a new proposal for representing binary relations. This structure achieves space proportional to the zero-order entropy of the binary relation, and can be made dynamic in order to support insertion and deletion of new pairs in the relation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A compressed data structure, besides answering the queries supported by its classical (uncompressed) counterpart, uses little space for its representation. Nowadays this kind of structures is receiving much attention because of two reasons: (1) the enormous amounts of information digitally available, (2) the ever-growing speed gaps in the memory hierarchy. As an example of the former, the graph of the 2005 static indexable Web was estimated to contain more than 11.5 billion nodes [GS05] and more than 150 billion links. A plain adjacency list representation of this graph would need around 600 GB. As an example of (2), access time to main memory is about one million times faster than to disk. Similar phenomena (albeit less pronounced) arise at other levels of memory hierarchy. Although memory sizes have been growing fast, new applications have appeared with data management requirements that exceed the capacity of the faster memories. Distributed computation has been explored as a solution to those problems [BBYRNZ01, TGM93]. However, access to a remote memory involves a waiting time which is closer to that of a disk access than to a local one. Because of this scenario, it is attractive to design and use compressed data structures, even if they are several times slower than their classical counterpart. They will run much faster anyway if they manage to fit in a faster memory.

In this scenario, compressed data structures for graphs have gained interest in recent years, because a (directed) graph is a natural model of the Web structure. Several algorithms used by the main search engines to rank pages, discover communities, and so on, are run over those Web graphs. Needless to say, relevant Web graphs are huge and maintaining them in main memory is a challenge, especially if we wish to access them in compressed form, say for navigation purposes.

In this work we focus on building compressed data structures for Web graphs by treating them as text and using known text compression methods (sometimes adapted) to achieve better space while answering queries within competitive time.

1.1 Outline of the Thesis

Chapter 2 gives the basic concepts needed to read this thesis. We address data compression and coding, the related work in graph compression, rank/select capable data structures and compressed text indexes.

Chapter 3 proposes and studies different variants of rank/select data structures for binary sequences and texts over larger alphabets. We compare them on different kind of sequences and show the implications of these results for self-indexing.

Chapter 4 introduces a practical approximate version of Re-Pair and compares it with the optimal Re-Pair [LM00] and LZ78 [ZL78] on different types of sequences.

Chapter 5 presents a first approach for compressing Web graphs, which allows us to answer all the queries we have considered, but uses much space compared to other known structures and the time for answering queries is not competitive.

Chapter 6 introduces a simplification of the previous approach (Chapter 5) which achieves better time/space tradeoffs than the existing Web graph representations. The space needed by our structure is similar to the best known results while navigation is much faster.

Chapter 7 shows how to extend the representation presented in Chapter 6 in order to add backward navigation without using too much extra space.

Chapter 8 gives our conclusions and further lines of research that could be explored based on the results obtained in this work.

1.2 Contributions of the Thesis

Chapter 3 : We present a practical implementation of Raman, Raman and Rao's compressed rank/select data structure for binary sequences [RRR02]. We compare different variations of the wavelet tree [GGV03, FMMN07, MN07] and show how to implement them omitting pointers. We also present the first practical implementation of Golynski et al.'s rank/select data structure for strings over large alphabets [GMR06]. Using the results of this comparison, we implement a variant of the SSA compressed text-index [FGNV07], where we achieve the smallest self-index seen so far for counting. This work will appear in SPIRE 2008.

Chapter 4 : We introduce a new variation of Re-Pair [LM00] that works using a small amount of memory on top of the text and advantageously trades compression ratio for speed. We include a simple technique to allow local decompression in sequences compressed with Re-Pair and LZ78 [ZL78], based on the technique presented by González and Navarro [GN07], and present experimental results for this technique.

Chapter 5 : We present a graph representation that allows easy forward and backward navigation, and can represent undirected graphs without paying twice for every edge in the graph. We also show that this representation is suboptimal and that the representation shown in Chapter 6 achieves better time/space tradeoffs.

Chapter 6 : We demonstrate that an adjacency list based representation, combined with the technique of Chapter 4, improves by far the best known space/time tradeoff [BV04] and that it can be further compressed by combining the resulting structure with the techniques of Chapter 3. This work was published in SPIRE 2007 [CN07], and submitted to IEEE TKDE.

Chapter 7 : We explore an alternative way of compressing Web graphs, relating them to binary relations. We work mainly in a decomposition of a binary relation into two relations defined by the resulting structures of Chapter 6: the sequence compressed with Re-Pair and the dictionary. This approach achieves good space and supports reverse queries. We also present a new data structure for binary relations that achieves space proportional to the entropy of the binary relation, which can be dynamized and supports insertion and deletion of pairs to/from the relation.

Chapter 2

Related Work

2.1 Compression of Sequences

The goal of data compression is to store and access information using less space than its plain (uncompressed) representation.

A common measure of the compressibility of a sequence is the empirical entropy. For a text T of length n whose symbols are drawn from an alphabet Σ of size σ, the zero-order empirical entropy is defined as follows:

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

where n_c is the number of occurrences of c in T [1]. $H_0(T)$ represents the average number of bits needed to represent a symbol of T and is a lower bound of the compression that can be achieved without considering contexts, that is, encoding each symbol independently of the text surrounding it.

If we consider the context in which each symbol appears, we can achieve better compression ratios. For example, if we consider the context `th` in English, it is more likely to find an `a,e,i,o` or `u`, and it is very unlikely to find a `k`. The definition of empirical entropy can be extended to consider contexts as follows [Man01]:

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

[1] All logarithms are in base 2 unless stated otherwise.

where T^s is the sequence of symbols preceded by the context s in T. It can be proved that $H_k(T) \leq H_{k-1}(T) \leq \ldots \leq H_0(T) \leq \log(\sigma)$.

One way to achieve compression is to use so-called dictionary based methods, which focus on representing a sequence by factorization or reference to other areas of the sequence. Some examples of this technique are Lempel-Ziv [ZL77, ZL78] and Re-Pair [LM00]. It has been proved that those techniques achieve space proportional to $nH_k(T)$ plus some lower order terms [KM99, NR08].

Another way is to encode the source symbols with a variable-length encoder. For example Huffman [Huf52], which generates prefix-free codes for every symbol in the text, assigns shorter codes to the most probable symbols in order to represent the text using less space. This method achieves space proportional to $nH_0(T)$ plus some lower order terms.

Other methods transform the text prior to compressing it. An example is the Burrows-Wheeler Transform [BW94]. Using this approach one can achieve $nH_k(T) + o(n \log \sigma)$ bits of space using local zero-order compression.

2.2 Encoding

When representing data in compact or compressed form we usually need to represent symbols using binary variable-length prefix-free codes. In the case of Huffman, the codes depend on the symbol frequencies. There are other coding techniques for alphabets of unbounded size, which give shorter codes to the smaller symbols. In this work, we make use of Gamma codes, Delta codes and Rice codes [WMB99] tailored to positive numbers. Table 2.1 shows Unary, Gamma, Delta and Rice codes for the first 10 integers.

Unary Codes The unary representation is commonly used within other encodings; the idea is to encode the value n as $1^{n-1}0$. For example, for 5 the codification is 11110. The final zero allows to delimit the code (i.e., makes it prefix-free).

Gamma Codes The Gamma code of a given integer n is the concatenation of the length of its binary representation in unary, and the binary representation of n omitting the most significant bit. For example, for $5 = (101)_2$, its codification is 11001. The representation of a symbol n uses $2\lfloor \log n \rfloor + 1$ bits: $\lfloor \log n \rfloor + 1$ are used to represent the symbol length in unary and $\lfloor \log n \rfloor$ bits are used to represent the symbol without its most significant bit.

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 2.1: Different encodings for integers 1 to 10.

Delta Codes Delta codes are the natural extension of Gamma codes for larger symbols. They represent the binary length of the symbol using Gamma codes. This allows us to represent a symbol using $1 + 2\lfloor \log \log n \rfloor + \lfloor \log n \rfloor$ bits.

Rice Codes Rice codes are parameterized codes that receive two values, the symbol n and a parameter b. Then n is represented as $q = \lfloor (n-1)/2^b \rfloor$ in unary concatenated with $r = n - q2^b - 1$ in binary using b bits, for a total of $\left\lfloor \frac{n-1}{2^b} \right\rfloor + b$ bits.

2.3 Graph Compression

Let us consider graphs $G = (V, E)$, where V is the set of vertices and E is the set of edges. We call $n = |V|$ and $m = |E|$. Standard graph representations such as the incidence matrix and the adjacency list require $n(n-1)/2$ and $2m \log n$ bits, respectively, for undirected graphs. For directed graphs the numbers are n^2 and $m \log n$, respectively. We call the *neighbors* of a node $v \in V$ those $u \in V$ such that $(v, u) \in E$.

The oldest work on graph compression focuses on undirected unlabeled graphs. The first result we know of [Tur84] shows that planar graphs can be compressed into $O(n)$ bits. The constant factor was later improved [KW95], and finally a technique yielding the optimal constant factor was devised [HKL00]. Results on planar graphs can be generalized to graphs with constant *genus* [Lu02]. More generally, a graph with genus g can be compressed into $O(g + n)$ bits [DL98].

Some classes of planar graphs have also received special attention, for example trees, triangulated meshes, triconnected planar graphs, and others [IR82, KW95, HKL99, Ros99]. For dense graphs, it is shown that little can be done to improve the space required by the adjacency matrix [Nao90].

The above techniques consider just the compression of the graph, not its access in compressed form. The first compressed data structure for graphs we know of [Jac89] requires $O(gn)$ bits of space for a g-page graph. A page is a subgraph whose nodes can be written in a linear layout so that its edges do not cross (the ordering of nodes in those linear layouts must be consistent across pages). Edges of a page hence form a nested structure that can be represented as a balanced sequence of parentheses. The operations are supported using succinct data structures that permit navigating a sequence of balanced parentheses. The neighbors of a node can be retrieved in $O(\log n)$ time each (plus an extra $O(g)$ complexity for the whole query). The $O(\log n)$ time was later improved to constant by using improved parentheses representations [MR97], and also the constant term of the space complexity was improved [CGH$^+$98]. The representation also permits finding the degree (number of neighbors) of a node, as well as testing whether two nodes are connected or not, in $O(g)$ time.

All those techniques based on number of pages are unlikely to scale well to more general graphs, in particular to Web graphs. A more powerful concept that applies to this type of graph is that of graph *separators*. Although the separator concept has been used a few times [DL98, HKL00, CPMF04] (yet not supporting access to the compressed graph), the most striking results are achieved in recent work [BBK03, Bla06]. Their idea is to find graph components that can be disconnected from the rest by removing a small number of edges. Then, the nodes within each component can be renumbered to achieve smaller node identifiers, and only a few external edges must be represented.

They [Bla06] apply the separator technique to design a compressed data structure that gives constant access time per delivered neighbor. They carefully implement their techniques and experiment on several graphs. In particular, on a graph of 1 million (1M) nodes and 5M edges from the Google programming contest[2], their data structures require 13–16 bits per edge (*bpe*), and work faster than a plain uncompressed representation using arrays for the adjacency lists. It is not clear how these results would scale to larger graphs, as much of their improvement relies on smart caching, and this effect should vanish with real Web graphs, which have no chance of fitting a significant portion in today's caches, even if compressed.

[2] `www.google.com/programming-contest`, not anymore available.

There is also some work specifically aimed at compression of Web graphs [BKM+00, AM01, SY01, BV04]. In this graph, the (labeled) nodes are Web pages and the (directed) edges are the hyperlinks. Several properties of Web graphs have been identified and exploited to achieve compression:

Skewed distribution: The in- and out-degrees of the nodes distribute according to a power law, that is, the probability that a page has i links is $1/i^\theta$ for some parameter $\theta > 0$. Several experiments give rather consistent values of $\theta = 2.1$ for incoming and $\theta = 2.72$ for outgoing links [ACL00, BKM+00].

Locality of reference: Most of the links from a site point within the site. This motivates the use of lexicographical URL order to list the pages, so that outgoing links go to nodes whose position is close to that of the current node [BBH+98]. Gap encoding techniques are then used to encode the differences among consecutive target node positions.

Similarity of adjacency lists: Nodes tend to share many outgoing links with some other nodes [KRRT99, BV04]. This permits compressing them by a reference to the similar list plus a list of edits.

Suel and Yuan [SY01] partition the adjacency lists considering the popularity of the nodes, and use different coding methods for each partition. A more hierarchical view of the nodes is exploited by Raghavan and Garcia-Molina [RGM03]. Different authors [AM01, RSWW01] take explicit advantage of the similarity property. A page with similar outgoing links is identified with some heuristic, and then the current page is expressed as a reference to the similar page plus some edit information to encode the deletions and insertions needed to obtain the current page from the referenced one. Finally, probably the best current result is from Boldi and Vigna [BV04], who build on previous work [AM01, RSWW01] and further engineer the compression to exploit the properties above.

Experimental figures are not always easy to compare, but they give a reasonable idea of the practical performances. Over a graph with 115M nodes and 1.47 billion (1.47G) edges from the Internet Archive, Suel and Yuan [SY01] require 13.92 bpe (plus around 50 bits per node, *bpn*). Randall et al. [RSWW01], over a graph of 61M nodes and 1G edges, achieve 5.07 bpe for the graph. Adler and Mitzenmacher [AM01] achieve 8.3 bpe (no information on bpn) over TREC-8 Web track graphs (WT2g set), yet they cannot access the graph in compressed form. Broder at el. [BKM+00] require 80 bits per node plus 27.2 bpe (and can answer reverse neighbor queries as well).

By far the best figures are from Boldi and Vigna [BV04]. For example, they achieve space close to 3 bpe to compress a graph of 118M nodes and 1G link from WebBase[3]. This space, however, is not sufficient to access the graph in compressed form. They carried out an experiment including the extra information required for navigation on a graph of 18.5M nodes and 292M links, and their method needs 6.7 bpe to achieve access times below the microsecond. Those access times are of the same order of magnitude than other representations [SY01, RGM03, RSWW01]. For example, the latter reports times around 300 nanoseconds per delivered edge.

A recent proposal [Nav07] advocates regarding the adjacency list representation as a text sequence and using compressed text indexing techniques [NM07], so that neighbors can be obtained via text decompression and reverse neighbors via text searching. The concept and the results are interesting but not yet sufficiently competitive with those of Boldi and Vigna.

2.4 Phrase-Based Compression

2.4.1 Re-Pair

Re-Pair [LM00] is a phrase-based compressor that permits fast and local decompression. It consists 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. More precisely, Re-Pair over a sequence T works as follows:

1. It identifies the most frequent pair ab in T
2. It adds the rule $s \rightarrow ab$ to a dictionary R, where s is a new symbol not appearing in T.
3. It replaces every occurrence of ab in T by s.[4]
4. It iterates until every pair in T appears once.

Let us call C the resulting text (i.e., T after all the replacements). It is easy to expand any symbol s from C in time linear on the expanded data (i.e., optimal): We expand s using rule $s \rightarrow s's''$ in R, and continue recursively with s' and s'', until we obtain the original symbols of T.

[3] `www-diglib.stanford.edu/~testbed/doc2/WebBase/`

[4] As far as possible, e.g., one cannot replace both occurrences of aa in aaa.

Despite its quadratic appearance, Re-Pair can be implemented in linear time [LM00]. However, this requires several data structures to track the pairs that must be replaced. This is usually problematic when applying it to large sequences, as witnessed when using it for natural language text compression [Wan03]. Indeed, it was also a problem when using it over suffix arrays [GN07], where an approximate algorithm (that is, it does not always choose the most frequent pair to replace) performs much better. The approximate algorithm runs very fast, with limited extra memory, and loses very little compression. Unfortunately, it only applies to suffix arrays.

In addition, the method works well in secondary memory. If we store C on disk and the dictionary in main memory, the decompression process is I/O-optimal, since we only need to expand a contiguous piece of the sequence.

2.4.1.1 Dictionary Compression

As each new rule added to R costs two integers of space, replacing pairs that appear twice does not involve any gain unless R is compressed. In the original proposal [LM00], a very space-effective dictionary compression method is presented. However, it requires R to be fully decompressed before using it. In this work, we are interested in being able to *operate* the graphs in little space. Thus, we favor a second technique to compress R [GN07], which reduces the space it takes to about a half and can operate in compressed form. We use this dictionary representation in our experiments, and explain it here.

The main idea is to represent the set of rules as a set of binary trees. Every tree is represented by the sequence of symbols (S) and a bitmap (BRR) that defines the shape of the tree. For example, the set of rules $c \rightarrow ab$, $d \rightarrow cb$ and $e \rightarrow ac$ can be represented by the tree shown in Figure 2.1.

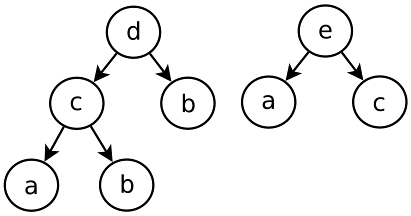


Figure 2.1: Example Re-Pair rules representation.

In BRR, internal nodes are represented by ones and leaves by zeroes. The example shown in Figure 2.1 would result in `11000100` and the sequence S is `abba2`, where the last ‘2’ represents the non-terminal ‘c’, whose tree is at position 2 in BRR. In general, the dictionary is a set of trees, which are concatenated. Internal nodes are identified with the position of their 1 in the bitmap, for example $d = 1$, $c = 2$ and $e = 6$. Given the starting position of a non-terminal symbol (in the bitmap), it is easy to expand it: We have to traverse the tree until the number of 0s exceeds the number of 1s. To map the i-th 0 in the bitmap to a symbol, we have to access $S[rank_{BRR}(0, i)]$ (see Section 2.5).

The description of symbol c is included inside that of symbol d. This saves one integer to represent c, and it can be done only once per symbol. The rest of the occurrences of c are not expanded but appear in S. Those non-terminals have to be recursively expanded. Figure 2.2 shows the expansion process.

```
decompress($s$)
    If $s \leq \sigma$ Then output $s$
    Else
        $p \leftarrow s - \sigma$
        $r \leftarrow 1$
        $i \leftarrow rank_{BRR}(0, p)$
        While $r > 0$ Do
            If $BRR[p] = 1$ Then $r \leftarrow r + 1$
            Else
                decompress($S[i]$)
                $r \leftarrow r - 1$
                $i \leftarrow i + 1$
```

Figure 2.2: Re-Pair decompression. We assume terminals are in $[1, \sigma]$ and nonterminals are positions in the bitmap shifted by σ.

2.4.2 Lempel-Ziv

The Lempel-Ziv compression family [ZL77, ZL78] achieves compression by replacing repeated sequences found in the text by a pointer to a previous occurrence thereof. In particular, the LZ78 variant [ZL78] stands as a plausible candidate for graph compression.

LZ78 compresses the text by dividing it into *phrases*. Each phrase is built as the concatenation of the longest previous phrase that matches the prefix of the text yet to be compressed and an extra character which makes this phrase different from all the previous ones. The algorithm is as follows:

1. It starts with a *dictionary* S of known phrases, containing initially the empty string.

2. It finds the longest prefix $T_{i,j}$ of the text $T_{i,n}$ yet to be processed, which matches an existing phrase. Let p be that phrase number.

3. It adds a new phrase to S, with a fresh identifier, and content (p, T_{j+1}).

4. It returns to step 2, to process the rest of the text $T_{j+2,n}$.

In order to carry out efficiently step 2, S is organized as a trie data structure. The output of the compressor is just the sequence of pairs (p, T_{j+1}). The phrase identifier is implicitly given by the position of the pair in the sequence.

The content of any phrase in the compressed text can be obtained backwards in linear time. Let p_0 the phrase we wish to expand. We read the p_0-th pair in the compressed sequence and get (p_1, c_0). Then c_0 is the last character of the phrase. Now we read the p_1-th pair and get (p_2, c_1), thus c_1 precedes c_0. We continue until reaching $p_i = 0$, which denotes the empty phrase. In i constant-time steps we obtained the content $c_{i-1}c_{i-2} \ldots c_1c_0$.

Just as for Re-Pair, this extraction can be made I/O-optimal if we limit the creation of phrases to what can be maintained in main memory. A simple way to achieve this is as follows: After the main memory is full, the process continues identically but no new phrases are inserted into S (hence not all the phrase contents will be different).

2.5 Rank and Select on Sequences

Probably the most basic tool, used in virtually all compressed data structures, is the sequence of symbols supporting *rank*, *select* and *access*. *Rank(a,i)* counts the number of occurrences of character a until position i (included). *Select(a,i)* finds the position of the i-th occurrence of a in the sequence. *Access(i)* returns the symbol at position i in the sequence. The most basic case is when the sequence is drawn from a binary alphabet. Theoretically and practically appealing solutions have been proposed for this case, achieving space close to the zero-order entropy of the sequence and good time performance.

Variant	Size	*Rank*	*Select*
`esp`	$nH_0(B) + o(n)$	$O(1)$	$O(1)$
`recrank`	$1.44m \log \frac{n}{m} + m + o(n)$	$O\left(\log \frac{n}{m}\right)$	$O\left(\log \frac{n}{m}\right)$
`vcode`	$m \log(n/\log^2 n) + o(n)$	$O(\log^2 n)$	$O(\log n)$
`sdarray`	$m \log \frac{n}{m} + 2m + o(n)$	$O\left(\log \frac{n}{m} + \frac{\log^4 m}{\log n}\right)$	$O\left(\frac{\log^4 m}{\log n}\right)$
`darray`	$n + o(n)$	$O(1)$	$O\left(\frac{\log^4 m}{\log n}\right)$

Table 2.2: Space in bits and query time achieved by the data structures proposed by Okanohara and Sadakane.

2.5.1 Binary Sequences

Many solutions have been proposed for the case of binary sequences. Consider a bitmap $B[1, n]$ with m ones. The first compact solution to this problem is capable of answering the queries in constant time and uses $n+o(n)$ bits [Cla96] (i.e., B itself plus $o(n)$ extra space); the solution is straightforward to implement [GGMN05]. This was later improved by Raman, Raman and Rao (RRR) [RRR02] achieving $nH_0(B) + o(n)$ bits while answering the queries in constant time, but the technique is not anymore simple to implement. Several practical alternatives achieving very close results have been proposed by Okanohara and Sadakane [OS07], tailored to the case of small m: `esp`, `recrank`, `vcode`, `sdarray`, and `darray`. Table 2.2 shows the time and space complexities achieved by these variants. Most of them are very good for *select* queries, yet *rank* queries are slower. The variant `esp` is indeed a practical implementation of RRR structure that saves space by replacing some pointers by estimations based on entropy.

In this work, we implement the RRR data structure [RRR02]. It divides the sequence into blocks of length $u = \frac{\log n}{2}$ and every block is represented as a tuple (c_i, o_i). The first component, c_i, represents the *class* of the block, which corresponds to its number of 1s. The second, o_i, represents the *offset* of that block inside a list of all the possible blocks in class c_i. Three tables are defined:

- Table E: stores every possible combination of u bits, sorted by class, and by offset within each class. It also stores all answers for *rank* at every position of each combination.
- Table R: corresponds to the concatenation of all the c_i's, using $\lceil \log(u + 1) \rceil$ bits per field.

- Table S: stores the concatenation of the o_i's using $\left\lceil \log \binom{u}{c_i} \right\rceil$ bits per field.

This structure also needs two partial sum structures [RRR01], one for R and the other for the length of the o_i's in S, $posS$. For answering *rank* until position i, we first compute $sum(R, \lfloor i/u \rfloor) = \sum_{j=0}^{\lfloor i/u \rfloor} R_j$, the number of 1s before the beginning of i's block, and then *rank* inside the block until position i using table E. For this, we need to find o_i: using $sum(posS, \lfloor i/u \rfloor)$ we determine the starting position of o_i in S, and with c_i and u we know how many bits we need to read. For *select* queries, they store the same extra information as Clark [Cla96], but no practical implementation for this extra structure has been shown. *Access* can be answered with two *rank*s, $access(i) = rank(1, i) - rank(1, i-1)$.

2.5.2 Arbitrary Sequences

Rank, *select* and *access* operations can be extended to arbitrary sequences drawn from an alphabet Σ of size σ. The two most prominent data structures that solve this problem are reviewed next.

Wavelet Trees. A wavelet tree [GGV03, FMMN07, NM07] is a perfectly balanced tree that stores a bitmap of length n in the root; every position in the bitmap is either 0 or 1 depending on the value of the most significant bit of the symbol in that position in the sequence.[5] A symbol with a 0 goes to the left subtree and a symbol with a 1 goes to the right subtree. This decomposition continues recursively with the next highest bit, and so on. The tree has σ leaves and requires $n\lceil \log \sigma \rceil$ bits, n bits per level. Every bitmap in the tree must be capable of answering *access*, *rank* and *select* queries.

The *access* query for position i can be answered by following the path described for position i. At the root, if the bitmap at position i has a 0/1, we descend to the left/right child, switching to the bitmap position $rank(0/1, i)$ in the left/right subtree. This continues recursively until reaching the last level, when we finish forming the binary representation of the symbol.

The *rank* query for symbol a until position i can be answered in a similar way as *access*, the difference being that instead of considering the bit at position i in the first level, we consider the most significant bit of a; for the second level we consider

[5] In general wavelet trees are described as dividing alphabet segments into halves. The description we give here, based on the binary decomposition of alphabet symbols, is more convenient for the solutions shown in the next chapter.

the second highest bit, and so on. We update the position for the next subtree with $rank(b, i)$, where b is the bit of a considered at this level. At the leaves, the final bitmap position corresponds to the answer to $rank(a, i)$ in S.

The *select* query does a similar process as *rank*, but upwards. To select the i-th occurrence of character a, we start at the leaf where a is represented and do *select*(b, i) where, as before, b is the bit of a corresponding to this level. Using the position obtained by the binary *select* query we move to the parent, querying for this new position. At the root, the position is the final result.

The cost of the operations is $O(\log \sigma)$ assuming constant-time *rank*, *select* and *access* over bitmaps. If we use a multiary wavelet tree, using general sequences for the levels, the time drops to $O(1 + \frac{\log \sigma}{\log \log n})$ [FMMN07]. However, no practical implementation of this variant has succeeded up to now.

A practical variant to achieve $n(H_0(S) + 1)$ bits of space is to give the wavelet tree the same shape than the Huffman tree of S [GGV03, NM07]. This saves space and even time on average.

Golynski et al. Golynski et al. [GMR06] proposed a data structure capable of answering *rank*, *select* and *access* in time $O(\log \log \sigma)$ using $n \log \sigma + n\, o(\log \sigma)$ bits of space. The main idea is to reduce the problem over one sequence of length n and alphabet size σ to n/σ sequences of length σ.

Consider a binary matrix M with n columns and σ rows. The value of $M[i, j]$ is 1 if the j-th symbol of the text is i and 0 otherwise. Let A be obtained by writing M in row-major order. We can answer *rank* and *select* very easily: *Rank* for symbol a until position i is $rank_A(1, (a-1)n + i) - rank_A(1, (a-1)n)$. *Select* for the i-th occurrence of symbol a is $select_A(1, i + rank_A(1, (a-1)n)) - (a-1)n$. The space required by A is too high, $n\sigma$ bits, so they divide A into pieces of length σ and write the cardinality (number of 1s) of every piece in unary in a new bitmap B. For example if $A = 001011010111101$ and $\sigma = 3$, the resulting B is 01011010111011. Using this new bitmap we can answer *rank* only for positions that are multiples of σ and we can only determine in which block is the i-th position for *select*. In exchange, B uses $2n + o(n)$ bits instead of $n\sigma$. In order to complete the structure, we must be able to answer *rank*, *select* and *access* inside the blocks formed by B. This is solved by using a structure they call a *chunk*.

Every *chunk* stores σ symbols of the text using a bitmap X and a permutation π. X stores the cardinality of every symbol of the alphabet in the *chunk* using the same encoding as B. π stores the permutation obtained by stably sorting the sequence represented by the *chunk*, and uses a data structure that allows constant-time computation of π^{-1} [MRRR03]. This requires $2\sigma + o(\sigma)$ bits for X and $\sigma \log \sigma + o(\sigma \log \sigma)$ for π. Summing over the n/σ *chunk*s we get $2n + n \log \sigma + n\, o(\log \sigma)$. For example, if the sequence is 231221 then $X = 011011101$ and $\pi = [3, 6, 1, 4, 5, 2]$.

Every query is divided into two subqueries, the first one over B and the second over the corresponding chunk. Using B and assuming that σ divides n, we can determine *rank* for a symbol a until position $\lfloor i/\sigma \rfloor \sigma$ by computing $rank_B(1, select_B(0, (a+1)\lfloor n/\sigma \rfloor + \lfloor i/\sigma \rfloor)) - rank_B(1, select_B(0, (a+1)\lfloor n/\sigma \rfloor))$. For *select*, we can determine the chunk where the i-th occurrence of a appears by computing $rank_B(0, select_B(1, rank_B(1, select_B(0, (a-1)\lfloor n/\sigma \rfloor)) + i))$.

Inside the *chunk*, a *select* query can be answered by computing $\pi(rank_X(1, select_X(0, a) + i))$. The *rank* queries can be answered by doing a binary search over $\pi[rank_X(1, select_X(0, a)) \ldots rank_X(1, select_X(0, a+1))]$. Using a Y-Fast Trie [Wil83] to speed up this process, Golynski et al. achieve $O(\log \log \sigma)$ instead of $O(\log \sigma)$ time. The *access* query for retrieving the symbol at position i is calculated in constant time as $rank_X(0, select_X(1, \pi^{-1}(i)))$.

We note that the $n\, o(\log \sigma)$ extra term does not vanish asymptotically with n but with σ. This suggests, as we verify experimentally later, that the structure performs well only on large alphabets.

2.6 Compressed Full-Text Indexes

A *full-text index* is a data structure that indexes a text T of length n, drawn from an alphabet Σ of size σ. The index supports the following operations:

- *count(pattern p)*: counts the number of occurrences of p in T.
- *locate(pattern p)*: reports the positions where p appears in T.
- *extract(position i, position j)*: extracts the substring $T[i \ldots j]$.

Classical indexes, like suffix arrays [MM93], consist of a large data structure built on top of the text, which answers *count* and *locate*. Extracting strings is trivial since T is stored as well.

A *compressed full-text self-index* is a full-text index that in addition replaces the text and takes space proportional to the compressed text size. Since they do not

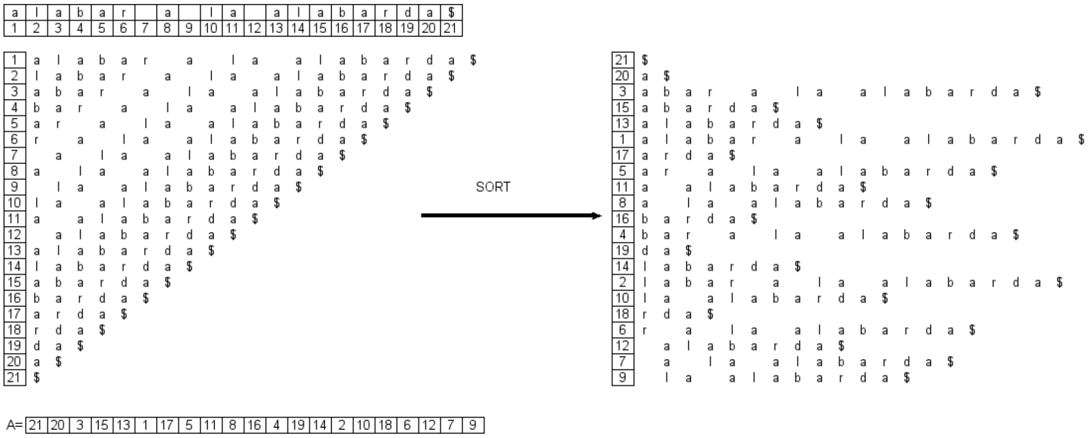


Figure 2.3: Example of a suffix array. It assumes that ‘ ’ is greater than any other symbol.

store the text anymore, the *extract* operation can be far from obvious now. In this work, in particular, we consider Sadakane’s Compressed Suffix Array (CSA) [Sad03] and the FM-Index [FM00].

2.6.1 Suffix Arrays

A *suffix array* [MM93] is a data structure built on top of the text that speeds up the *count* and *locate* operations. The suffix array stores the positions of all the suffixes of the text, sorted alphabetically. This is not a compressed full-text index, but the idea behind the suffix array is used as a base for many compressed full-text indexes.

Figure 2.3 shows the suffix array for the text ‘`alabar a la alabarda`’. We add an extra symbol $ at the end of the text, which is alphabetically smaller than all the others.

For *counting* and *locating* we perform a binary search over the suffixes, finding the left and right boundaries where the pattern appears. The number of elements between these two positions is the answer to *count*, and the positions listed in the suffix array in that range are the answer to *locate*, since every substring in the text is the prefix of a suffix. For a pattern of length m, *count* takes time $O(m \log n)$ and *locate* takes $O(m \log n + occ)$, where occ is the number of occurrences of the pattern.

$i =$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
$A =$	21	20	3	15	13	1	17	5	11	8	16	4	19	14	2	10	18	6	12	7	9
$\Psi =$	15	3	12	8	18	20	10	21	16	9	19	5	14	4	11	7	17	13	2	1	6

Table 2.3: Example of Ψ

2.6.2 Sadakane's Compressed Suffix Array (CSA)

Sadakane's CSA [Sad03] is based on the function Ψ, which is computed using the suffix array.

Definition 1 (Ψ) *Let $A[1 \ldots n]$ be the suffix array of a text T. Then $\Psi(i)$ is defined as the position i' in the suffix array where $A[i'] = (A[i] \bmod n) + 1$, which is $A[i]+1$, except for the case $A[i] = n$, where $A[i'] = 1$.*

In simple words, $\Psi(i)$ tells the position where the "next" text suffix is located in the suffix array. Table 2.3 shows Ψ for the suffix array given in Figure 2.3.

The CSA also uses function $Occ(s)$, defined for every symbol $s \in \Sigma$, which counts the number of occurrences of all the symbols $s' < s$ in the text.

Using Ψ, Occ, and regular sampling of the suffix array and the text, we can build an index that replaces the text and supports the three operations described above. Before explaining how to answer the queries we highlight some important properties:

- Ψ is a permutation from $[1 \ldots n]$. In the ranges corresponding to one symbol in the suffix array, Ψ is increasing [GV00].
- $Occ(s)$ corresponds to the first position where the suffixes starting with s appear in the suffix array.

To count the number of occurrences of a pattern $p = p_1 p_2 \ldots p_m$ in a text indexed using the CSA, we perform a backward search. First, using Occ we determine the range of the suffix array where the suffixes starting with p_m are. Then, we search the range where the suffixes start with p_{m-1}, for the subrange where the values of Ψ are within the prior range. This iterates until reaching p_1, where the length of the range corresponds to the number of occurrences of the pattern in the text. For locating, we retrieve the values of the suffix array within the range obtained during counting, and those are the positions where the pattern appears. For retrieving the values of the suffix array, we must store a sample of it. If we have to apply Ψ k times on the original position to reach a sampled value a_i, then the answer is $a_i - k$. Figure 2.4 shows the algorithm for counting in the CSA. For locating, we have to retrieve the values in the range determined by the count operation.

```
CSA-Count($p_1 p_2 \ldots p_m$)
    $i \leftarrow m$
    $sp \leftarrow 1$
    $ep \leftarrow n$
    While $sp \leq ep$ and $i \geq 1$ Do
        $c \leftarrow p_i$
        $sp' \leftarrow min\{j \in [Occ(c)+1, Occ(c+1)], \Psi(j) \in [sp, ep]\}$
        $ep \leftarrow max\{j \in [Occ(c)+1, Occ(c+1)], \Psi(j) \in [sp, ep]\}$
        $sp \leftarrow sp'$
        $i \leftarrow i - 1$
    If $ep < sp$ Then
        return 0
    Else
        return $ep - sp + 1$
```

Figure 2.4: Count - CSA

This index requires $\frac{1}{\epsilon} n H_0(T) + O(n \log\log\sigma) + \sigma\log\sigma$ bits, for any $0 < \epsilon \leq 1$. Mäkinen and Navarro later improved this space to $nH_k(T) + O(n\log\log\sigma)$ for $k \leq \alpha\log_\sigma n$ and constant $0 < \alpha < 1$ [MN07]. The time for counting is $O(m \log n)$, the time for locating is $O(\log n)$ per occurrence, and the time to display ℓ symbols of the text is $O(\ell + \log n)$.

2.6.3 The FM-Index

The FM-Index family is based on the Burrows-Wheeler transform (BWT) [BW94]. We first explain the transformation, how to recover the text from the transformation, and how to search for a pattern inside the transformed text. Then we describe some full-text self-indexes based on this transformation and how they represent the BWT.

2.6.3.1 The Burrows-Wheeler Transform (BWT)

For a text T of length n, imagine a matrix of dimensions $n \times n$ of all the cyclic shifts of T. If we sort this matrix, then the BWT (T^{bwt}) is the last column of that matrix. This is equivalent to collecting the character preceding every suffix of the suffix array A of T, hence BWT can be identified with A. Assuming that the last symbol of T is lexicographically smaller than all the other symbols in T, and that it only appears in that position, it is possible to reverse the transformation: $T[n-1]$ is located at $T^{bwt}[1]$, since the first element in the sorted matrix starts with $T[n]$. A mapping function LF allows us to navigate T backwards, $T[n-2] = T^{bwt}[LF(1)]$. For a position k, $T[n-k] = T^{bwt}[LF^{k-1}(1)]$.

Definition 2 *The LF-mapping is defined as $LF(i) = Occ[c] + rank_{T^{bwt}}(c, i)$, where $c = T^{bwt}[i]$ and $Occ[c]$ is the number of symbols lexicographically smaller than c in T.*

Note that LF is the inverse function of Ψ: LF corresponds to the position in A of the suffix pointing to $T[A[i]-1]$. Using the BWT we can perform backward search in a similar way as for the CSA index. Figure 2.5 shows the algorithm for counting the occurrences of a pattern P.

```
FM-Count($p_1 p_2 \dots p_m$)
    $i \leftarrow m$
    $sp \leftarrow 1$
    $ep \leftarrow n$
    While $sp \leq ep$ and $i \geq 1$ Do
        $c \leftarrow p_i$
        $sp \leftarrow C[c] + rank_{T^{bwt}}(c, sp-1) + 1$
        $ep \leftarrow C[c] + rank_{T^{bwt}}(c, ep)$
        $i \leftarrow i - 1$
    If $ep < sp$ Then
        return 0
    Else
        return $ep - sp + 1$
```

Figure 2.5: Counting on FM-Indexes

Since the suffix array is not explicitly stored, we must use a regular text sampling to retrieve the positions covered by the range resulting from the *counting*.

Given a position i in the BWT, we can traverse the text backwards by jumping across the BWT using the LF function, thus if after applying the LF function k times we get a sampled value $A[j]$, then the value for the original position is $A[j]+k$. A suitable representation of the BWT, which supports *rank* queries, is a wavelet tree (see Section 2.5.2).

The most prominent indexes based on the BWT are [NM07, FGNV07]:

SSA [MN05, FMMN07] Uses a Huffman-shaped wavelet tree on the BWT and plain bitmaps to approach zero-order entropy space ($n(H_0(T)+1)(1+o(1))$ bits).

AFFM-Index [FMMN07] Splits the BWT into segments and represents each with a Huffman-shaped wavelet tree, to approach high-order entropy space. The space required is $nH_k(T)+o(n\log\sigma)$ bits for $k \leq \alpha\log_\sigma n$, $0<\alpha<1$.

RLFM-Index [MN05] Builds a wavelet tree over the run heads of the BWT, requiring $O(nH_k(T)\log\sigma)$ bits for $k \leq \alpha\log_\sigma n$, $0<\alpha<1$.

Each of them supports counting in $O(m\log\sigma)$ time and locating in $O(\log^{1+\epsilon} n)$ per occurrence, for any constant $\epsilon > 0$.

Chapter 3

Rank, Select and Access on Sequences

In this chapter we propose and study practical implementations of sequences with rank and select capabilities. Our first contribution is a compressed representation of binary sequences based on Raman, Raman, and Rao's (RRR) [RRR02] theoretical proposal. We combine a faithful implementation of the theory with some common sense decisions. The result is compared, on uniformly distributed bitmaps, with a number of very well-engineered implementations for compressible binary sequences [OS07], and found to be competitive when the sequence is not too compressible, that is, when the proportion of 1s is over 10%.

Still this result does not serve to illustrate the local compressibility property of RRR data structure, that is, that it adapts well to local variations in the sequence. Mäkinen and Navarro [MN07] showed that the theoretical properties of RRR structure makes it an excellent alternative for full-text indexing: Combining it with the BWT (see Section 2.6.3) immediately yields a high-order compressed self-index, without all the extra sophistications previously used [NM07]. In this chapter we show experimentally that the proposed combination does work well in practice, achieving (sometimes significantly) better space than any other existing self-index, with moderate or no slowdown. The other compressed bitmap representations do not achieve this result: the bitmaps are globally balanced, but they exhibit long runs of 0s or 1s that only the RRR technique exploits so efficiently.

We then turn our attention to representing sequences over larger alphabets. Huffman-shaped wavelet trees have been used to approach zero-order compression of sequences [GGV03, FGNV07]. This requires $O(\sigma \log n)$ bits for the symbol table and the tree pointers, where n is the sequence length. On large alphabets, this factor can be prohibitive in space and ruin the compression ratios. We propose an alternative representation that uses no (or just $\log \sigma$) pointers, and concatenates all the bitmaps of the wavelet tree by levels. As far as we know, no previous solution to *select* over this representation existed. Combined with our compressed bitmap representation, the result is an extremely useful tool to represent a sequence up to its zero-order entropy, disregarding any problem related to alphabet size. We illustrate this point by improving an existing result on graph compression (see Chapter 6), in a case where no other considered technique succeeds.

Finally, we present the (as far as we know) first implementation of Golynski et al.'s data structure for sequences [GMR06], again combining faithful implementation of the theory with common sense. The result is a representation that does not compress the sequence, yet it answers queries very fast without using too much extra space. In particular, its performance over a sequence of word identifiers provides a representation that uses about 70% of the original space of the text (in character form) and gives the same functionality of an inverted index. In this sense, it might become an interesting alternative to recent wavelet-tree-based proposals for representing text collections [BFLN08], and to inverted indexes in general.

3.1 Practical Implementations

In the next three subsections we describe our practical implementations of RRR [RRR02] structure for binary strings, of wavelet trees [GGV03, FMMN07, NM07], and of Golynski et al.'s [GMR06] data structure. (See Section 2.5.)

3.1.1 Raman, Raman and Rao's Structure

We store the table E, which only depends on u (recall Section 2.5.1), and we fix $u = 15$ so that encoding the c_i's requires at most 4 bits each. We store table E using 16-bit integers for the bitstring contents, and for the pointers to the beginning of each class in E. The answers to $rank$ are not stored but computed on the fly from the bitstrings, so E uses just 64 KB. Table R is represented by a compact array using 4 bits per field, achieving fast extraction. Table S stores each offset using $\left\lceil \log \binom{u}{c_i} \right\rceil$ bits.

The partial sums are represented by a one-level sampling. For table R, we sample the sum every k values, and store these values in a new table *sumR* using $\lceil \log m \rceil$ bits per field, where m is the number of ones. To obtain the partial sum until position i, we compute $sumR[j] + \sum_{p=jk}^{i} c_p$, where $j = \lfloor i/k \rfloor$, and the summation of the c_p's is done sequentially over the R entries. The positions in S are represented the same way: We store the sampled sums in a new table called $posS$ using $\lceil \log(\sum_{i=1}^{n/u} \lceil \log \binom{u}{c_i} \rceil) \rceil$ bits per field. We compute the position for block i as $posS[j] + \sum_{p=jk}^{i} \lceil \log \binom{u}{c_p} \rceil$. We precompute the 16 possible $\lceil \log \binom{u}{c_p} \rceil$ values in order to speed up this last sequential summation.

With this support, we answer *rank* queries by using the same RRR procedure. Yet, $select(1, i)$ queries are implemented in a simpler and more practical way. We use a binary search over $sumR$, finding the rightmost sampled block for which $sumR[k] \leq i$. Then we traverse table R looking for the block in which we expect to find the i-th bit set (i.e., adding up c_p's until we exceed i). Finally we access this block in table E and traverse it bit by bit until finding the i-th 1. $Select(0, i)$ can be implemented analogously.

3.1.2 Wavelet Trees without Pointers

There exist already Huffman-shaped wavelet tree implementations that achieve close to zero-order entropy space [NM07, FGNV07]. Yet, those solutions are not efficient when the alphabet is very large: The overhead of storing the Huffman symbol assignment and the wavelet tree pointers, $O(\sigma \log n)$, ruins the compression if σ is large. In this section, we present an alternative implementation that achieves zero-order entropy with a very mild dependence on σ (i.e., $O(\log \sigma \log n)$ bits of space), thus extending the existing results to the case of very large alphabets. We use the following bitmaps:

- $DA[1, \sigma]$ stores which symbols appear in the sequence, $DA[i] = 1$ if symbol i appears in S. This allows us to remap the sequence in order to get a contiguous alphabet; using *rank* and *select* over DA we can map in both directions.
- $Occ[1, n]$ records the number of occurrences of symbols $i \leq r \in [1, \sigma]$ by placing a one at $\sum_{i=1}^{r} n_i$. For example, the sequence 113213323, where $n_1 = 3, n_2 = 2$ and $n_3 = 4$, would generate $Occ = 001010001$. In the compressed version of the wavelet tree, this array is stored in compressed form using our RRR implementation.

Our practical implementation of the wavelet tree stores $\lceil \log \sigma \rceil$ bitmaps of length n. The tree is mapped to these bitmaps levelwise: the first bitmap corresponds to the root, the next one corresponds to the concatenation of left and right children of the root, and so on. In this set of bitmaps, we must be able to calculate the interval $[s, e]$ corresponding to the bitmap of a node, and to obtain the new interval $[s', e']$ upon a child or parent operation. Assume the current node is at level ℓ ($\ell = 1$ at the leaves) on a tree of h levels. Further, assume that a is the symbol related to the query, that $\Sigma = \{0, \ldots, \sigma - 1\}$, and that $select_{Occ}(1, 0) = 0$.

- Computing the left child: $s' = s$ and $e' = e - rank(1, e) + rank(1, s - 1)$.
- Computing the right child: $s' = e + 1 - rank(1, e) + rank(1, s - 1)$ and $e' = e$.
- Computing parent: $s' = select_{Occ}(1, \lfloor a/2^\ell \rfloor \cdot 2^\ell) + 1$ and $e' = select_{Occ}(1, (\lfloor a/2^\ell \rfloor + 1) \cdot 2^\ell)$.

Let us explain the left child formula. In the next level, the current bitmap segment is partitioned into a left child and right child parts. The left child starts at the same position of the current segment in this level, so $s' = s$. To know the end of its part, we must add the number of 0s in the current segment, $e' = s + rank(0, e) - rank(0, s - 1) - 1 = s + (e - rank(1, e)) - ((s - 1) - rank(1, s - 1)) - 1$. The right child formula is similar.

For the parent formula, the idea is to consider the binary representation of a, as this is the way our wavelet tree is structured. A node at level ℓ should contain all the combinations of the ℓ lowest bits of a. For example, if $\ell = 1$ and $a = 5 = (101)_2$, its parent is the node at level $\ell = 2$ comprising the symbols $4 = (100)_2$ to $5 = (101)_2$. The parent of this node, at level $\ell = 3$, comprises the symbols $4 = (100)_2$ to $7 = (111)_2$. The formula blurs the last ℓ bits of a and uses $select_{Occ}$ to find the right segments at any level corresponding to the symbol intervals.

We can now navigate the tree and answer every query as if we had the explicit structure.

It is possible to represent the wavelet tree using only one bitmap and omitting the $\lceil \log \sigma \rceil$ pointers; the navigation is similar. We implemented and tested this variation, but the results do not differ much from the first proposal. When compressing the wavelet tree with RRR, the first variation achieves better space because the absolute samples are smaller, thus it uses less space even considering the pointers. The sample used in RRR offers a time/space trade-off for this version.

3.1.3 Golynski et al.'s Structure

We implement Golynski et al.'s proposal rather faithfully, with one important exception: We do not use the Y-Fast trie, but rather perform a binary search over the positions for the *rank* query. In practice, this yields a gain in space and time except for large σ values and biased symbol distribution within the *chunk* (remind that we must search within the range of occurrences of a symbol of Σ in a *chunk* of size σ, that is, the range is $O(1)$ size on average). Hence the time for *rank* is $O(\log \sigma)$ worst case, and $O(1)$ on average. *Select* and *access* are not affected. The version used for permutations [MRRR03] requires $(1+\epsilon)n\lceil \log n \rceil$ bits for n elements and computes π^{-1} in $O(1/\epsilon)$ worst-case time[1].

In the case when $n \approx \sigma$, we also experiment with using only one *chunk* to represent the structure. This speeds up all the operations since we do not need to compute which *chunk* should we query, and all the operations over B become unnecessary, as well as storing B itself.

3.2 Experimental Results

We first test the data structures for binary sequences, on random data and on the BWT of real texts, showing that RRR is an attractive option. Second, we compare the data structures for general sequences on various types of large-alphabet texts, obtaining several interesting results. Finally, we apply our machinery to obtain the best results so far on compressed text indexing.

The machine is a Pentium IV 3.0 GHz with 4GB of RAM using Gentoo GNU/Linux with kernel 2.6.13 and `g++` with `-O9` and `-DNDEBUG` options.

3.2.1 Binary Sequences

We generated three random uniformly and independently distributed bitmaps of length $n = 10^8$, with densities (fraction of 1s) of 5%, 10% and 20%. Fig. 3.1 compares our RRR implementation against the best practical ones in previous work [OS07], considering operations *rank* and *select* (*access* can be implemented as the difference of two *rank*'s, and in some cases it can be done slightly better, yet only some of the structures in [OS07] support it). As control data we include a fast uncompressed implementation [GGMN05], which is insensitive to the bitmap density.

Our RRR implementation is far from competitive for very low densities (5%), where it is totally dominated by `sdarray`, for example. For 10% density it is already competitive with `esp`, its equivalent implementation [OS07], while offering

[1] Thanks to Diego Arroyuelo for providing his implementation for this solution.

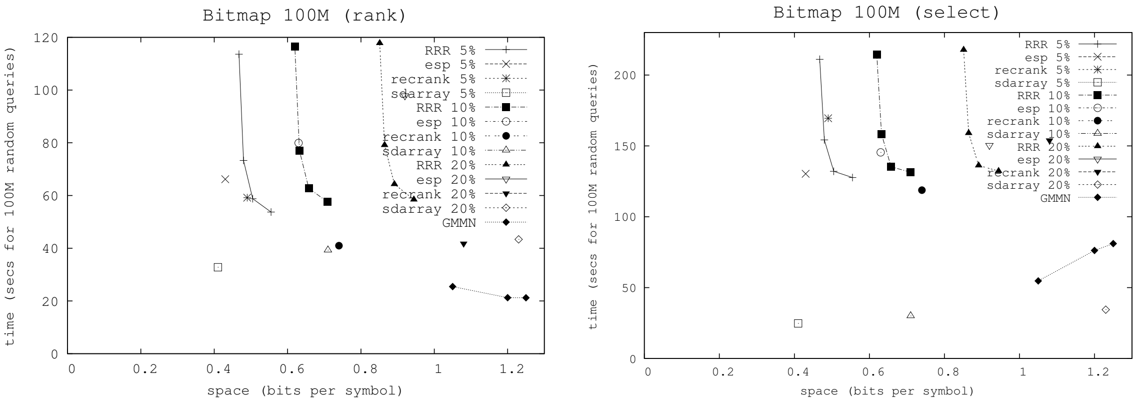


Figure 3.1: Space in bits per symbol and time in seconds for answering 10^8 random queries over a bitmap of 10^8 bits.

more space/time tradeoffs and achieving the best space. For 20% density, RRR is unparalleled in space usage, and alternative implementations need significantly more space to beat its time performance. We remark that RRR implements $select(0, i)$ in the same time as $select(1, i)$ with no extra space, where competing structures could have to double the space they devote to *select* to retain their performance.

A property of RRR that is not apparent over uniformly distributed bitmaps, but which becomes very relevant for implementing the wavelet tree of a BWT-transformed text, is its ability to exploit local regularities in the bit sequence. To test this, we extracted the 50MB `English` text from *Pizza&Chili* (`http://pizzachili.dcc.uchile.cl`), computed the balanced wavelet tree of its BWT, and concatenated all the bitmaps of the wavelet tree levelwise (this corresponds to variant WT in Section 3.2.2, where the space directly depends on the compression of the bit sequence). Table 3.1 shows the compression achieved by the different methods. Global methods like `sdarray` and `recrank` fail, taking more space than an uncompressed implementation. RRR stands out as the clear choice for this problem, followed by `esp` (which is based on the same principle). The bitmap density is around 40%, yet RRR achieves space similar to 5% uniformly distributed density. We will explore further consequences of this fact in Section 3.2.3.

3.2.2 General Sequences

We compare our implementations of Golynski et al.'s and different variants of wavelet trees. We consider three alphabet sizes. The smaller one is byte-size: We consider our plain text sequences `English` and `DNA`, seeing them as character sequences. Next, we consider a large alphabet, yet not large enough to compete with the sequence size: We take the 200MB `English` text from *Pizza&Chili* and regard this as a sequence of *words* (a word is a maximal string of letters and digits). The result is a sequence of $46,582,195$ words over an alphabet of size $270,096$. Providing *access* and *select* over this sequence mimics a word-addressing inverted index functionality [BFLN08]. Finally, we consider a case where the alphabet is nearly as large as the text. The sequence corresponds to the graph `Indochina` after applying *Re-Pair* compression on its adjacency list (see Chapter 6). The result is a sequence of length $15,568,253$ over an alphabet of size $13,502,874$. The result of *Re-Pair* can still be compressed with a zero-order compressor, but the size of the alphabet challenges all of the traditional methods. Our techniques can achieve such compression and in addition provide *rank*/*select* functionality, which supports backward traversal on the graph for free (see Chapter 7).

We consider full and 1-chunk variants of Golynski, as well as different combinations of choices for wavelet trees. Note that DA and RRR can be combined in 4 ways. RRR can also be combined with the last two options. Each plot will omit obviously non-competitive combinations.

- WT: Concatenates all the wavelet tree levels into $\lceil \log \sigma \rceil$ bitmaps, which are not compressed. Spurious symbols are added to the sequence to make the alphabet contiguous.
- WT+DA: Instead of adding spurious symbols, maps the sequence to a contiguous range using a bitmap (see Section 3.1.2).
- WT+RRR: Uses RRR compressed bitmap representation instead of a plain one.

Variant	Size	*Rank* time
`sdarray`	2.05	> uncompressed
`recrank`	1.25	> uncompressed
`esp`	0.50	0.594
RRR (ours)	0.48	0.494
uncompressed	1.05	0.254

Table 3.1: Space (as a fraction of the bitmap size) and *rank* time (in μsec per query) achieved by different data structures on the wavelet tree bitmaps of a real BWT transformed text.

- WT Ptrs: Uses pointers to keep the (balanced) tree shape, so it wastes σ pointers for them. Uses uncompressed bitmaps within the tree nodes. No contiguous alphabet is necessary here.

- WT Ptrs+Huff: Gives the tree a Huffman tree shape, so it approaches zero-order entropy at the price of wasting σ words of space for the Huffman codes.

Figure 3.2 shows the results for byte alphabets. Golynski's structure is not competitive here. This shows that their $o(\log \sigma)$ term is not yet negligible for $\sigma = 256$. On wavelet trees, the Ptrs+Huff variant excels in space and in time. Adding RRR reduces the space very slightly in some cases; in others keeping balanced shape gives better time in exchange for much more space. We also included Naive, an implementation for byte sequences that stores the plain sequence plus regularly sampled *rank* values for all symbols [BFLN08]. The results show that we could improve their wavelet trees on words by replacing their Naive method by WT Ptrs (as the alphabet is of size 127 and uniformly distributed in that application).

Figure 3.3 shows experiments on larger alphabets. Here Golynski et al.'s structure becomes relevant. On the sequence of words, it adds to the previous scenario a third space/time tradeoff point, offering much faster operations (especially *select*) in exchange for significantly more space. Yet, this extra space is acceptable for the sequence of word identifiers, as overall it is still a representation taking 70% *of the original text*. As such it competes with a word-addressing inverted index. This follows a recent trend of replacing inverted indexes by a succinct data structure for representing sequences of word identifiers [BFLN08], which can retrieve the text using *access* but also find the consecutive positions of a word using *select*. Golynski et al.'s implementation takes around 2 μsec for *select*. The structure used in that work [BFLN08] requires just 55% of the text space to achieve the same performance. Yet some simple optimizations (such as treating stopwords and separators in a different way) could close the gap. Another tradeoff point is WT Ptrs+Huff, which takes 40% of the original text and 15 μsec for *select*. The other structure [BFLN08] requires around 35% of the text for about the same time.

Again, adding RRR reduces the space of Ptrs+Huff, yet this time the reduction is more interesting, and might have to do with some locality in the usage of words across a text collection.

For the case of graphs, where the alphabet size is close to the text length, the option of using just one chunk in Golynski et al.'s structure becomes extremely relevant. It does not help to further compress the Re-Pair output, but for 20% extra space it provides very efficient backward graph traversal (see Chapter 7). On the

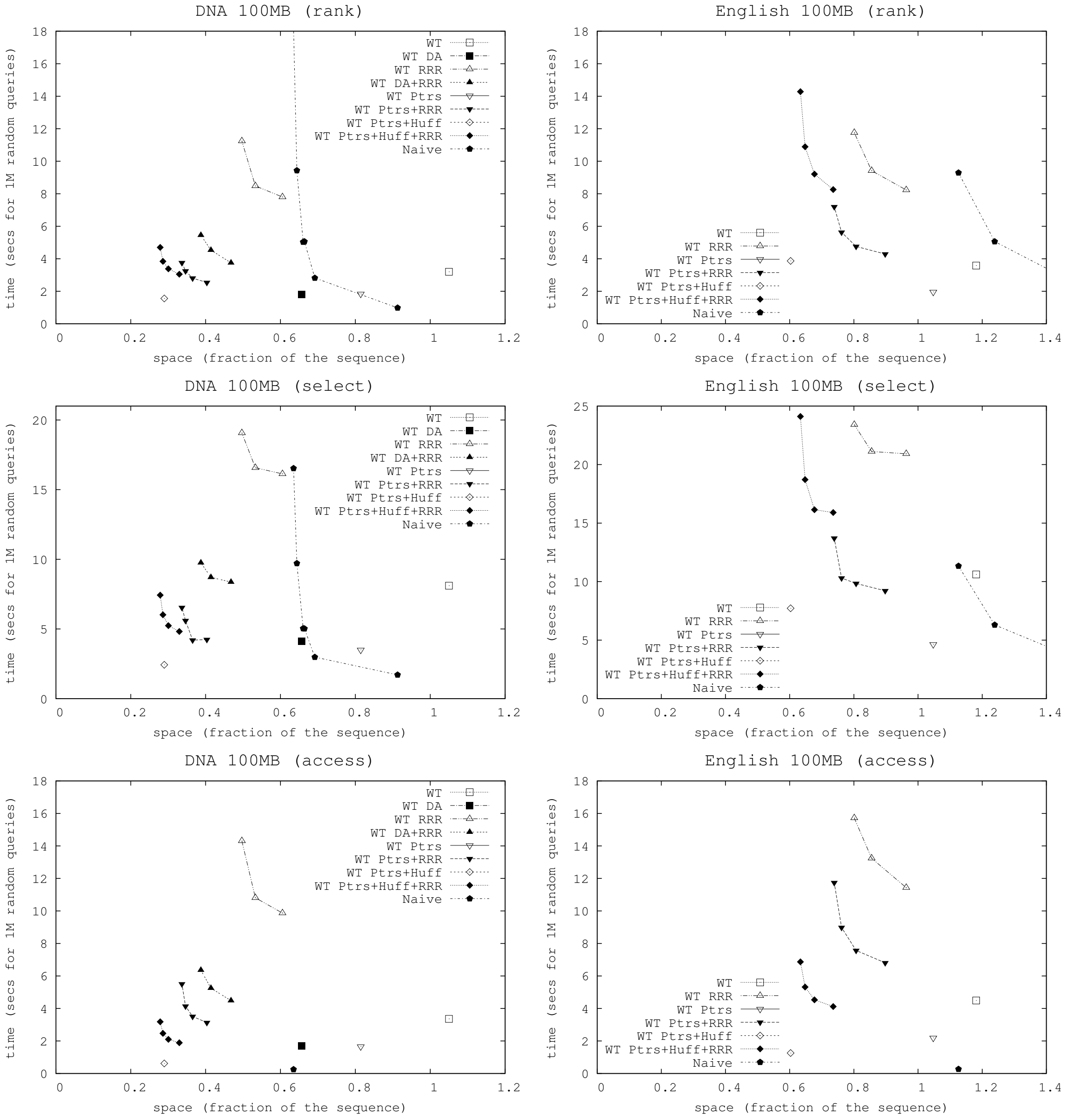


Figure 3.2: Results for byte alphabets. Space is measured as a fraction of the sequence size (assuming one byte per symbol) and the time is measured in seconds for 10^6 queries.

other hand, the wavelet trees with DA+RRR offer further compression up to 70% of the Re-Pair output size, which is remarkable for a technique that already achieved excellent compression results (see Chapter 6). The price is much higher access time. Without RRR, the times are in between both extremes, with no compression yet no space penalty either. An interesting point here is that the versions with pointers are not applicable here, as the alphabet overhead drives their space over 3 times the sequence size. Hence exploring versions that do not use pointers pays off.

3.2.3 Compressed Full-Text Self-Indexes

It was recently proved [MN07] that the wavelet tree of the Burrows-Wheeler transform (BWT) of a text (the key ingredient of the successful FM-index family of text self-indexes [NM07]), achieves high-entropy space without any further sophistication, provided the bitmaps of the wavelet tree are represented using RRR structure [RRR02]. Hence a simple and efficient self-index emerges, at least in theory. In Section 3.2.1, we showed that RRR indeed takes unique advantage from the varying densities along the bitmap typical of the BWT transform. We can now show that this proposal [MN07] has much practical value.

Fig. 3.4 compares the best suffix-array based indexes from *Pizza&Chili*: SSA, AFFM-Index, RLFM-Index and the CSA (see Section 2.6). We combine SSA with our most promising versions for this setup, WT Ptrs+RRR and WT Ptrs+Huff+RRR. All the spaces are optimized for the *count* query, which is the key one in these self-indexes (the others depend linearly on the product of these times and a suffix array sampling step parameter, which we set to ∞).

We built the index over the 100 MB texts `English`, `DNA`, `Proteins`, `Sources`, `Pitches`, and `XML` provided in *Pizza&Chili*. We chose 10^5 random patterns of length 20 from the same texts and ran the *count* query on each. Each *count* triggers 40 *rank* operations per wavelet tree level (in Huffman shaped trees this quantity could change).

As can be seen, our new implementation is extremely space-efficient, achieving a space performance *never seen before* in compressed full-text indexing (see, e.g., `English`, but also the hard-to-compress `DNA` and `Proteins`). In some cases, there is no even time penalty!

It is interesting to question why combining RRR with Huffman shape is better than RRR alone, since RRR by itself should in principle exploit all of the compressibility captured by Huffman. The answer is in the c component of the (c, o) pairs of RRR, which poses a fixed overhead per symbol which is not captured by

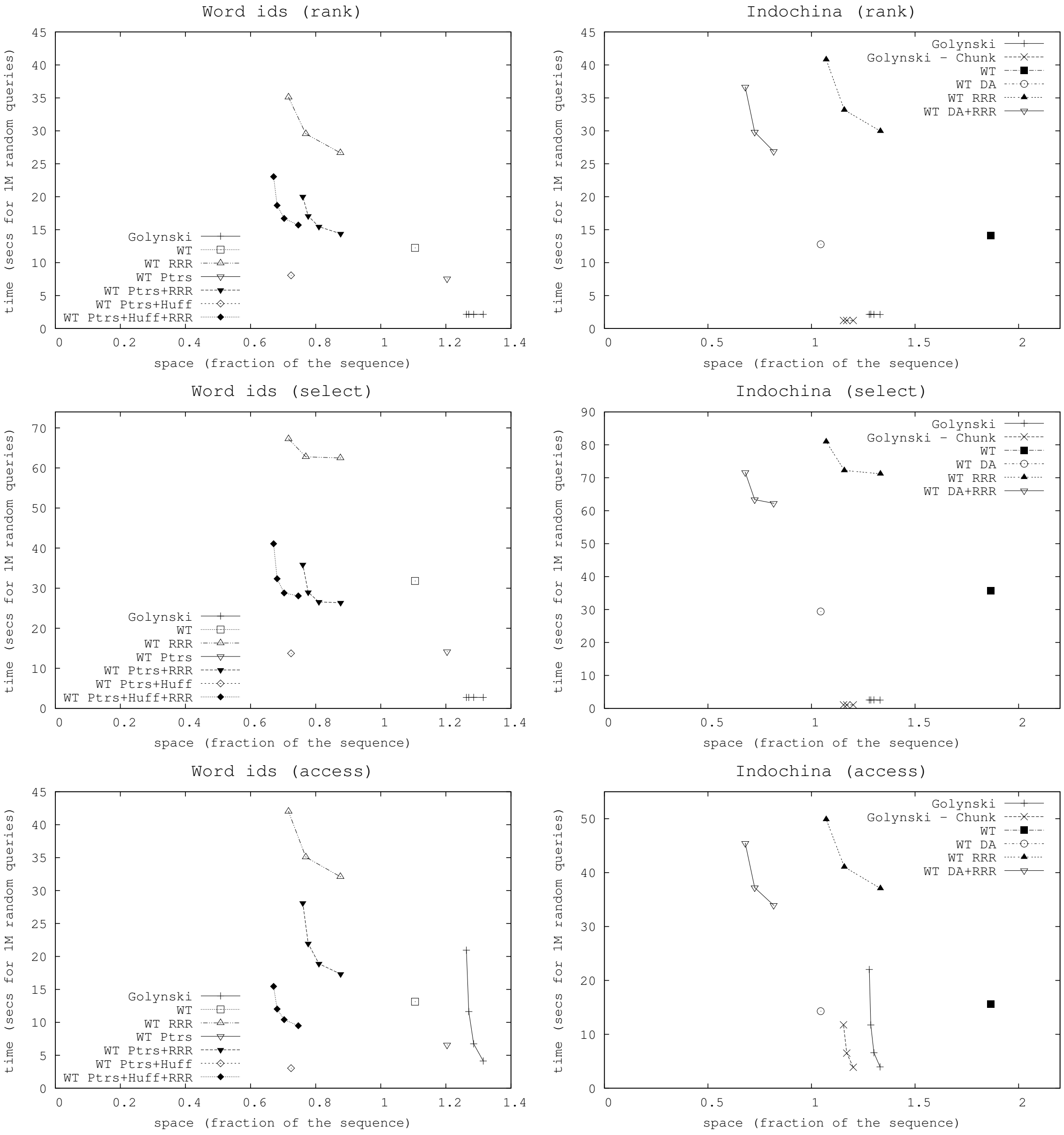


Figure 3.3: Results for word identifiers (left) and a graph compressed with Re-Pair (right). Space is measured as a fraction of the sequence (using $\lceil \log \sigma \rceil$ bits per symbol) and the time is measured in seconds for 10^6 queries.

the entropy. Indeed, we measured the length of the o components (table S) in both cases and the difference was 0.02%. Yet the difference among the c components was 56.67%. Thus the Huffman shape helps reduce the total number of symbols to be indexed, and hence it reduces the overhead due to the c components.

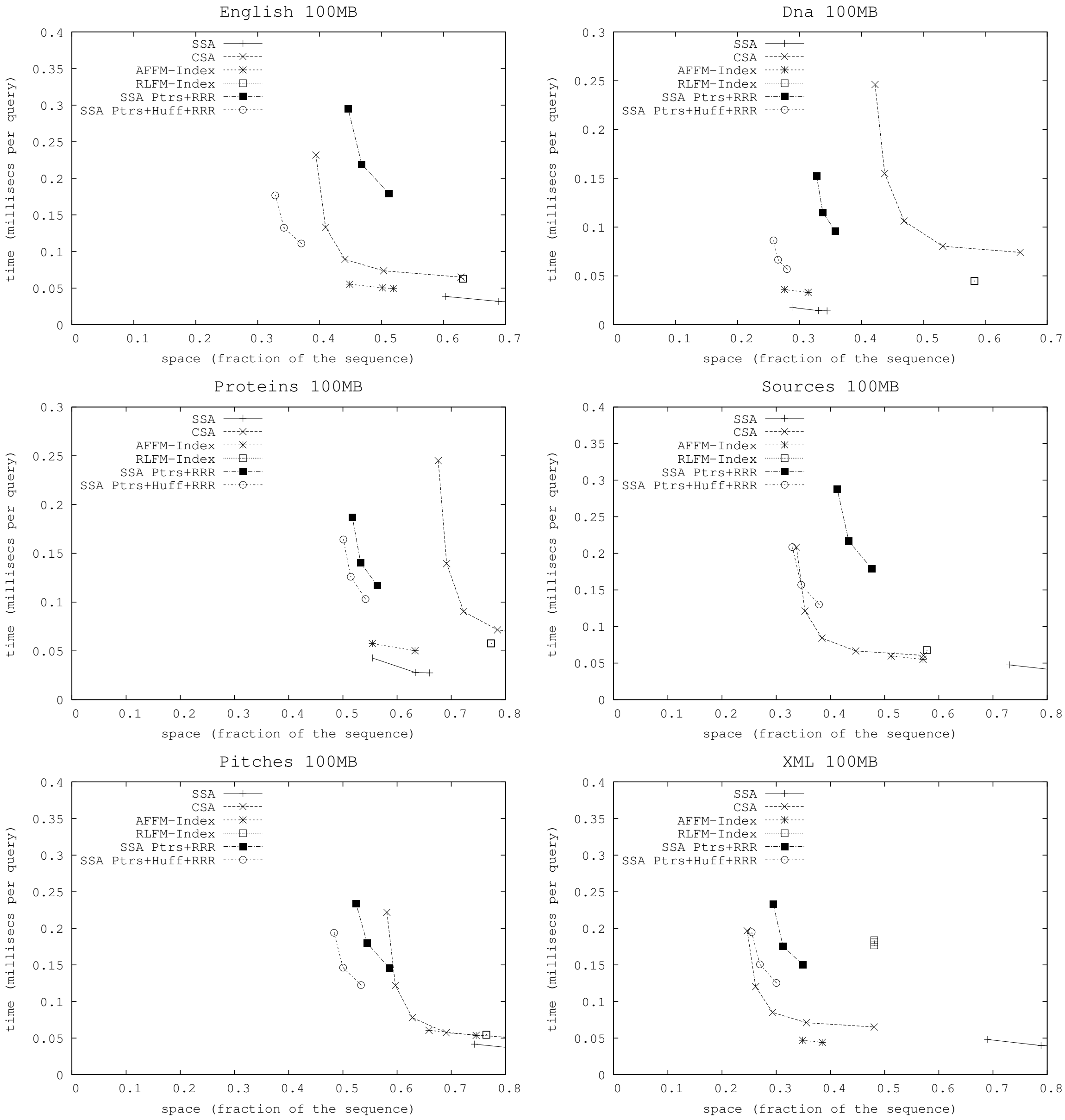


Figure 3.4: Times for counting, averaged over 10^5 repetitions, for patterns of length 20.

Chapter 4

Re-Pair and Lempel-Ziv with Local Decompression

4.1 Re-Pair

Recall from Section 2.4.1 that the exact Re-Pair compression algorithm requires too much main memory and hence it is not suitable for compressing large sequences. We present now an approximate Re-Pair compression method that: (1) works on any sequence; (2) uses as little memory as desired on top of T; (3) given a fixed extra memory to work, can trade accurateness for speed; (4) is able to work smoothly on secondary memory due to its sequential access pattern.

4.1.1 Approximate Re-Pair

In this section, we describe the method assuming we have $M > |T|$ main memory available, that is, the text plus some extra space fit in main memory. Section 4.1.2 considers the case of larger texts.

We place T inside the bigger array of size M, and use the remaining space as a (closed) hash table H of size $|H| = \min(M - |T|, 2|T|)$. Table H stores unique pairs of symbols $ab = t_i t_{i+1}$ occurring in T, and a counter of their number of occurrences in T. The key $ab = t_i t_{i+1}$ is represented as a single integer by its position i in T (any occurrence works). Thus each entry in H requires two integers.

The algorithm carries out several *passes*. At each pass, we identify the k most promising replacements to carry out, and then try to materialize them. Here $k \geq 1$

is a time/quality tradeoff parameter. At the end, the new text is shorter and the hash table can grow. We detail now the steps carried out for each pass.

Step 1 (counting pair frequencies). We traverse $T = t_1 t_2 \ldots$ sequentially and insert all the pairs $t_i t_{i+1}$ into H. If, at some point, the table surpasses a load factor $0 < \alpha < 1$ (defined by efficiency considerations, as the expected insertion time is $O\left(\frac{1}{1-\alpha}\right)$ [Knu98]), we do not insert new pairs anymore, yet we keep traversing T to increase the counters of already inserted pairs. This step requires $O\left(\frac{1}{1-\alpha}|T|\right) = O(n)$ time on average. H stores the counters and the position of one occurrence of the pair, so we need just one integer to represent the pair instead of two.

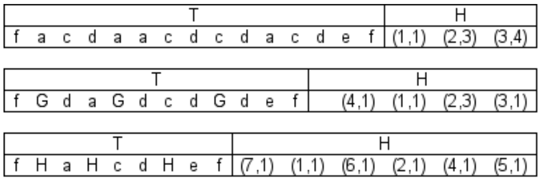
Step 2 (finding k promising pairs). We scan H and retain the k most frequent pairs from it. A heap of k pointers to cells in H is sufficient for this purpose. Hence we need also space for k further integers. This step requires $O(|H| \log k) = O(n \log k)$ time.

Step 3 (simultaneous replacement). The k pairs identified will be simultaneously replaced in a single pass over T. For this sake we must consider that some replacements may invalidate others, for example we cannot replace both *ab* and *bc* in *abc*. Some pairs can have so many occurrences invalidated that they are not worthy of replacement anymore (especially at the end, when even the most frequent pairs occur a few times). These considerations complicate the process.

We first empty H and reinsert only the k pairs to be replaced. This time we store the explicit key *ab* in the table, as well as a field *pos*, the position of its first occurrence in T. Special values for *pos* are *null* if we have not yet seen any occurrence in this second pass, and *proceed* if we have already started replacing it. We now scan T and use H to identify pairs that must be replaced. If pair *ab* is in H and its *pos* value is *null*, then this is its first occurrence, whose position we now record in *pos* (that is, we do not immediately replace the first occurrence until we are not sure there will be at least two occurrences to replace). If, on the other hand, its *pos* value is *proceed*, we just replace *ab* by *sz* in T, where s is the new symbol for pair *ab* and z is an invalid entry. Finally, if the pair *ab* already has a first position recorded in *pos*, we read this position in T and if it still contains *ab* (after possible replacements that occurred since we saw that position), then we make both replacements and set the *pos* value to *proceed*. Otherwise, we set the *pos* value of pair *ab* to the current occurrence we are processing (i.e., its new first position). This method ensures that we create no new symbols s that will appear just once in T. It takes $O(|T|) = O(n)$ time on average.

Step 4 (compacting T and enlarging H). We compact T by deleting all the z entries, and restart the process. As now T is smaller, we can have a larger hash table of size $|H| = \min(M - |T|, 2|T|)$. The traversal of T, regarded as a circular array, will now start at the point where we stopped inserting pairs in H in Step 1 of the previous pass, to favor a uniform distribution of the replacements. This step takes $O(|T|) = O(n)$ time.

Figure 4.1 illustrates the execution of the algorithm on an example sequence.



T	H
f a c d a a c d c d a c d e f	(1,1) (2,3) (3,4)

T	H
f G d a G d c d G d e f	(4,1) (1,1) (2,3) (3,1)

T	H
f H a H c d H e f	(7,1) (1,1) (6,1) (2,1) (4,1) (5,1)

Figure 4.1: Part of the execution of the approximate version of Re-Pair. H represents the space used for the hash table and T the space used for the text. The arrows point from the counter in the hash table to the first occurrence of the pair counted by that field. In the first iteration we replace ac by G; we do not replace cd because ac blocks every replacement. During the second iteration we replace Gd by H.

Approximate analysis. Although not being complete, the following analysis helps understand the accuracy/time tradeoff involved in the choice of k. Assume the exact method creates $|R|$ new symbols. The approximate method can also carry out $|R|$ replacements (achieving hopefully similar compression, since these need not be the same replacements of the exact method) in $p = \lceil |R|/k \rceil$ passes, which take overall average time $O(\lceil |R|/k \rceil \; n \log k)$. Thus we can trade time for accurateness by tuning k. The larger k, the faster the algorithm (as there is an $O(\log(k)/k)$ factor), but the less similar the result compared to the exact method. This analysis, however, is only an approximation, as some replacements could be invalidated by others and thus we cannot guarantee that we carry out k of them per round. Hence p may be larger than $\lceil |R|/k \rceil$ for some texts.

Note that even $k = 1$ does not guarantee that the algorithm works exactly as Re-Pair, as we might not have space to store all the different pairs in H for some values of α and sizes of H. In this respect, it is interesting that the algorithm becomes more accurate (thanks to a larger H) in its later stages, as by that time the frequency distribution is flatter and more precision is required to identify the best pairs to replace.

4.1.2 Running on Disk

The process described above also works well if T is too large to fit in main memory. In this case, we maintain T on disk and table H occupies almost all the main memory, $|H| \approx M < |T|$. We must also reserve sufficient main memory for the heap of k elements. To avoid random accesses to T in Step 1, we do not store anymore in H the position of pairs ab, but instead ab explicitly. Thus Step 1 carries out a sequential traversal of T. Step 2 runs entirely in main memory. Step 4 involves another sequential traversal of T.

Step 3 is, again, the most complicated part. In principle, a sequential traversal of T is carried out. However, when a *pos* value changes to *proceed*, we make two replacements: one at its first occurrence (at value *pos*) and one at the current position in the traversal of T. The first involves a random access to T. Yet, this occurs only when we make the first replacement of an occurrence of a pair ab. This occurs at most k times per pass. However, checking that the first position *pos* still contains ab and has not been overwritten, involves another random access to T, and these cannot be bounded.

To carry out Step 3 efficiently, we note that there are at most k positions in T needing random access at any time, namely, those containing the *pos* ($\notin \{null, proceed\}$) values of the k pairs to be replaced. We maintain those k disk pages cached in main memory[1]. Those must be replaced whenever value *pos* changes. This replacement does not involve reading a new page, because the new *pos* value always corresponds to the current traversal position (whose block is also cached in main memory). Thus cached pages not pointed anymore from any *pos* values are simply discarded (an elementary reference counting mechanism is sufficient), and the current page of T might be retained in main memory if, after processing it, some *pos* values now point to it.

As explained, most changes to T are done at the current traversal position, hence it is sufficient to write back the current page of T after processing it to handle those changes. The exceptions are the cases when one writes at some old position

[1] Note that the pairs could span two pages, in that case we need $2k$ pages at most. We thank Hernán Arroyo for pointing this.

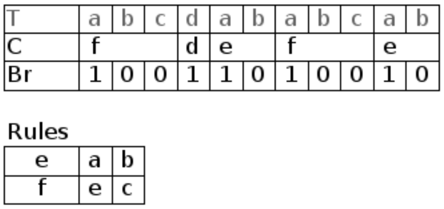
pos. In those cases the pages we have cached in main memory must be written back to disk. Yet, as explained, this occurs at most k times per pass. (Note that using a dirty bit for the cached pages might avoid some of those write-backs, as the dirty page could be modified several times before being abandoned by all the pairs.)

Thus the worst-case I/O cost of this algorithm, if p passes are carried out, is $O(p\ (n/B + k))$, where B is the disk block size. That is, the algorithm is almost I/O optimal with respect to its main memory version. Indeed, it is asymptotically I/O optimal if k is chosen to be in $O(n/B)$, a rather reasonable condition.

4.1.3 Adding Local Decompression

After compressing a text with Re-Pair, we can decompress the whole file but cannot access any piece of it at a random position without decompressing all of the preceding data.

In order to allow random access to the compressed file, we add some structures that allow fast decompression of any snippet of the text, by expanding the optimal number of symbols. The idea is to add a rank/select-capable structure `Br` in which we mark every position of the real text where a phrase of Re-Pair begins. For example, imagine the text `abcdababcab` where we replace $e \rightarrow$`ab` and $f \rightarrow e$`c`. The result of C and `Br` is shown in Figure 4.2. This idea was originally presented in [GN07] for locally decompressing suffix arrays.



T	a	b	c	d	a	b	a	b	c	a	b
C	f			d	e		f			e	
Br	1	0	0	1	1	0	1	0	0	1	0

Rules

e	a	b
f	e	c

Figure 4.2: Example over $T =$`abcdababcab`.

For extracting the snippet that starts at position i and ends at position j, we determine which symbols in C contain those positions. The substring of C that we have to decompress is $C[i', j']$ where $i' = \texttt{rank}_{\texttt{Br}}(i)$ and $j' = \texttt{rank}_{\texttt{Br}}(j)$. The only problem is that we must find out what is the real position of i' in the text. That can be answered with `Br` by calculating the start position $s = \texttt{select}_{\texttt{Br}}(i')$. After that, the extraction is trivial: We start expanding $C[i', j']$ knowing that the first symbols will correspond to position s, and expand until position i. Then we start reporting until reaching position j, which is achieved while expanding $C[j']$.

4.2 Local Decompression on Lempel-Ziv

We add a bitmap like `Br` to the structure. This new bitmap stores every position where a phrase ends. The decompression can be done pretty much in the same way as for Re-Pair. We just ask in which phrase, j', the position j is, and then decompress backwards until reaching position i. The decompression could expand many phrases; the important fact is that if we reach the end of the phrase and we have not reached position i, we continue with phrase $j' - 1$ and then with $j' - 2$ and so on. Note that every phrase is similar to a symbol in C that is expanded, thus the idea is the same as for Re-Pair: It expands the optimal number of phrases just as Re-Pair expands the optimal number of symbols.

4.3 Experimental Results

In order to validate our approximate Re-Pair method, we compared the compression ratios obtained with the original method with those obtained with our version. The parameter that determines the space used by the hash table does not include the space used by the heap. Thus one has to add $O(k)$ words to the space requirements, which is negligible in our experiments.

We measured the compression ratio for several texts downloaded from Pizza&Chili (`http://pizzachili.dcc.uchile.cl`). Table 4.1 shows the compression ratios obtained for the six 200MB files. We can see that the approximate version of Re-Pair achieves better compression ratio than LZ78 and is comparable with other common compressors. The compression lost by applying the approximate version instead of the original method is acceptable $(1\% - 5\%)$, especially if we focus on large Web graphs, for which the approximate version might be the only option. We use 50% extra space for the hash table and $k = 10000$. The value of α is 0.8.

In order to decompress locally we need to add the bitmap `Br`. Table 4.2 shows the space used by this bitmap using different implementations. The space is measured as percentage of the implementation of González et al. [GGMN05], which requires $1.05n$ bits. The first implementation shown is our

Text	Exact Re-Pair	App. Repair	LZ78	gzip -1	gzip -9	bzip2 -1	bzip2 -9
xml.200MB	13.93%	16.72%	21.76%	21.19%	17.12%	14.37%	11.36%
english.200MB	29.06%	33.53%	42.13%	44.99%	37.64%	32.83%	28.07%
dna.200MB	34.37%	35.41%	30.25%	32.51%	27.02%	26.57%	25.95%
proteins.200MB	51.18%	52.32%	57.37%	48.86%	46.51%	45.74%	44.80%
sources.200MB	25.71%	30.51%	40.50%	28.01%	22.38%	21.30%	18.67%
pitches.50MB	53.92%	57.31%	61.30%	33.92%	33.57%	34.99%	36.12%

Table 4.1: Compression ratio (size of compressed file compared with the original one) for exact Re-Pair, our approximate Re-Pair, LZ78, gzip and bzip2.

version of RRR (see Chapter 3 [RRR02]). The second implementation is an unpublished work from González and Navarro that is available in the site `http://pizzachili.dcc.uchile.cl`. As we can see, the space can be considerably reduced with the last two implementations.

File	*RRR*	*GN*
xml.200MB	45.51%	40.12%
dna.200MB	66.58%	64.84%
english.200MB	56.22%	55.32%
pitches.50MB	66.79%	69.71%
proteins.200MB	63.96%	68.16%
sources.200MB	53.80%	51.63%

Table 4.2: Size of compressed representation of Br (as a percentage of [GGMN05]).

Finally, Figure 4.3 shows the time for decompressing a random snippet from a compressed text stored in main memory for the approximate version of Re-Pair and our implementation of LZ78. Both use the bitmap `Br` represented with the implementation of González et al. [GGMN05]. Figure 4.3 shows the plots for every text. The LZ78 implementation consistently outperforms Re-Pair, yet is uses significantly more space (see also Chapter 6).

We conclude that dictionary-based methods are a good alternative for achieving acceptable compression ratios, together with good performance for local decompression.

Lempel-Ziv achieves faster decompression because the phrases are shorter on average, so Re-Pair has to decompress larger symbols. For the same reason, Re-Pair yields a better compression ratio.

We do not focus on reducing much the space of the bitmap `Br` since the local decompression for Web graphs uses a denser (but smaller) bitmap. The

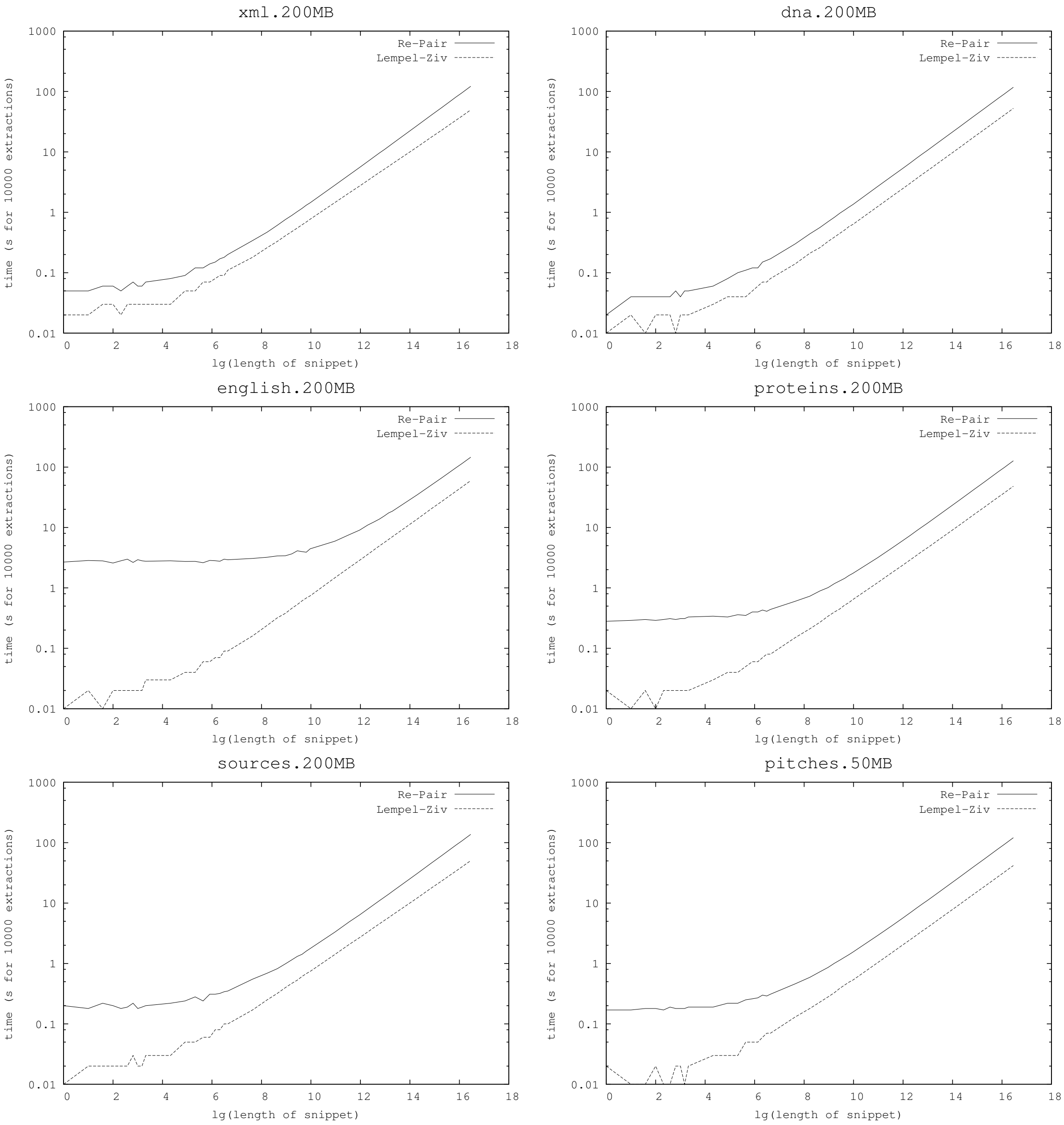


Figure 4.3: Local decompression times for several texts.

representation using Re-Pair achieves better space and the speed can be improved by representing the dictionary using plain integers.

Chapter 5

Edge List Representation

In this chapter[1], we present a textual representation for a directed graph $G = (V, E)$. The main idea of the new representation is to encode all the graph edges, adjacency-list-wise, and build a text with the concatenation of all the lists.

Let $n = |V|$ and $m = |E|$, and $\delta(v) = |\{u,\ (v, u) \in E\}|$. For every node $v \in V$ we will have two identifiers, v and $\bar{v}$.

Definition 3 (Encoded edge) *The edge (u, v) is encoded as the concatenation of the identifier of u and the second identifier of v, that is, $u\bar{v}$. We will assume that $\forall v, u \in V, \bar{v} < u$.*

Definition 4 (Encoded adjacency list) *Every adjacency list of a node $v \in V$ is encoded as the concatenation of the encoded edges that leave from v followed by the identifier of v. For example if the adjacency list of v is $\{u_1, u_2, u_3\}$, the encoded adjacency list is $v\bar{u_1}v\bar{u_2}v\bar{u_3}v$. However, if $\delta(v) = 0$ the encoded list is the empty string, ϵ (as opposed to v).*

Finally, the graph itself is written as a text by concatenating all the encoded adjacency lists, we call $T = T(G)$ this text. The space needed by this representation is obviously larger than the classical adjacency list. In fact:

$$|T| \quad = \quad \sum_{\substack{v \in V \\ \delta(v) > 0}} \quad (1 + 2\delta(v))$$

[1] This is a joint work with Paolo Ferragina, University of Pisa.

where $|T|$ is measured in number of node identifiers.

This representation offers a very simple way to answer every query. We now explain how each query is answered, assuming the queries are over a node v.

1. Adjacency list: report the node following each occurrence of v in the text, except for the last one.
2. Reverse adjacency list: report the node following each occurrence of $\bar{v}$ in the text.
3. Outdegree: report the number of occurrences of v minus one.
4. Indegree: report the number of occurrences of $\bar{v}$.
5. Existence: edge (v, u) exists if and only if $v\bar{u}$ appears in the text.

We use a self-index to answer these queries, namely the CSA of Sadakane [Sad03]. We use structures D and Ψ, described in Section 2.6.2. Based on these two structures we explain how to answer the queries just described, again assuming the query is over a node v.

The bitmap D (as the structure that answers Occ) works only if the alphabet is contiguous. We assume that we have a function `map` that associates a node identifier with a symbol in the text, and `unmap` which is the inverse of `map`. The symbols $\bar{v}$ are represented as $-v$ (before applying the `map` function).

In order to avoid the special conditions arising when $T(v) = \epsilon$, we renumber the nodes: the first range is assigned to the nodes with $T(v) = \epsilon$ that are pointed by other nodes (type I), then the nodes with non-empty adjacency list (type II) and finally the nodes with outdegree and indegree 0 (type III).

Call n_I the number of nodes of type I, n_{II} the nodes of type II, and n_{III} those of type III. Figure 5.1 gives the mapping function using during the rest of this chapter. It also shows the idea behind the function: The marked region is mapped into a contiguous range.

Figures 5.2 to 5.5 show the algorithms for the first four queries.

```
map($v$)
If $v > 0$ Then
    $v \leftarrow v - n_I$
$v \leftarrow v + n_{II}$
```

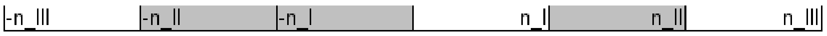


Figure 5.1: Map function for the CSA-based representation, and schematic idea: the function maps the grey areas to a contiguous interval starting at 0.

```
$i \leftarrow$ rank($D$, map($v$))
$j \leftarrow$ rank($D$, map($v$) $+ 1) - 2$
For $k \leftarrow i$ to $j$ Do
    report(unmap(rank($D$, $\Psi[k]$)))
```

Figure 5.2: Retrieval of the adjacency list in the CSA-based representation.

```
$i \leftarrow$ rank($D$, map($\bar{v}$))
$j \leftarrow$ rank($D$, map($\bar{v}$) $+ 1) - 1$
For $k \leftarrow i$ to $j$ Do
    report(unmap(rank($D$, $\Psi[k]$)))
```

Figure 5.3: Retrieval of the reverse adjacency list in the CSA-based representation.

```
$i \leftarrow$ rank($D$, map($v$))
$j \leftarrow$ rank($D$, map($v$) $+ 1)$
return $j - i - 1$
```

Figure 5.4: Computation of the outdegree in the CSA-based representation.

```
$i \leftarrow \texttt{rank}(D, \texttt{map}(\bar{v}))$
$j \leftarrow \texttt{rank}(D, \texttt{map}(\bar{v}) + 1)$
return $j - i$
```

Figure 5.5: Computation of the indegree in the CSA-based representation.

5.1 Building the Index

The index was implemented in a 32-bit machine that allows only 3GB of RAM per process. The initial experiments were run using the UK crawl (downloaded from `http://law.dsi.unimi.it/`, see Chapter 6) whose size is around 1.2GB, so the edge representation takes more than 2GB. Hence, the construction of the suffix array and Ψ had to be carried out on disk. Figure 5.6 shows a simple folklore method for the construction of the suffix array on disk. The main idea is to make several passes on the text, sorting consecutive lexicographic ranges of suffixes. In each pass, we need to sort at least the suffixes starting with one symbol. Since the maximal indegree and outdegree of Web graphs is not so big, this is not a problem. Figure 5.7 shows the construction of the inverted suffix array, and finally Figure 5.8 shows the construction of Ψ on disk. Those methods are tailored to the case where we can hold $|T|$ integers in RAM, but not $2|T|$.

5.2 Compressing the Index

We have shown how to represent the graph as a text and how to answer the desired queries using a *CSA full-text self-index* structure. The structure itself is very promising from a performance point of view, but unfortunately its size makes it unpractical.

The standard *CSA* is able of compressing a text to $nH_0(T) + O(n \log\log\sigma)$, and run-length compression on Ψ can achieve space proportional to $H_k(T)$ for $k > 0$. The problem of our representation is that every substring of size 2 in T is unique, thus we cannot hope to achieve $H_2(T)$ space ($H_2(T) = 0$). We experimented with compressing Ψ using a truncated Huffman code (representing symbols until a given number and escaping the others). We also tried to exploit runs in Ψ, but there was no noticeable improvement. Even compressing it with Gamma Codes and Rice

```
**buildSA**($T[1\ldots n], C[1\ldots\sigma], buffer[1\ldots k]$)
    $l \leftarrow 1$
    $r \leftarrow 1$
    **While** $l \leq \sigma$ **Do**
        $acc \leftarrow 0$
        **While** $r \leq \sigma$ and $acc + C[r] \leq k$ **Do**
            $acc \leftarrow acc + C[r]$
            $r \leftarrow r + 1$
        **If** $r = l$ **Then** $r \rightarrow l + 1$
        **For** $i \leftarrow 1\ldots n$ **Do**
            **If** $l \leq T[i] < r$ **Then**
                append $i$ to $buffer$
        $l \leftarrow r$
        sort($buffer$)
        write $buffer$ to disk
```

Figure 5.6: Algorithm for building the suffix array on disk, when T and $buffer$ fit in main memory. The C array counts how many times does each symbol appear in T. The function *sort* alphabetically sorts the positions stored in $buffer$.

```
**buildiSA**($SA[1\ldots n], buffer[1\ldots k]$)
    **For** $i \leftarrow 1\ldots n/k$ **Do**
        **For** $j \leftarrow 1\ldots n$ **Do**
            **If** $(i-1)k + 1 \leq SA[j] \leq ik$ **Then**
                $buffer[SA[j] - (i-1)k] \leftarrow j$
        write $buffer$ to disk
```

Figure 5.7: Algorithm for building the inverted suffix array on disk. We assume $n \bmod k = 0$ and that SA and $buffer$ fit in main memory.

```
**buildPsi**($iSA[1 \ldots n], buffer1[1 \ldots k], buffer2[1 \ldots k]$)
    **For** $i \leftarrow 1 \ldots n/k$ **Do**
        load $SA[(i-1)k+1 \ldots ik]$ into $buffer1$
        **For** $j \leftarrow 1 \ldots k$ **Do**
            **If** $buffer1[j] = n$ **Then** $buffer2[j] \leftarrow iSA[1]$
            **Else** $buffer2[j] \leftarrow iSA[buffer1[j]+1]$
        write $buffer2$ to disk
```

Figure 5.8: Algorithm for building Ψ on disk, where $buffer1$, $buffer2$ and iSA fit in main memory. We assume $n \texttt{ mod } k = 0$.

Method	Bits per edge
Huffman	39.28
Rice Codes	55.84
Gamma Codes	58.56

Table 5.1: Compression ratio, $|\Psi|/m$ in bits/edge, for the UK crawl. Within this space we support reverse navigation as well.

Codes, the results were all poor. Table 5.1 shows the sizes obtained using the ideas above.

A way to improve the result is to note that if two nodes have similarity in their adjacency list, then the differences in Ψ would repeat too. This allows us to use a compressor that could exploit this regularity, that is, common sequences in the differences of Ψ. For example, let us assume we have two nodes v_1 and v_2 for which $\texttt{map}(v_1) = \texttt{map}(v_2) - 1$, and the two nodes point to u_1, u_2 and u_3. The text would look like this:

$$\ldots v_1\bar{u}_1v_1\bar{u}_2v_1\bar{u}_3v_1v_2\bar{u}_1v_2\bar{u}_2v_2\bar{u}_3v_2\ldots$$

Figure 5.9 gives an idea of how the pointers in Ψ look like. The important fact is that the first occurrence of v_1 in the suffix array points to some place into the area of $\bar{u}_1$, and for v_2 it points to the same position plus one. The same happens with $\bar{u}_2$ and $\bar{u}_3$, so if we calculate the *differences of* Ψ it would result in a repetition: The area of v_1 has the same values as the area of v_2 except for the borders, that is, the position that points to $\bar{u}_1$ and of course the position of v_1 that points to v_2.

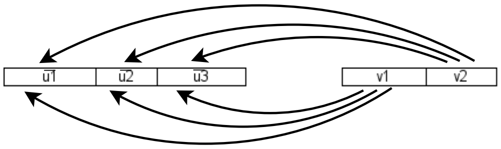


Figure 5.9: Example of Ψ

In order to exploit the regularities present in Ψ, we used Re-Pair on the differences of Ψ. It achieved 22.45 bits per edge for the UK crawl. This seems much more promising. Yet, there is an important fact to notice: If we decompress Ψ in the range corresponding to a node v, we have to pay one *rank* query for every element in the adjacency list (plus the cost of decompression of Ψ, see Figure 5.2). We also notice that this compression exploits the similarity between adjacency lists for a node and the similarities of the reverse lists, but we lose some of them in the differences of Ψ. Hence, this representation should not achieve better space than compressing the adjacency lists and their reverse adjacency lists separated using Re-Pair. In fact, there can be regularities in the differences of Ψ that are not related with those in adjacency lists, but those are essentially random coincidences. This fact motivates the work presented in Chapter 6, which compresses much better and has less penalty in time.

An example of the regularities lost by computing the differences of Ψ can be seen using Figure 5.9. Consider a new node u' between u_1 and u_2 that is pointed by v_2. This node inserts a new value in between the original two values. Now the differences for the pointers inside v_2 do not have the same value. Consider the adjacency lists of v_1 and v_2, $\{u_1, u_2, u_3\}$ and $\{u_1, u', u_2, u_3\}$: the symbols u_2 and u_3 will form a pair, while in this representation this does not happen. The same problem arises for the reverse graph.

5.3 Undirected Graphs

We have shown how to answer queries over a directed graph indexed like a text that is built by concatenating the encoded edges of the graph. An interesting extension of this idea is to apply this to undirected graphs. The normal way of representing every edge (u, v) is to add the edges (u, v) and (v, u) to the graph in order to answer adjacency list efficiently.

With the textual representation, we have shown that it is not necessary to represent both edges in an undirected graph, because the adjacency list of a node can be seen as the union between the adjacency list and its reverse list. As we have seen in the previous section, these two queries are answered almost the same way. So, the adjacency list is the union between the direct adjacency list and the reverse adjacency list and we just represent each edge once.

Chapter 6

Nodes Representation

This chapter presents two approaches to compressing the adjacency lists of Web graphs while providing random access. The first approach is based on Re-Pair and the second on Lempel-Ziv (see Section 2.4).

6.1 Re-Pair Compression of Web Graphs

Let $G = (V, E)$ be the graph we wish to compress and navigate. Let $V = \{v_1, v_2, \ldots, v_n\}$ be the set of nodes in arbitrary order, and $adj(v_i) = \{v_{i,1}, v_{i,2}, \ldots v_{i,a_i}\}$ the set of neighbors of node v_i. Finally, let $\overline{v_i}$ be an alternative identifier for node v_i. We represent G by the following sequence:

$$T \ = \ T(G) \ = \ \overline{v_1}\ v_{1,1}\ v_{1,2} \ldots v_{1,a_1}\ \overline{v_2}\ v_{2,1}\ v_{2,2} \ldots v_{2,a_2}\ \ldots\ \overline{v_n}\ v_{n,1}\ v_{n,2} \ldots v_{1,a_n}$$

so that $v_{i,j} < v_{i,j+1}$ for any $1 \leq i \leq n$, $1 \leq j < a_i$. This is essentially the concatenation of all the adjacency lists with separators that indicate the node each list belongs to. Figure 6.1 shows an example graph and its textual representation.

The application of Re-Pair to $T(G)$ has several important properties:

- Re-Pair permits fast local decompression, as it is a matter of extracting successive symbols from C (the compressed T) and expanding them using the dictionary of rules R. Moreover, Re-Pair handles well large alphabets, V in our case.

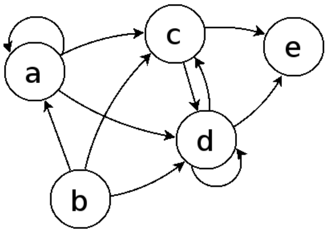


Figure 6.1: An example graph. The textual representation is $T(G) = \bar{a}acd\bar{b}acd\bar{c}de\bar{d}cde\bar{e}$.

- This works also very well if $T(G)$ must be anyway stored in secondary memory because the accesses to C are local and sequential, and moreover we access fewer disk blocks because it is a compressed version of T. This requires, however, that R (the set of rules) fits in main memory. This can be enforced at compression time, at the expense of losing some compression ratio, by preempting the compression algorithm when $|R|$ reaches the memory limit.

- As the symbols $\overline{v_i}$ are unique in T, they will not be replaced by Re-Pair. This guarantees that the beginning of the adjacency list of each v_i will start at a new symbol in C, so that we can decompress it in optimal time $O(|adj(v_j)|)$ without decompressing unnecessary symbols.

- If there are similar adjacency lists, Re-Pair will spot repeated pairs, therefore capturing them into shorter sequences in C. Actually, assume $adj(v_i) = adj(v_j)$. Then Re-Pair will end up creating a new symbol s which, through several rules, will expand to $adj(v_i) = adj(v_j)$. In C, the text around those nodes will read $\overline{v_i}s\overline{v_{i+1}} \ldots \overline{v_j}s\overline{v_{j+1}}$. Even if those symbols do not appear elsewhere in $T(G)$, the compression method for R [GN07] (Section 2.4.1) will represent R using $|adj(v_i)|$ numbers plus $1 + |adj(v_i)|$ bits. Therefore, in practice we are paying almost the same as if we referenced one adjacency list from the other. Thus we achieve, with a uniform technique, the result achieved by Boldi and Vigna [BV04] by explicit techniques such as looking for similar lists in an interval of nearby nodes.

- Even when the adjacency lists are not identical, Re-Pair can take partial advantage of their similarity. For example, if we have *abcde* and *abde*, Re-Pair can transform them to scs' and ss', respectively. Again, we obtain automatically what Boldi and Vigna [BV04] achieve by explicitly encoding the differences using gaps, bitmaps, and other tools.

- The locality property (i.e., the fact that most outgoing links from each page point within the same domain) is not exploited by Re-Pair, unless its translates into similar adjacency lists. This, however, makes our technique independent of the numbering. In Boldi and Vigna's work [BV04] it is essential to be able of renumbering the nodes according to site locality. Despite this is indeed a clever numbering for other reasons, it is possible that renumbering is forbidden if the technique is used inside another application. However, we show next a way to exploit locality.

The representation $T(G)$ we have described is useful for reasoning about the compression performance, but it does not give an efficient method to know where a list $adj(v_i)$ begins. For this sake, after compressing $T(G)$ with Re-Pair, we remove all the symbols $\overline{v_i}$ from the compressed sequence C (as explained, those symbols remain unaltered in C). Using essentially the same space we have gained with this removal, we create a table that, for each node v_i, stores a pointer to the beginning of the representation of $adj(v_i)$ in C. With it, we can obtain $adj(v_i)$ in optimal time for any v_i. Integers in C are stored using the minimum bits required to store the maximum value in C (see Chapter 4).

6.1.1 Improvements

We describe now several possible improvements over the basic scheme. Some can be combined, some not. Several possible combinations are explored in the experiments.

Differential encoding. If we are allowed to renumber the nodes, we can exploit the locality property in a subtle way. We let the nodes be ordered and numbered by their URL, and encode every adjacency list using differential encoding. The first value is absolute and the rest represents the difference to the previous value. For example the list 4 5 8 9 11 12 13 is encoded as 4 1 3 1 2 1 1.

Differential encoding is usually a previous step to represent small numbers with fewer bits. We do not want to do this as it hampers decoding speed. Our main idea to exploit differential encoding is that, if many nodes tend to have local links, there will be many small differences we could exploit with Re-Pair, say pairs like $(1, 1)$, $(1, 2)$, $(2, 1)$, etc. The price is slightly slower decompression.

Reordering lists. Since the adjacency list does not need to be output in any particular order, we can alter the original order to spot more global similarities[1]. Consider the lists $1, 2, 3, 4, 5$ and $1, 2, 4, 5$. Re-Pair can replace $1, 2$ by 6 and $4, 5$ by 7, but the common subsequence $1, 2, 4, 5$ cannot be fully exploited because the first list has a 3 in between. If we sort both adjacency lists after compressing we get $3, 6, 7$ and $6, 7$, and then we can replace $6, 7$, thus exploiting global regularities in both adjacency lists. The method is likely to improve compression ratios. The compression process is slightly slower: it works almost as in the original version, except that the lists are sorted after each pass of Re-Pair, so we cannot combine this method with differences. Decompression and traversal, on the other hand, are not affected at all. The experimental results show that this approach achieves better compression ratios than applying Re-Pair without differences. Note that this reordering is just a heuristic, and one could aim to finding the optimal ordering. However, similar problems have been studied for differential encoding of inverted lists, and they have been found to be hard [FV99, SCSC03].

Removing pointers. It might be advantageous, for relatively sparse graphs, to remove the need to spend a pointer for each node (to the beginning of its adjacency list in C). We can replace the pointers by two bitmaps. The first one, $B_1[1, n]$, marks in $B_1[i]$ whether node v_i has a non-empty adjacency list. The second bitmap, $B_2[1, c]$ (where $c = |C| \leq m$), marks the positions in C where adjacency lists begin. Hence the starting position of the list for node v_i in C is $select(B_2, rank(B_1, i))$ if $B_1[i] = 1$ (otherwise the list is empty). The list extends up to the next 1 in B_2. The space is $n + c + o(n + c)$ bits, instead of $n \log c$ needed by the pointers. When n is significant compared to c, space reduction is achieved at the expense of slower access to the adjacency lists.

6.2 Lempel-Ziv Compression of Web Graphs

The Lempel-Ziv compression family [ZL77, ZL78] achieves compression by replacing repeated sequences found in the text by a pointer to a previous occurrence thereof. In particular, the LZ78 variant [ZL78] stands as a plausible alternative candidate to Re-Pair for our goals: it detects duplicate lists of links in the adjacency lists, handles well large alphabets, and permits fast local decompression. Moreover, LZ78 admits efficient compression without requiring approximations (see Chapter 4).

[1] Thanks to Rodrigo Paredes for pointing out this idea during a “Miércoles de Algoritmos” meeting (`http://www.dcc.uchile.cl/gnavarro/algoritmos/`).

For a graph $G = (V, E)$, where $V = \{v_1, v_2, \ldots, v_n\}$ and $adj(v_i) = \{v_{i1}, v_{i2}, \ldots, v_{ia_i}\}$ is the set of neighbors of node v_i, the textual representation used for LZ78 compression is slightly different from that of Section 6.1:

$$T = T'(G) = v_{11}v_{12}v_{13}\ldots v_{1a_1}v_{21}v_{22}\ldots v_{2a_2}\ldots v_{n1}v_{n2}\ldots v_{na_n},$$

where we note that the special symbols $\overline{v_i}$ have been removed. The reason is that removing them later is not as easy as for Re-Pair. To ensure that adjacency lists span an integral number of phrases (and therefore can be extracted in optimal time $O(|adj(v_i)|)$), we run a variant of LZ78 compression. In this variant, when we look for the longest phrase $T_{i,j}$ in Step 2, we never cross a list boundary. More precisely, the character t_{j+1} to be appended to the new phrase must still belong to the current adjacency list. This might produce repeated phrases in the compressed text, which of course are not inserted into S.

Like C, the array of pointers and symbols added are stored using the minimum number of bits required by the largest pointer and symbol, respectively.

In addition, we store a pointer to every beginning of an adjacency list in the compressed sequence, just as for Re-Pair. Some of the improvements in Section 6.1.1 can be applied as well: differential encoding (which will have a huge impact with LZ78) and replacing pointers by bitmaps.

6.3 Experimental Results

We carried out several experiments to measure the compression and time performance of our graph compression techniques, comparing them to the state of the art.

We downloaded four Web crawls from the WebGraph project, `http://law.dsi.unimi.it/`. Table 6.1 shows their main characteristics. The last column shows the size required by a plain adjacency list representation using 4-byte integers.

6.3.1 Compression Performance

Our compression algorithm (see Section 2.4.1) is parameterized by M, k, and α. Those parameters yield a tradeoff between compression time and compression effectiveness. In this section, we study those tradeoffs. As there are several possible

Crawl	Nodes	Edges	Edges/Nodes	Plain size (MB)
`EU`	862,664	19,235,140	22.30	77
`Indochina`	7,414,866	194,109,311	26.18	769
`UK`	18,520,486	298,113,762	16.10	1,208
`Arabic`	22,744,080	639,999,458	28.14	2,528

Table 6.1: Some characteristics of the four crawls used in our experiments.

variants of our method, we stick in this section to the one called *Re-Pair Diffs CDict NoPtrs* in Section 6.3.3. The machine used in this section is a 2GHz Intel Xeon (8 cores) with 16 GB RAM and 580 GB Disk (SATA 7200rpm), running Ubuntu GNU/Linux with kernel 2.6.22-14 SMP (64 bits). The code was compiled with `g++` using the `-Wall`, `-O9` and `-m32` options. The space is measured in bits per edge (bpe), dividing the total space of the structure by the number of edges in the graph.

Parameter α (the maximum loading ratio of the hash table H before we stop inserting new pairs) turns out to be not too relevant, as its influence on the results is negligible for a wide range of reasonable choices. We set $\alpha = 0.6$ for all our experiments.

Value M is related to the amount of extra memory we require on top of T. Our first experiment aims at demonstrating that we obtain competitive results using very little extra memory. Table 6.2 shows the compression ratios achieved with different values of M (as a percentage over the size of T). As it can be seen, we gain little compression by using more than 5% over $|T|$, which is extremely modest (the linear-time exact Re-Pair algorithm [LM00] uses at the very least 200% extra space). The rest of our experiments are run using 3% extra space[2].

We now study the effect of parameter k in our time/quality compression tradeoff. Table 6.3 shows the time and compression ratio achieved for different k on our crawls. For the smaller crawls we also run the exact algorithm (using a relatively compact implementation [GN07] that requires 260MB total space for `EU` and 2.4GB for `Indochina`). It can be seen that our approximate method is able of getting very close to the exact result while achieving reasonable performance (around 1 MB/sec). Lempel-Ziv compression is much faster but compresses far less.

It is interesting to notice that, as k doubles, compression time is almost halved (especially for small k). This is related to the approximate analysis of our methods

[2] That is, in the beginning. As the text is shortened along the compression process we enlarge the hash table and keep using the absolute space originally allowed.

Graph	1%	3%	5%	10%	50%
EU	4.68	4.47	4.47	4.47	4.47
Indochina	2.53	2.53	2.53	2.52	2.52
UK	4.23	4.23	4.23	4.23	4.23
Arabic	3.16	3.16	3.16	3.16	3.16

Table 6.2: Compression ratios (in bpe) achieved when using different amounts of extra memory for H (measured in percentage over the size of the sequence to compress). In all cases we use $k = 10,000$.

(see Chapter 4), where we could not guarantee that all the k pairs chosen are actually replaced. Table 6.4 measures the number of replacements actually done by our algorithm on crawls EU and Indochina. As it can be seen, for k up to 10,000, more than 85% of the planned replacements are actually carried out, and this improves for larger graphs. Note also that the number of passes made by the algorithm is rather reasonable. This is relevant for secondary memory, as it means for example that with $k = 10,000$ we expect to do about 60 passes over the (progressively shrinking) text on disk for the EU crawl, and 263 for the Indochina crawl.

For the rest of the experiments we use $k = 10,000$.

6.3.2 Limiting the Dictionary

As explained, we can preempt Re-Pair compression at any pass in order to limit the size of the dictionary. This is especially interesting when the graph, even in compressed form, does not fit in main memory. In this case, we can take advantage of the locality of accesses to C to speed up the access to the graph: If we are able to compress $T(G)$ by a factor c, then access to long adjacency lists can be speeded up by a factor up to c. However, some Re-Pair structures need random access, and those must reside in RAM. This includes the dictionary, but also the structure that tells us where each adjacency list starts in C. The latter could still be kept on disk at the cost of one extra disk access per list, whereas the former definitely needs to lie in main memory.

Figure 6.2 shows the tradeoffs achieved between the size of the main sequence C and that of the RAM structures, as we modify the preemption point. It is interesting to notice that the main memory usage has a minimum, due to the fact that, as compression progresses, the dictionary grows but the width of the pointers

EU

k	time (min)	bpe
exact	86.15	4.40
10,000	1.77	4.47
25,000	1.03	4.70
50,000	0.83	4.74
75,000	0.72	4.76
100,000	0.73	4.79
250,000	0.62	4.91
500,000	0.62	4.95
1,000,000	0.67	4.95
LZ Diffs	0.07	7.38

Indochina

k	time (min)	bpe
exact	5,230.67	2.50
10,000	52.97	2.53
25,000	20.73	2.53
50,000	12.68	2.54
75,000	8.70	2.54
100,000	7.75	2.54
250,000	4.85	2.56
500,000	4.07	2.59
1,000,000	3.77	2.62
LZ Diffs	0.53	4.89

UK

k	time (min)	bpe
10,000	341.32	4.23
25,000	142.57	4.24
50,000	74.20	4.25
75,000	49.08	4.25
100,000	38.22	4.25
250,000	20.45	4.26
500,000	14.23	4.27
1,000,000	10.60	4.29
LZ Diffs	1.32	8.56

Arabic

k	time (min)	bpe
10,000	1,034.53	3.16
25,000	370.08	3.18
50,000	191.60	3.19
75,000	132.72	3.19
100,000	102.55	3.19
250,000	53.77	3.20
500,000	30.48	3.21
1,000,000	24.57	3.23
LZ Diffs	2.72	6.11

Table 6.3: Time for compressing different crawls with different k values. For the smaller graphs, we also include the exact method. We also include the results of our LZ variants for the four crawls. The LZ version was compiled without the `-m32` flag, since our implementation requires more than 4GB of RAM for the larger graphs.

EU

k	Passes	Total Pairs	Pairs/pass	% of k
5,000	108	497,297	4,604	92.08
10,000	58	502,530	8,664	86.64
20,000	33	513,792	15,569	77.85
50,000	19	543,417	28,600	57.20
100,000	14	576,706	41,193	41.19
500,000	12	676,594	56,382	11.28
1,000,000	12	676,594	56,382	5.64

Indochina

k	Passes	Total Pairs	Pairs/pass	% of k
10,000	263	2,502,880	9,516	95.16
20,000	136	2,502,845	18,403	92.02
50,000	60	2,503,509	41,725	83.45
100,000	34	2,528,530	74,368	74.37
500,000	16	2,772,091	173,255	34.65
1,000,000	14	2,994,149	213,867	21.39
5,000,000	14	3,240,351	231,453	4.63
10,000,000	14	3,240,351	231,453	2.31

Table 6.4: Number of pairs created by approximate Re-Pair over two crawls.

to C decreases[3].

At those optima, the overall size of C plus RAM data is not the best possible one, but rather close. In our graphs, the optimum space in RAM is from 0.2 to 0.4 bpe. This means, for example, that just 15MB of RAM is needed for our largest graph, `Arabic`. If we extrapolate to the 600GB graph of the *whole* static indexable Web, we get that we could handle it in secondary memory with a commodity desktop machine of 4GB to 8GB of RAM. If the compression would stay at about 6 bpe, this would mean that access to the compressed Web graph would be up to 5 times faster than in uncompressed form, on disk.

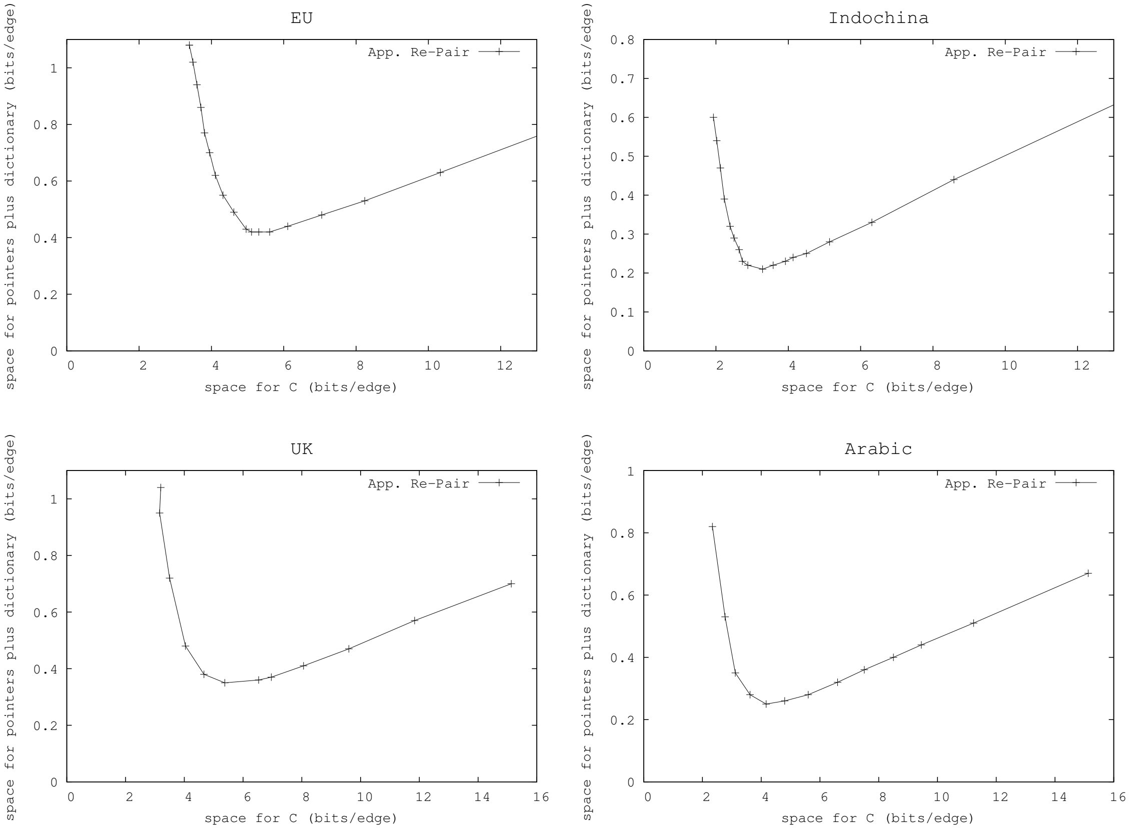


Figure 6.2: Space used by the sequence versus the dictionary plus the pointers, all measured in bits per edge.

[3] In the variant *NoPtrs* we use a bitmap of $|C|$ bits, which produces the same effect.

6.3.3 Compressed Graphs Size and Access Time

We now study the space versus access time tradeoffs of our graph compression proposals based on Re-Pair and LZ78. From all the possible combinations of improvements[4] depicted in Sections 6.1 and 6.2 we have chosen the following, which should be sufficient to illustrate what can be achieved (see in particular Section 6.1.1).

- *Re-Pair*: Normal Re-Pair.
- *Re-Pair Diffs*: Re-Pair with differential encoding.
- *Re-Pair Diffs NoPtrs*: Re-Pair with differential encoding and with pointers to C replaced by bitmaps.
- *Re-Pair Diffs CDict NoPtrs*: Re-Pair with differential encoding and a compacted dictionary. In the other implementations, every element of the dictionary is stored as an integer in order to speed up the access. This version stores every value using the required number of bits and not 32 by default. It also replaces the pointers to C by bitmaps.
- *Re-Pair Reord*: Normal Re-Pair with list reordering.
- *Re-Pair Reord CDict*: Re-Pair with list reordering and compacted dictionary.
- *LZ*: Normal LZ78.
- *LZ Diffs*: LZ78 on differential encoding.

For each of those variants, we measured the size needed by the structure versus the time required to access random adjacency lists. Structures that offer a space/time tradeoff will appear as a line in this plot, otherwise they will appear as points. The time is measured by extracting full adjacency lists and then computing the time per extracted element in $adj(v_i)$. More precisely, we generate a random permutation of all the nodes in the graph and sum the user time of recovering all of the adjacency lists (in random order). The time per edge is this total time divided by the number of edges in the graph.

These experiments were run on a Pentium IV 3.0 GHz with 4GB of RAM using Gentoo GNU/Linux with kernel 2.6.13 and `g++` with `-O9` and `-DNDEBUG` options.

[4] We can devise 16 combinations of Re-Pair and 8 combinations of LZ78 variants.

We compared to Boldi and Vigna's implementation [BV04] run on our machine with different space/time tradeoffs. The implementation of Boldi and Vigna gives a size measure that is consistent with the sizes of the generated files (and with their paper [BV04]). However, their process (in Java) needs significantly more memory to run. This could suggest that they actually use some structures that are not stored on the file, but built on the fly at loading time. Those should be accounted for in order to measure the size needed by the data structure to operate. Yet, this is difficult to quantify because of other space overheads that come from Java itself and from the WebGraph framework their code is inside.

To account for this, we draw a second line that shows the minimum amount of RAM needed for their process to run. In all cases, however, the times we show are obtained with the garbage collector disabled and sufficient RAM to let the process achieve maximum speed. Although our own code is in C++, the Java compiler achieves very competitive results[5].

We also show, in a second plot, a comparison of our variants with plain adjacency list representations. One representation, called "plain", uses 32-bit integers for nodes and pointers. A second one, called "compact", uses $\lceil \log_2 n \rceil$ bits for node identifiers and $\lceil \log_2 m \rceil$ for pointers to the adjacency list.

Figures 6.3 to 6.6 show the results for the four Web crawls. The different variants of LZ achieve the worst compression ratios (particularly without differences), but they are the fastest (albeit for a very little margin). The normal Re-Pair achieves a competitive result both in time and space. The other variants achieve different competitive space/time tradeoffs. The most space-efficient variant is *Re-Pair Diffs CDict NoPtrs*.

Node reordering usually achieves better compression without any time penalty, yet it cannot be combined with differential encoding.

A similar time/space tradeoff shown between *Re-Pair Diffs* and *Re-Pair Diffs NoPtrs* can be achieved with the other representations that use Re-Pair, since the pointers are the same for all of them. The time/space tradeoff between compacting the dictionary or not should be almost the same for the other Re-Pair implementations too.

The results show that our method is a very competitive alternative to Boldi and Vigna's technique, which is currently the best by a wide margin for Web graphs. In all cases, our method can achieve almost the same space (and less in some cases).

[5] See `http://www.idiom.com/∼zilla/Computer/javaCbenchmark.html` or `http://www.osnews.com/story/5602`.

Moreover, using the same amount of space, our method is always faster (usually 2–3 times faster, even considering their best line). In addition, some of our versions (those that do not use differential encoding) do not impose any particular node numbering.

Compared to an uncompressed graph representation, our method is also a very interesting alternative. It is 3–10 times smaller than the compact version and 2–4 times slower than it; and it is 5–13 times smaller than the plain version and 4–8 times slower.

6.4 Further Compression

The compressed sequence C is still stored with fixed-length integers. This is amenable of further compression: After applying Re-Pair, every pair of symbols in C is unique, yet individual symbols are not. Thus, zero-order compression could still reduce the space (probably at the expense of increasing the access time).

Yet, it is not immediate how to apply a zero-order compressor to such sequence, because its alphabet is very large. For example, applying Huffman would be impractical because of the need to store the table (i.e., at least the symbol permutation in decreasing frequency order). Instead, one could consider approximations such as Hu-Tucker's [HT71], which does not permute the symbols and thus needs only to store the tree shape. Hu-Tucker achieves less than 2 bits over the entropy.

To get a rough idea of what could be achieved, we estimated the space needed by Huffman and Hu-Tucker methods on our graphs, for the version *Re-Pair Diffs*. Let us call Σ the alphabet of C, and σ its size ($n \leq \sigma \leq n + |R|$), and say that n_i is the number of occurrences of the symbol i in C. A lower bound on the maximum size that Huffman can achieve is:

$$\textit{Huffman} \quad \geq \quad \sigma \log \sigma + \sum_{i \in \Sigma} n_i \log \frac{n}{n_i},$$

where we have optimistically bounded its output with the zero-order entropy and also assumed that the tree shape information is free (it is indeed almost free when using canonical Huffman codes, and the entropy estimation is at most 1 bit per symbol off, so the lower bound is rather tight).

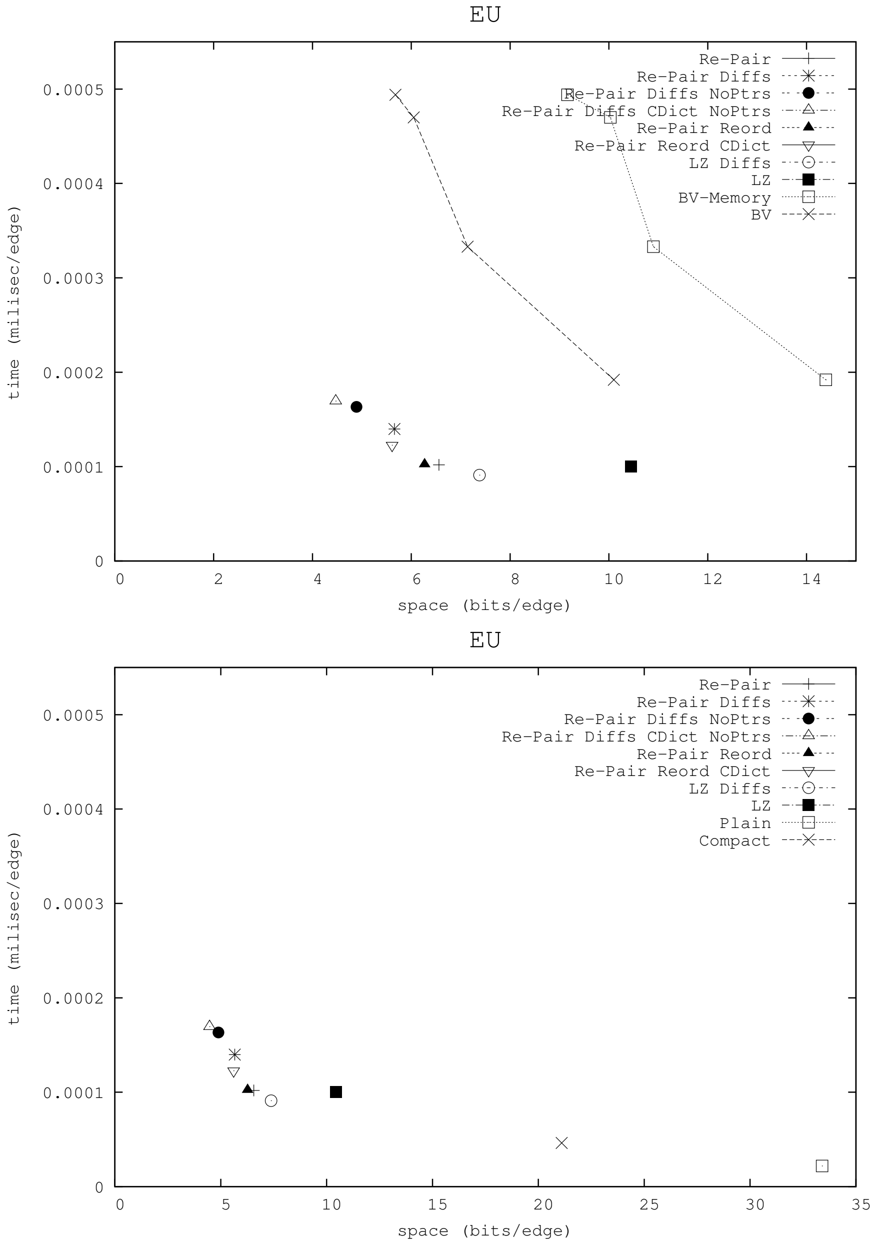


Figure 6.3: Space and time to find neighbors for different graph representations, over `EU` crawl. BV-Memory represents the minimum heap space needed by the process to run.

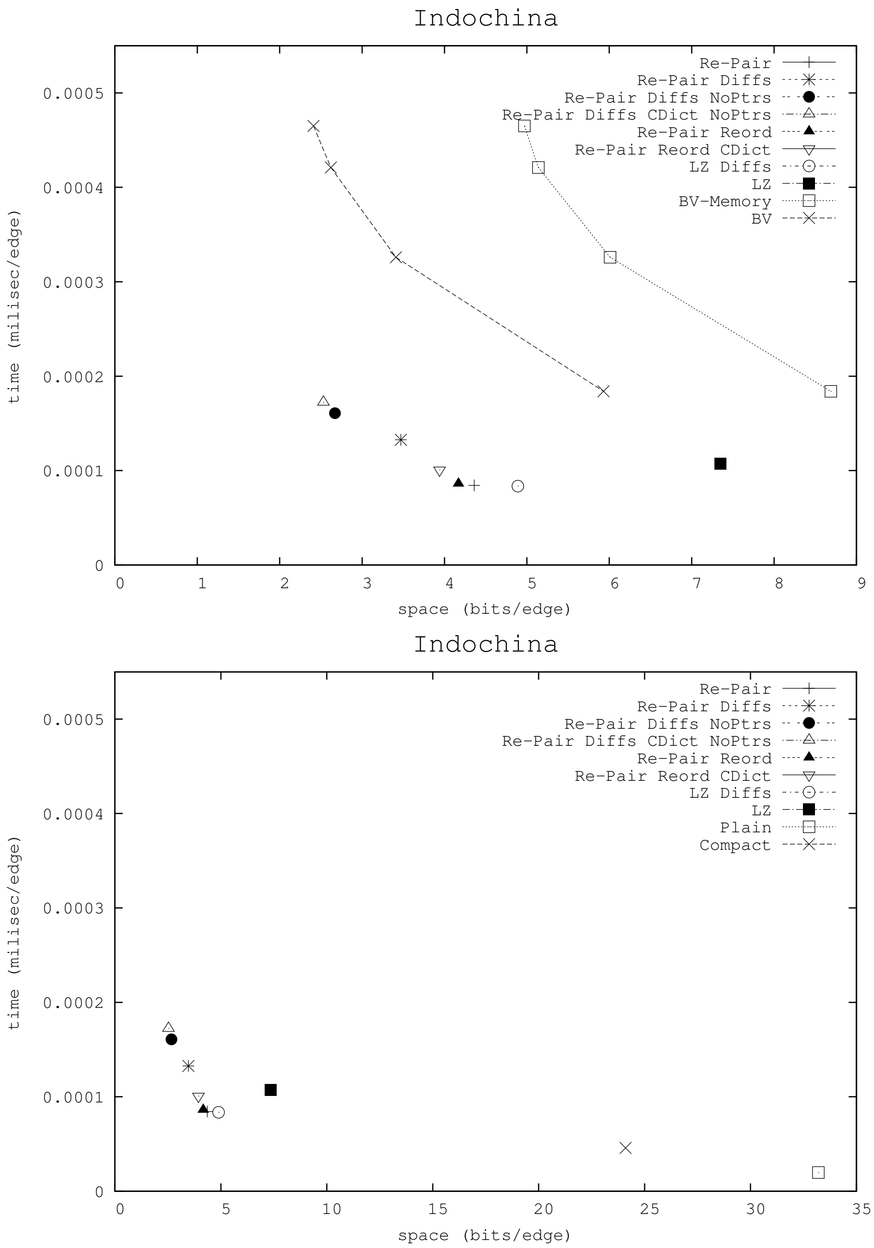


Figure 6.4: Space and time to find neighbors for different graph representations, over `Indochina` crawl. BV-Memory represents the minimum heap space needed by the process to run.

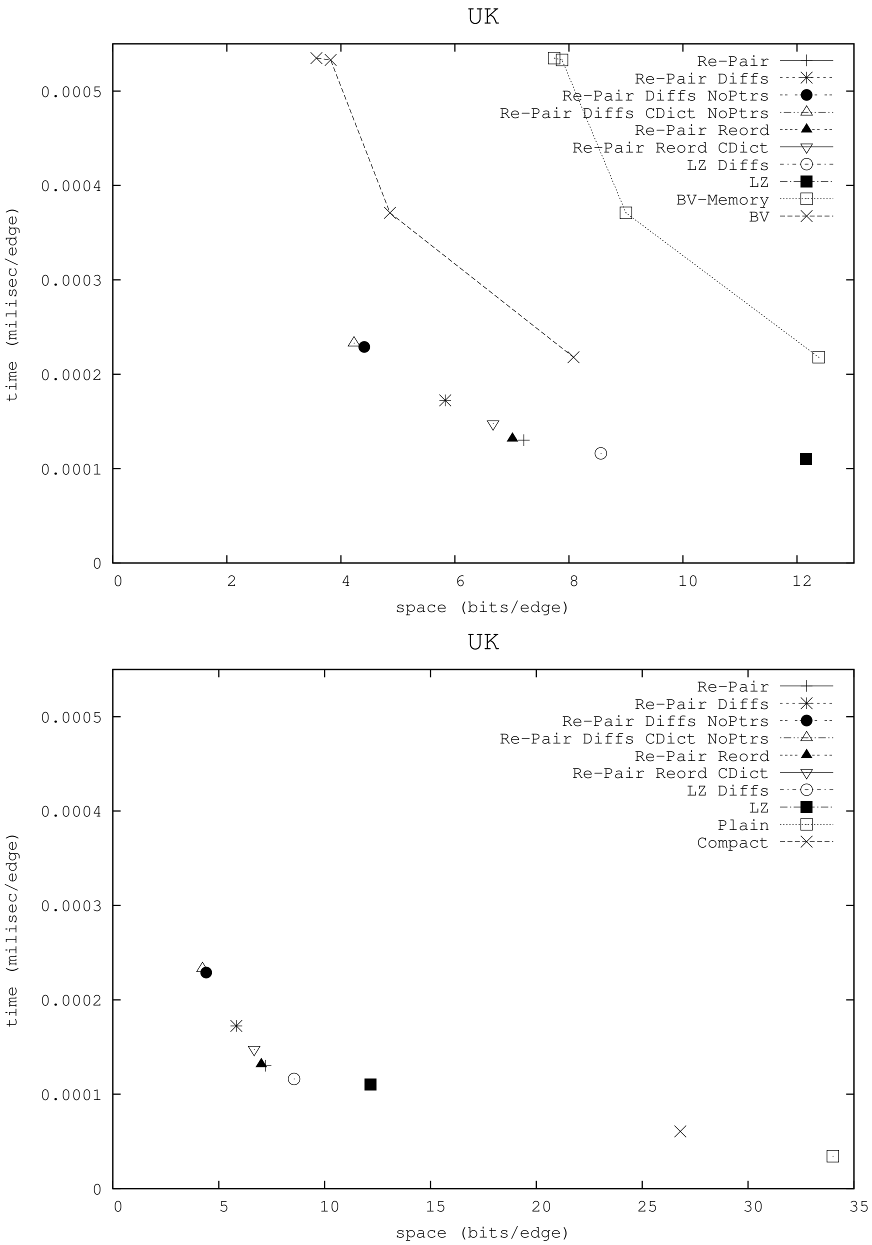


Figure 6.5: Space and time to find neighbors for different graph representations, over `UK` crawl. BV-Memory represents the minimum heap space needed by the process to run.

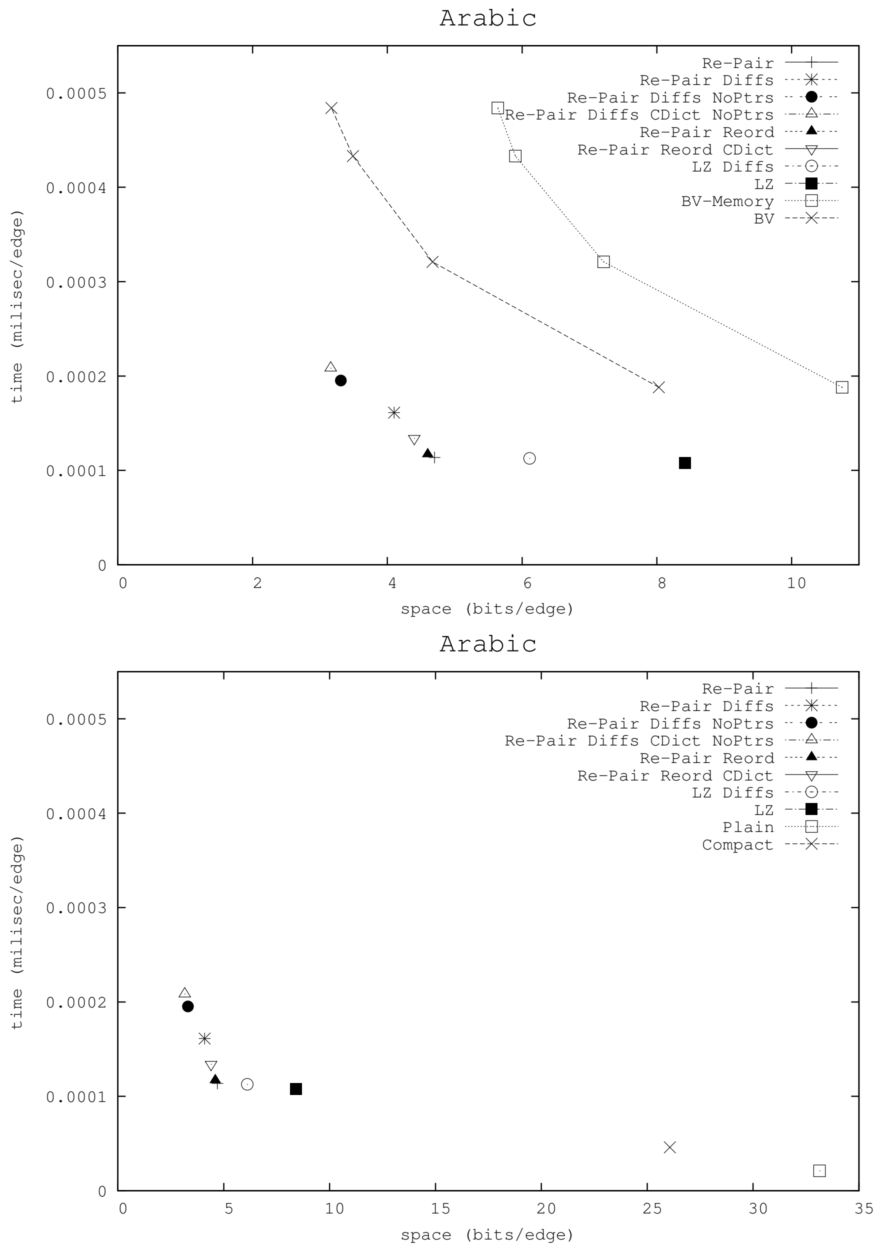


Figure 6.6: Space and time to find neighbors for different graph representations, over `Arabic` crawl. BV-Memory represents the minimum heap space needed by the process to run.

Since Hu-Tucker achieves more competitive results, we lower and upper bound its performance:

$$2\sigma + \sum_{i \in \Sigma} n_i \log \frac{n}{n_i} \quad \leq \quad HT \quad \leq \quad 2\sigma + \sum_{i \in \Sigma} n_i \left(\log \frac{n}{n_i} + 2 \right),$$

where the term 2σ arises because we have to represent an arbitrary binary tree of σ leaves, so the tree has $2\sigma - 1$ nodes and we need basically $2\sigma - 1$ bits to represent it (e.g., using 1 for internal nodes and 0 for leaves).

Table 6.5 shows the compression ratio bounds for C (i.e., not considering the other structures). As expected, Huffman compression is not promising, because just storing the symbol permutation offsets any possible gains. Yet, Hu-Tucker stands out as a promising alternative to achieve further compression. However, because of the bit-wise output of these zero-order compressors, the pointers to C must be wider[6]. Table 6.6 measures the size of the whole data structure with and without Hu-Tucker (we use the lower bound estimation for the latter). It can be seen that compression is not attractive at all, and in addition we will suffer from increased access time due to bit manipulations.

Graph	Huffman lower bound	Hu-Tucker lower bound	Hu-Tucker upper bound
EU	145.68%	84.65%	94.18%
Indochina	161.57%	82.11%	90.44%
UK	168.87%	82.94%	90.64%
Arabic	162.96%	82.81%	90.51%

Table 6.5: Compression ratio bounds for C, using *Re-Pair Diffs*. We measure the compressed C size as a percentage of the uncompressed C size.

An alternative, more sophisticated, approach to achieve zero-order entropy is to represent C using a wavelet tree where the bitmaps are compressed using the technique described in Section 2.5.2. This guarantees zero-order entropy (plus some sublinear terms for accessing the sequence), and it can take even less because each small block of around 16 entries of C is compressed to its own zero-order entropy. The sum of those zero-order entropies add up to at most the zero-order entropy of the whole sequence, but it can be significantly less if there are local biases of symbols (as it could perfectly be the case in Web graphs due to locality of reference).

[6] In the *NoPtrs* case this is worse, as we now need to spend one extra bit per *bit* of C, not per *number* in C.

Graph	Hu-Tucker (*Diffs NoPtrs*)	Hu-Tucker (*Diffs*)	Original
EU	6.61	4.89	4.47
Indochina	3.64	3.13	2.53
UK	6.14	5.33	4.23
Arabic	4.01	3.14	3.16

Table 6.6: Total space required by our original structures and the result after applying Hu-Tucker (lower-bound estimation).

Table 6.7 shows some results on the achievable space, using different sampling rates for the partial sums. We note that, because we can still refer to entry offsets (and not bit offsets) in C, our pointers to C do not need to change (nor the *NoPtrs* bitmap). We achieve impressive space reductions, to 70%–75% of the original space, and for Indochina we largely break the 2 bpe barrier.

In exchange, symbol extraction from C becomes rather slow. We measured the access time per link for the Arabic crawl using a sampling step of 32, and found that this approach is 22 times slower than our smallest (and slowest) version based on Re-Pair. For a sampling of 128 the slowdown is 43 times.

This can be alleviated by extracting all the symbols from an adjacency list at once, as no new *rank* operations are needed once we go through the same wavelet tree node again. In the worst case, we pay $O(k(1+\log \frac{\sigma}{k}))$ time, instead of $O(k \log \sigma)$, to extract k symbols. This improvement can only be applied when the symbols can be retrieved in any order, so it could not be combined with differences.

Our goal in this experiment was to show that it is still possible to achieve better results in terms of space, whereas more research is needed to couple those with attractive access times.

Graph	WT (8)	WT (32)	WT (128)	WT (∞)	Original
EU	4.59	3.71	3.49	3.42	4.47
Indochina	2.52	1.97	1.84	1.79	2.53
UK	4.36	3.40	3.16	3.08	4.23
Arabic	3.34	2.60	2.42	2.36	3.16

Table 6.7: Total space, measured in bpe, achieved when using compressed wavelet trees to represent C, with different sampling rates.

Chapter 7

Extending Functionality

Retrieving the list of neighbors is just the most basic graph traversal operation. One could explore other relevant operations to support within compressed space. An obvious one is to know the indegree/outdegree of a node. Those can be stored in plain form using $O(n \log m)$ bits, or using bitmaps: if we write in a bitmap a 1 for each new adjacency list and a 0 for each new element in the current adjacency list, one can compute outdegree as $select(i+1) - select(i)$. This bitmap requires $m + n + o(m+n)$ bits of space (i.e., little more than 1 bpe on typical Web graphs). This could be compressed to $O(n \log \frac{m+n}{n}) = O(m)$ bits (around 0.25 extra bpe in practice on typical Web graphs) and still support *select* in constant time. Indegree can be stored with a similar bitmap.

More ambitious is to consider *reverse neighbors* (list the nodes that point to v), which permits backward traversal in the graph (as in Chapter 5). One way to address this is to consider the graph as a *binary relation* on $V \times V$, and then use the techniques of Barbay et al. [BGMR06], where forward and reverse traversal operations can be solved in time $O(\log \log n)$ per node delivered. A more recent followup [BHMR07] retains those times and reduces the space to the zero-order entropy *of the binary relation*, that is, $\log \binom{m}{n}$.

This compression result is still poor for Web graphs, see Table 7.1 where we give lower bound estimations of the space that can be achieved (that is, we do not charge for any sublinear space terms on top of the binary relation data structures, which are significant in practice). The space for the binary relation is not much smaller than that of a plain adjacency list representation, although within that space it can solve reverse traversal queries. To achieve the same functionality we would need to store the original and the transposed graphs, hence we included the column "2 ×

Plain" in the table. We also included our best result based on Re-Pair (we add the space used by the directed and reverse graphs) [1].

Graph	Plain	2 × Plain	Bin.Rel.	Our Re-Pair
EU	20.81	41.62	15.25	7.65
Indochina	23.73	47.46	17.95	4.54
UK	25.89	51.78	20.13	7.50
Arabic	25.51	51.02	19.66	5.53

Table 7.1: Expected space usage (bpe) using the binary relation method, compared to other results.

7.1 A Simple and Complete Representation

It is interesting to note[2] that our textual representation $T(G)$ of Chapter 6, armed with symbol *rank* and *select* operations, is able of handling the extended set of queries:

- adjacency list of v_i: extract $T(G)$ between positions $select(\bar{v}_i, 1)$ and $select(\bar{v}_{i+1}, 1)$.
- outdegree of v_i: $select(\bar{v}_{i+1}, 1){-}select(\bar{v}_i, 1) - 1$.
- indegree of v_i: $rank(v_i, m + n)$.
- reverse adjacency list of v_i: using $select(v_i, k)$ we can retrieve the k-th occurrence of v_i in $T(G)$, and then using a bitmap B which marks the beginning of every adjacency list in $T(G)$ we can determine the lists of each such occurrence: $rank_B(select(v_i, k))$.

The main problem of this scheme is that $T(G)$ is essentially an uncompressed adjacency list representation, and the scheme does not compress it (or compresses it to its zero-order entropy [FMMN07], which is not good enough in this context). Our current research focus is to apply this same idea to the *compressed* representation C of $T(G)$ obtained from Re-Pair (see Chapter 6). This is particularly challenging for reverse neighbors, as the same symbol v_i might have been involved in the formation of many different new symbols, which must be found in the dictionary, and then all of them must be searched for in C.

[1] The transposed graph usually compresses better than the original in Web graphs [BV04].

[2] This is ongoing work with Paolo Ferragina and Rossano Venturini, University of Pisa.

7.2 Extended Functionality

We can regard our graph compression method as (and attribute its success to) the decomposition of the graph binary relation into two binary relations:

- Nodes are related to the Re-Pair symbols that conform their (compressed) adjacency list.
- Re-Pair symbols are related to the graph nodes they expand to.

Our result in Chapter 6 can be restated as: The graph binary relation can be efficiently decomposed into the product of the two relations above, achieving significant space gains. The regularities exposed by such a factorization go well beyond those captured by the zero-order entropy of the original binary relation. Now, representing these two binary relations with the technique of Barbay et al. [BGMR06] would yield a space comparable to our current solution, and $O(\log \log n)$ complexities to retrieve forward and reverse neighbors. Our next approaches build on this idea.

Our first proposal is based on the implementation of Re-Pair with reordering without pointers (see Section 6.1.1). We index C and the sequence of the dictionary, S (see Section 2.4.1), using one *chunk* of Golynski et al. (recall Section 2.5.2). The indegree and outdegree queries can be answered with the method presented in the beginning of this chapter, and the direct neighbors query is answered the same way as in Chapter 6, since the *chunk* structure allows *access*.

The main problem to find reverse neighbors of v is that v may appear in implicit form in many lists, in the form of a non-terminal that expands to v. Recall from Section 2.4.1 that the dictionary is represented as a sequence of symbols S and a bitmap BRR describing the trees. Thus we must look for v in the dictionary and, for each occurrence in the sequence S, collect all the ancestors and look for them in the text. The process has to be repeated recursively for every ancestor found.

Among the different possible solutions to find all the ancestors of an occurrence in S, we opted for a simple one: We mark the beginning of the top-level trees of BRR in another bitmap. Then we unroll the whole tree containing the occurrence and spot the ancestors.

Figure 7.1 shows the algorithm for retrieving the reverse neighbors. Note that no reverse neighbor can be reported twice.

```
**rev-adj**($v$)
**For** $k \leftarrow 1$ to $rank_C(v, |C|)$ **Do**
    $occ \leftarrow select_C(v, k)$
    report $rank_B(1, occ)$
**For** $k \leftarrow 1$ to $rank_S(v, |S|)$ **Do**
    $occ \leftarrow select_S(v, k)$
    **For each** $s$ ancestor of $S[occ]$ in $S$ **Do**
        rev-adj($s$)
```

Figure 7.1: Obtaining the reverse adjacency list

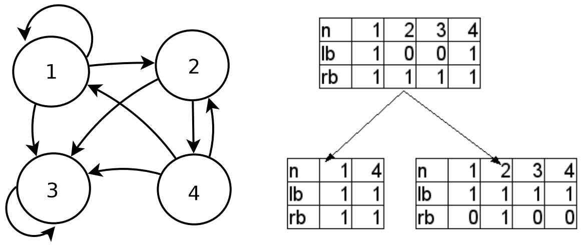
This representation can also be combined with the compressed version of the wavelet trees without pointers (see Section 3.1.2). This is much slower than the representation using Golynski et al.'s proposal (only one chunk), but it achieves better space.

7.3 Wavelet Trees for Binary Relations

Wavelet trees can be adapted to store binary relations (BRWT) while answering all the desired queries: in-degree, out-degree, direct neighbors and reverse neighbors. Since the order of the elements inside an adjacency list is not relevant, we can adapt wavelet trees to achieve better space than if we use the textual representation.

In this representation every node in the BRWT has two bitmaps. The first bitmap, *lb*, marks for every position whether the symbol is related to one or more symbols in the left side of the BRWT. The second bitmap, *rb*, marks for every position whether the symbol is related to at least one element in the right subtree. Figure 7.2 shows an example of this construction.

The space usage can be measured in terms of the entropy of the binary relation. Let n be the number of elements and m the number of related pairs. Consider a particular element i, related to other k_i elements. In the worst case, it would waste $2k_i \log n$ bits, 2 per level per pair. Yet, those k_i paths cannot be totally separated in the wavelet tree. At worst, they must share the first $\log k_i$ levels (occupying all the wavelet tree nodes up to there) and then each can split independently. So we have $2k_i$ nodes (i.e., $4k_i$ bits) up to the splitting level, and $2k_i(\log n - \log k_i)$ for the rest of the path to the leaves. If we sum up for all the elements we have



n	1	2	3	4
lb	1	0	0	1
rb	1	1	1	1

n	1	4
lb	1	1
rb	1	1

n	1	2	3	4
lb	1	1	1	1
rb	0	1	0	0

Figure 7.2: Example of BRWT for binary relations.

$$\sum_{i=1}^{n} 4k_i + 2k_i(\log n - \log k_i) = 4m + 2\sum_{i=1}^{n} k_i \log \frac{n}{k_i}$$

The worst case is $k_i = m/n$, yielding

$$4m + 2m \log \frac{n^2}{m} = 2H_0 + O(m)$$

being H_0 the zero-order entropy of the binary relation. This analysis can be extended to the case when we have a binary relation B over two sets V_1 of size n_1 and V_2 of size n_2, resulting in Theorem 7.3.1. Each bit requires $1 + o(1)$ space in order to support rank and select queries.

Theorem 7.3.1 *For a binary relation $\mathcal{B}$ over two sets V_1 of size n_1 and V_2 of size n_2, $\mathcal{B} \subseteq V_1 \times V_2$ with m related pairs, the BRWT requires $2m \log \frac{n_1 n_2}{m} + O(m) + o(H_0(\mathcal{B})) = 2H_0(\mathcal{B}) + O(m) + o\left(H_0(\mathcal{B})\right)$ bits of space.*

This is a worst-case analysis. The space can be much better if the binary representation of the elements to which an element is related share a common prefix. Call $p_1 \dots p_k$ the elements to which element i is related. We define $f(j)$ as the binary representation of p_j without the longest prefix it shares with the elements in $p_1 \dots p_{j-1}$. Then the space needed by the data structure to store this element can be expressed as $2f(j)$. Thus the final total space for element i can be calculated as $2\sum_{j=i}^{k_i} f(j)$. Note that defining $g(i)$ as the length of the binary representation without the prefix shared with any element in the list, the sum of all the $g(i)$s is a lower bound of the entropy defined for that binary relation, thus this function introduces a new measure which captures more regularities than the classical entropy.

The time to support the indegree and outdegree operators is $O(1)$, by adding two extra bitmaps of length $n + m$, iB and oB, using $O(m)$ bits each, as already explained. We add an extra bitmap oZ of length n, where we mark the nodes that have outdegree greater than zero. This also requires $O(m)$ bits.

The elements related to an element v can be retrieved by following the paths described by the ones in lb and rb, starting at the root in the position corresponding to v, until reaching the leaves (recall that we may have to go both left and right from some nodes). With rank queries over iB, we can determine the element represented by each node. The paths share some work during this process, the same way they share the space. If an element is related to k elements in the set, the worst case for retrieving its direct neighbors is $O\left(k + k \log \frac{n}{k}\right)$, assuming constant time rank/select queries on bitmaps.

The reverse list can be retrieved by traversing the path backwards. We start from the leaf corresponding to v and, for each element in the leaf, find its corresponding position in the root node. We start from the leaf corresponding to v, and use *select* of 1 over lb if we are coming from the left, and over rb if we are coming from the right. We use oZ to identify the node given the root position. In this case, we do not share any work and we have to pay the $\log n_2$ levels for every element retrieved. The time for reverse neighbors is $O(k \log n)$, assuming constant time rank/select queries on bitmaps.

A better alternative represents lb and rb together as a sequence drawn from an alphabet of size 3, $\Sigma = \{s_{01}, s_{10}, s_{11}\}$. If we assume that all elements are related to some other elements, we can compress these sequences and the resulting space is at most $(\log 3)H_0(\mathcal{B}) + O(m) + o(H_0(\mathcal{B}))$ instead of $2H_0(\mathcal{B}) + O(m) + o(H_0(\mathcal{B}))$. In practice we need only 2 (not 3) rank directories: for the sum of $s_{01} + s_{11}$ (to go left) and for $s_{01} + s_{11}$ (to go right). The case when an element is not related to other elements in the set can be fixed with the bitmap used for outdegree.

As before, the pointers can be omitted, but this is harder than for the original wavelet trees. Assume we represent the BRWT using $2\lceil \log n_2 \rceil$ bitmaps, one per level. Given a range that represents a node, we can compute the position where the left child starts in the next level by counting how many elements are before in the next level. That corresponds to *rank* in that level until the starting position of the current node (for both lb and rb). The right child has to add the number of elements going to the left child from the current node.

We can emulate this process using only one bitmap. We have to consider how to know the range covering every level. That can be done by adding the number of ones in lb and rb in the prior level, since this corresponds to the number of elements in the next level.

7.3.1 Dynamic Representation (dBRWT)

Hon et al. [HSS03] proposed a bitmap representation that, for a bitmap B of length n, supports rank/select in $O(\log n/\log\log n)$ time and insert/delete in $O(\text{polylog}(n))$, requiring $n + o(n)$ bits. This result was later improved by Mäkinen and Navarro [MN06], achieving $nH_0(B) + o(n)$ bits of space and supporting the four operations in $O(\log n)$ time. If we use these representations for the bitmaps in the BRWT, it is possible to insert and delete new pairs to/from the relation.

In order to insert a new pair $v \rightarrow u$, we start at the position corresponding to element v and follow the path to leaf u. At every level, if we find a 1 in the direction described by u, we do not modify anything in the node. Otherwise, we set the bit to 1, and for the rest of the path we insert two bits. If the path goes left, we insert a 1 in lb and a 0 in rb, otherwise we insert a 1 in rb and a 0 in lb.

The delete operation is similar. We have to traverse upwards the path described by u, and delete the corresponding bits in lb and rb only at the levels where the only remaining symbol contained is u.

For every operation that modifies the binary relation, we have to update the bitmaps for indegree and outdegree. This does not affect the asymptotic time.

Notice that at every level we need a constant number of rank/select queries, and at most two insert/delete queries. Every level has at most m bits, and assuming $n_1 \geq n_2$, we have $\log m = O(\log n_1)$.

It is also possible to use a dynamic representation for sequences. The solution proposed by González and Navarro [GN08] requires $nH_0(S) + o(n)\log\sigma$ bits of space for a sequence S of length n drawn from an alphabet of size σ. The queries *rank, select* and *access* are supported in $O\left(\log n\left(1 + \frac{\log\sigma}{\log\log n}\right)\right)$ time. In our case $\sigma = 3$ and we can perform the operations the same way as for the bitmaps, thus the space is at most $(\log 3)H_0(\mathcal{B}) + O(m) + o(H_0(\mathcal{B}))$ bits.

Theorem 7.3.2 gives a trade-off between the representations using one or $\lceil\log n_2\rceil$ bitmaps.

Theorem 7.3.2 *Consider a binary relation $\mathcal{B}$ over two sets, V_1 of size n_1 and V_2 of size n_2, $n_1 \geq n_2$. Assume there exists a data structure for representing a bitmap B of length n, that supports rank/select in time $f(n)$, insert/delete in time $g(n)$, and requires $n + o(n)$ bits.*

Then, the dBRWT that uses $\log n_2$ bitmaps requires $2H_0(\mathcal{B}) + O(m) + \log n_2 \log n_1 n_2 + o(H_0(\mathcal{B})) = 2H_0(\mathcal{B}) + O(m) + o\left(H_0(\mathcal{B})\right)$ bits of space and supports (k represents the size of the output in every query):

Operation	*Cost*
Direct neighbors	$O\left(f(m)\left(k + k\log\frac{n_2}{k}\right)\right)$
Reverse neighbors	$O(f(m)k\log n_2)$
In/outdegree	$O(f(m))$
Insert/delete (pair)	$O(g(m)\log n_2)$

For the case we use only one bitmap to represent the dBRWT, the space required is $2H_0(\mathcal{B}) + O(m) + o\left(H_0(\mathcal{B})\right)$ *bits and supports:*

Operation	*Cost*
Direct neighbors	$O\left(f\left(4m + 2m\log\frac{n_1 n_2}{m}\right)\left(k + k\log\frac{n_2}{k}\right)\right)$
Reverse neighbors	$O\left(f\left(4m + 2m\log\frac{n_1 n_2}{m}\right)k\log n_2\right)$
In/outdegree	$O(f(m))$
Insert/delete (pair)	$O\left(g\left(4m + 2m\log\frac{n_1 n_2}{m}\right)\log n_2\right)$

Corollary 7.3.3 *By applying Theorem 7.3.2 to the case when we represent lb and rb in a joint sequence using the representation proposed by González and Navarro [GN08], the constant multiplying the entropy for binary relations drops to* $\log 3$ *and* $f(n) = g(n) = O(\log n)$*.*

By updating our bitmaps, we can add new elements to V_1 without modifications to the wavelet tree structure: We only need to add a 1 at the corresponding position of the bitmaps oB and iB, and a 0 at the same position of oZ. We can also delete an element v from V_1 in time $O(g(n)(1 + k + \bar{k} + k\log n_2))$, where k is the number of elements related to v and $\bar{k}$ is the number of element to which v is related. The same cannot be done so easily for V_2 without major changes to the wavelet tree structure.

7.4 Experimental Results

The experiments were run on a 2GHz Intel Xeon (8 cores) with 16 GB RAM and 580 GB Disk (SATA 7200rpm), running Ubuntu GNU/Linux with kernel 2.6.22-14 SMP (64 bits). The code was compiled with `g++` using the `-Wall`, `-O9` and `-m32` options. For the wavelet tree built over the plain representation of the graph, we omitted the `-m32` directive.

Table 7.2 gives the results of representing the Web graphs without applying Re-Pair, but rather using compressed wavelet-tree-based representations of the plain adjacency lists. In the first column, we directly use a wavelet tree over the sequence, representing bitmaps with RRR and omitting pointers (see Section 3.1.2). In the second column, we represent the graph as a binary relation using BRWT (see Section 7.3). In the last column, we give the entropy of the binary relation for each crawl. The results show that the methods are interesting for general binary relations (in paticular better than the global entropy of the binary relation), but we show next that Re-Pair performs much better on Web graphs.

Table 7.3 shows the space required for the four crawls using Re-Pair based compression. In column one (Re-Pair+WT), we include the structure given in Section 7.2, which supports reverse queries based on the wavelet tree representation of the Re-Pair compressed sequence. In column two, we show the representation that combines Re-Pair with Golynski et al.'s *chunk* (Re-Pair+Golynski). We include in column three the best structure proposed in Chapter 6 (adding direct and reverse graphs). As a baseline for comparison, we show in the last column the binary relation entropy of these graphs.

Comparing Table 7.3 with Table 7.2, we have that the Re-Pair-induced decomposition into the product of two relations is key to the success of our approach. We also tried compressing the binary relations after Re-Pair with the BRWT, but the space achieved is worse than our simple implementation with reoridering (e.g., it spends 0.59 extra bpe in `EU`, 0.24 on `Indochina` and 0.49 on `UK`).

Crawl	Wavelet Tree	BRWT	Bin. Rel. Entropy
`EU`	13.67	10.31	15.25
`Indochina`	6.26	14.16	17.95
`UK`	15.05	8.23	20.13
`Arabic`	15.30	8.40	19.66

Table 7.2: Size required by simple (and complete) compressed representations of the plain adjacency lists (measured in bpe).

Figure 7.3 shows retrieval times obtained for the four crawls using the representation that combines Re-Pair with Golynski et al.'s *chunk* (Re-Pair+Golynski). We compare it against the best structure proposed in Chapter 6 (adding up the direct and reverse graphs). Re-Pair+Golynski is not as fast as the structure presented in Chapter 6, but it requires much less space when supporting reverse queries. We also include a second version of Re-Pair+Golynski for `EU` and `Indochina` which indexes the reverse graph in order to achieve better space. The results of this second version are not good since the reverse graph compresses better,

Crawl	Re-Pair + WT (samp 64)	Re-Pair + Golynski	Re-Pair (direct+reverse)	Bin. Rel. Entropy
EU	3.93	5.86	7.65	15.25
Indochina	2.30	3.65	4.54	17.95
UK	3.98	6.22	7.50	20.13
Arabic	2.72	4.15	5.53	19.66

Table 7.3: Space consumption (in bpe) of the Re-Pair based compressed representations of the adjacency lists.

generating more symbols in the dictionary, and thus the reverse queries (direct adjacency list for the original graph, `Re-Pair Golynski Direct v2`) become much slower. On the other hand, the direct queries (reverse adjacency list for the original graph, `Re-Pair Golynski Reverse v2`) are slightly faster than the direct queries in Re-Pair+Golynski. When comparing the same type of queries, the second version is always slower. Other alternatives considered in Tables 7.2 and 7.3 are much slower as well (see Section 6.4).

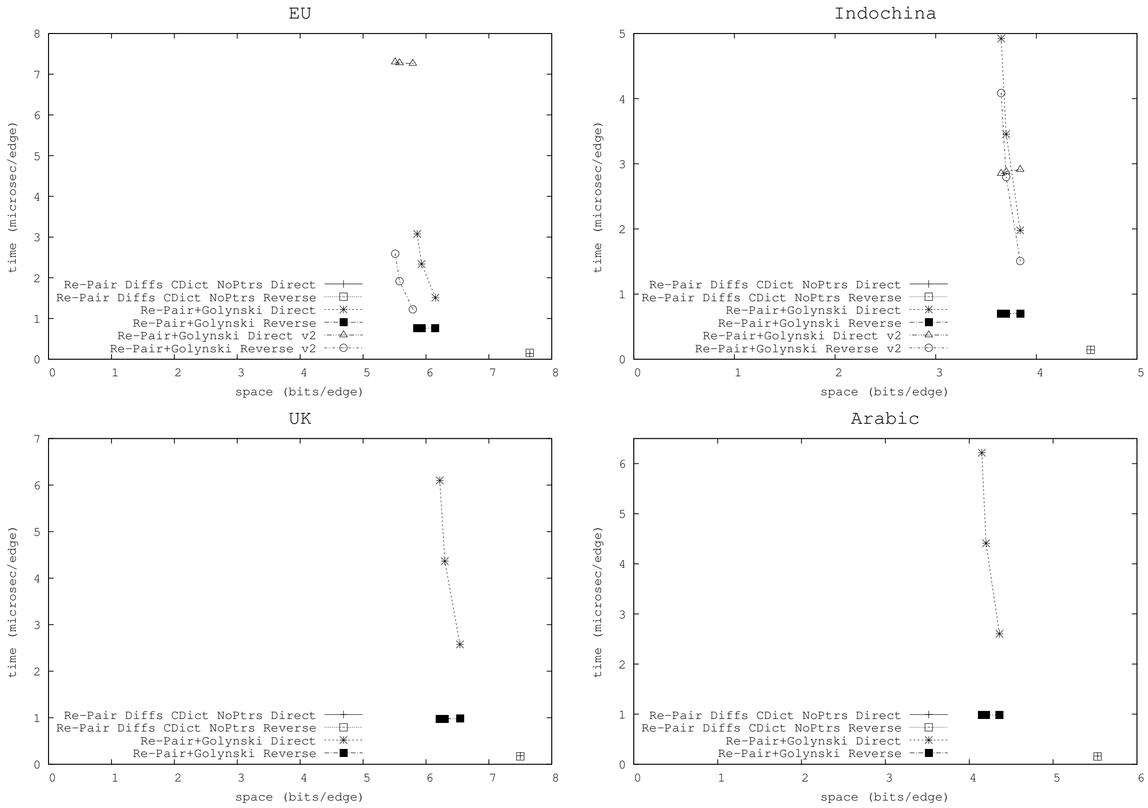


Figure 7.3: Experimental results of the proposed structure that combines Golynski et al.'s *chunk* with Re-Pair. It is compared against our best variant representing the graph and its reverse. Direct and Reverse refer to the times to retrieve each neighbor and reverse neighbor, respectively.

Chapter 8

Conclusions

We have presented a graph compression method that exploits the similarities between adjacency lists by using grammar-based compressors such as Re-Pair [LM00] and LZ78 [ZL78]. Our results demonstrate that those similarities account for most of the compressibility of Web graphs, on which our technique performs particularly well. Our experiments over different Web crawls demonstrate that our method achieves compression ratios very close to (sometimes slightly better than) those of the best current schemes [BV04], while being 2–3 times faster to navigate the compressed graph. Compared to a plain adjacency list representation, our compressed graphs can be 5 to 13 times smaller, at the price of a 4- to 8-fold traversal slowdown (this has to be compared to the hundred to thousand times slowdown caused by running on secondary memory). This makes it a very attractive choice to maintain graphs all the time in compressed form, without the need of a full decompression in order to access them. As a result, graph algorithms that are designed for main memory can be run over much larger graphs, by maintaining them in compressed form. In case the graphs do not fit in main memory, even in compressed form, our scheme adapts well to secondary memory, where it can take fewer accesses to disk than its uncompressed counterpart for navigation.

An interesting example is the `Arabic` crawl. It needs 2.4 GB of RAM with a plain representation, whereas our compressed version requires only 250 MB of RAM. This can be easily manipulated in a normal 1 GB machine, whereas the plain version would have to resort to disk. If we extrapolate to the 600GB graph of the *whole* static indexable Web, we get that we could handle it in secondary memory with a commodity desktop machine of 4GB to 8GB of RAM. If the compression would stay at about 6 bpe, this would mean that access to the compressed Web graph would be up to 5 times faster than in uncompressed form, on disk.

Our technique is not particularly tailored to Web graphs (beside trying to exploit similarities in adjacency lists). This could make it suitable to compress other types of graphs, whereas other approaches which are too tailored to Web graphs could fail. To us, this is a beautiful example where a general and elegant technique can compete successfully with carefully ad-hoc designed schemes.

We also presented several practical implementations for the *rank* and *select* problem in general sequences. We introduced a new variant of the wavelet tree, and presented the first known implementation of Golynski et al.'s proposal [GMR06] for *rank* and *select* over large alphabets. This experimental comparison is a useful resource for deciding which structure is suitable depending on the application.

The results obtained by applying the different structures for *rank* and *select* on full-text self-indexing show that a simple modification of the SSA index can achieve space proportional to nH_k in practice, thus validating the theoretical proposal of [MN07]. We also demonstrate that this index allows us to achieve the best space ever seen in self-indexes, sometimes without any time penalty.

We also included an experimental comparison of Re-Pair, LZ78 and our approximate version of the Re-Pair algorithm. The results showed that dictionary based methods achieve acceptable space, while supporting local decompression with very good performance. We developed an efficient approximate version of Re-Pair, which can work within very limited space and also works well on secondary memory. This can be of interest given the large amount of memory required by the exact Re-Pair compression algorithm.

In the extension of the solution based on Re-Pair (see Chapter 7), we proposed a solution to queries like indegree, outdegree, and reverse adjacency list. The reverse queries are based on a decomposition of the binary relation, and despite that our decomposition is tailored to Re-Pair compression, we believe that the perspective of achieving compressible decompositions of binary relations can be a very interesting research track on its own.

We also proposed a new data structure for representing binary relations. This structure uses space proportional to the entropy of the binary relation and can be dynamized, supporting insertion and deletion of new pairs to/from the relation.

An interesting problem related to graph compression is the definition of entropy for binary relations. The space achieved by our structures, and those proposed before, achieve much better space than the entropy for these graphs seen as binary relations. A new measure for this problem has not been found and would be of great interest, since the actual parallel, H_k for text, cannot be applied in this case, at least not for our proposals, since the alphabet size is too big. The result proved by Gagie [Gag06] shows that we cannot achieve compression better than H_2 for the adjacency lists (seeing the concatenation of them as a text).

This works pointed out interesting lines of research for the future. We first include some medium-term goals:

- Implementing a practical solution for RRR which includes run-length compression. This would improve the space achieved by the modified SSA, since the bitmap of the BWT usually has many runs.
- Measuring the space achieved by the representation based on the LZ78 algorithm when we replace the sequence by compressed wavelet trees.
- Extending the representation based on LZ78 to support reverse queries.
- A new representation based on Re-Pair with differences for the graph and its reverse that shares a common dictionary[1].
- Implement the alternative version of Golynski et al.'s structure for rank and select over large alphabets using the inverse permutation as π. This favors access over rank and select operations.
- Use this alternative for the graph representation proposed in Chapter 7. This would favor direct over reverse queries.

Some long-term goals are:

- Faster compressed representation of sequences supporting access, which would improve the compression in Chapter 6.
- Proposing a general factorization method for binary relations.
- Finding a better measure of entropy for binary relations, a definition like H_k for text.

[1] This idea came up during a conversation with Susana Ladra.

Bibliography

[ACL00] W. Aiello, F. Chung, and L. Lu. A random graph model for massive graphs. In *Proc. 32th ACM Symposium on Theory of Computing (STOC)*, pages 171–180, 2000.

[AM01] M. Adler and M. Mitzenmacher. Towards compressing Web graphs. In *Proc. 11th Data Compression Conference (DCC)*, pages 203–212, 2001.

[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. 7th World Wide Web Conference (WWW)*, pages 469–477, 1998.

[BBK03] D. Blandford, G. Blelloch, and I. Kash. Compact representations of separable graphs. In *Proc. 14th Symposium on Discrete Algorithms (SODA)*, pages 579–588, 2003.

[BBYRNZ01] C. Badue, R. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. In *Proc. 8th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 10–20, 2001.

[BFLN08] N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, 2008. To appear.

[BGMR06] J. Barbay, A. Golynski, I. Munro, and S. Srinivasa Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. In *Proc. 17th Symposium on Combinatorial Pattern Matching (CPM)*, 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. 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 680–689, 2007.

[BKM$^+$00] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the Web. *Journal of Computer Networks*, 33(1–6):309–320, 2000.

[Bla06] D. Blandford. *Compact data structures with fast queries.* PhD thesis, School of Computer Science, Carnegie Mellon University, 2006. Also as Tech. Report CMU-CS-05-196.

[BV04] P. Boldi and S. Vigna. The WebGraph framework I: compression techniques. In *Proc. 13th World Wide Web Conference (WWW)*, pages 595–602, 2004.

[BW94] M. Burrows and D. Wheeler. A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, 1994.

[CGH$^+$98] R. Chuang, A. Garg, X. He, M.-Y. Kao, and H.-I. Lu. Compact encodings of planar graphs with canonical orderings and multiple parentheses. In *LNCS v. 1443*, pages 118–129, 1998.

[Cla96] D. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, 1996.

[CN07] F. Claude and G. Navarro. A fast and compact Web graph representation. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 105–116, 2007.

[CPMF04] D. Chakrabarti, S. Papadimitriou, D. Modha, and C. Faloutsos. Fully automatic cross-associations. In *Proc. ACM Special Interest Group on Knowledge Discovery and Data Mining (SIGKDD)*, pages 79–88, 2004.

[DL98] N. Deo and B. Litow. A structural approach to graph compression. In *Proc. of the 23th MFCS Workshop on Communications*, pages 91–101, 1998.

[FGNV07] P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice! Manuscript. `http://pizzachili.dcc.uchile.cl`, 2007.

[FM00] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proc. 41st Annual Symposium on Foundations of Computer Science (FOCS)*, pages 390–398, Washington, DC, USA, 2000. IEEE Computer Society.

[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, 2007.

[FV99] A. Fink and S. Voß. Applications of modern heuristic search methods to pattern sequencing problems. *Computers & Operations Research*, 26:17–34, 1999.

[Gag06] T. Gagie. Large alphabets and incompressibility. *Information Processing Letters*, 99(6):246–251, 2006.

[GGMN05] R. González, Sz. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. 4th International Workshop on Efficient and Experimental Algorithms (WEA)*, pages 27–38, 2005. Posters.

[GGV03] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[GMR06] A. Golynski, I. Munro, and S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 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. 18th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4580, pages 216–227, 2007.

[GN08] R. González and G. Navarro. Improved dynamic rank-select entropy-bound structures. In *Proc. 8th Latin American Symposium on Theoretical Informatics (LATIN)*, LNCS 4957, pages 374–386, 2008.

[GS05] A. Gulli and A. Signorini. The indexable Web is more than 11.5 billion pages. In *Proc. 14th World Wide Web Conference (WWW)*, pages 902–903, 2005.

[GV00] R. Grossi and J. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proc. 32nd ACM Symposium on Theory of Computing (STOC)*, pages 397–406, 2000.

[HKL99] X. He, M.-Y. Kao, and H.-I. Lu. Linear-time succinct encodings of planar graphs via canonical orderings. *Journal on Discrete Mathematics*, 12(3):317–325, 1999.

[HKL00] X. He, M.-Y. Kao, and H.-I. Lu. A fast general methodology for information-theoretically optimal encodings of graphs. *SIAM Journal on Computing*, 30:838–846, 2000.

[HSS03] W. Hon, K. Sadakane, and W. Sung. Succinct data structures for searchable partial sums. In *14th Annual International Symposium on Algorithms and Computation (ISAAC)*, pages 505–516, 2003.

[HT71] T. Hu and A. Tucker. Optimal computer-search trees and variable-length alphabetic codes. *SIAM Journal of Applied Mathematics*, 21:514–532, 1971.

[Huf52] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the I.R.E.*, 40(9):1090–1101, 1952.

[IR82] A. Itai and M. Rodeh. Representation of graphs. *Acta Informatica*, 17:215–219, 1982.

[Jac89] G. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie Mellon University, 1989.

[KM99] R. Kosaraju and G. Manzini. Compression of low entropy strings with Lempel-Ziv algorithms. *SIAM Journal on Computing*, 29(3):893–911, 1999.

[Knu98] D. E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, April 1998.

[KRRT99] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. Extracting large scale knowledge bases from the Web. In *Proc. 25th Conference on Very Large Data Bases (VLDB)*, pages 639–650, 1999.

[KW95] K. Keeler and J. Westbook. Short encodings of planar graphs and maps. *Discrete Applied Mathematics*, 58:239–252, 1995.

[LM00] J. Larsson and A. Moffat. Off-line dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, 2000.

[Lu02] H.-I. Lu. Linear-time compression of bounded-genus graphs into information-theoretically optimal number of bits. In *Proc. 13th Symposium on Discrete Algorithms (SODA)*, pages 223–224, 2002.

[Man01] G. Manzini. An analysis of the Burrows-Wheeler transform. *Journal of the ACM*, 48(3):407–430, 2001.

[MM93] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal of Computing*, 22:935–948, 1993.

[MN05] V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. *Nordic Journal of Computing*, 12(1):40–66, 2005.

[MN06] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. In *Proc. 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, LNCS 4009, pages 307–318, 2006.

[MN07] V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proc. 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 4726, pages 214–226, 2007.

[MR97] I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. 38th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 118–126, 1997.

[MRRR03] J. Munro, R. Raman, V. Raman, and S. Rao. Succinct representations of permutations. In *Proc. 30th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS 2719, pages 345–356, 2003.

[Nao90] M. Naor. Succinct representation of general unlabeled graphs. *Discrete Applied Mathematics*, 28(303–307), 1990.

[Nav07] G. Navarro. Compressing web graphs like texts. Technical Report TR/DCC-2007-2, Dept. of Computer Science, University of Chile, 2007.

[NM07] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):article 2, 2007.

[NR08] G. Navarro and L. Russo. Re-pair achieves high-order entropy. In *Proc. 18th Data Compression Conference (DCC)*, page 537, 2008. Poster.

[OS07] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[RGM03] S. Raghavan and H. Garcia-Molina. Representing Web graphs. In *Proc. 19th International Conference on Data Engineering (ICDE)*, page 405, 2003.

[Ros99] J. Rossignac. Edgebreaker: Connectivity compression for triangle meshes. *IEEE Transactions on Visualization*, 5(1):47–61, 1999.

[RRR01] R. Raman, V. Raman, and S. Rao. Succinct dynamic data structures. In *Proc. 7th Workshop on Algorithms and Data Structures (WADS)*, LNCS 2125, pages 426–437, 2001.

[RRR02] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th Annual ACM-SIAM 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.

[Sad03] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[SCSC03] W. Shieh, T. Chen, J. Shann, and C. Chung. Inverted file compression through document identifier reassignment. *Information Processing & Management*, 39(1):117–131, 2003.

[SY01] T. Suel and J. Yuan. Compressing the graph structure of the Web. In *Proc. 11th Data Compression Conference (DCC)*, pages 213–222, 2001.

[TGM93] A. Tomasic and H. Garcia-Molina. Performance of inverted indices in shared-nothing distributed text document information retrieval systems. In *Proc. 2nd International Conference on Parallel and Distributed Information Systems (PDIS)*, pages 8–17, 1993.

[Tur84] G. Turán. Succinct representations of graphs. *Discrete Applied Mathematics*, 8:289–294, 1984.

[Wan03] R. Wan. *Browsing and Searching Compressed Documents*. PhD thesis, Dept. of Computer Science and Software Engineering, University of Melbourne, 2003. `http://eprints.unimelb.edu.au/archive/00000871`.

[Wil83] D. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters*, 17(2):81–84, 1983.

[WMB99] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, New York, second edition, 1999.

[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 length coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.