Efficient compression of large repetitive strings

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy

Christopher Hoobin B.Eng (Hons.), B.App.Sci (Hons.),
School of Computer Science and Information Technology,
College of Science, Engineering, and Health,
RMIT University,
Melbourne, Victoria, Australia.

October, 2015

Declaration

I certify that except where due acknowledgement has been made, the work is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program; any editorial work, paid or unpaid, carried out by a third party is acknowledged; and, ethics procedures and guidelines have been followed.

Christopher Hoobin
School of Computer Science and Information Technology
RMIT University
October, 2015

Credits

Portions of the material in this thesis have previously appeared in the following publications:

- C. Hoobin, S. J. Puglisi, and J. Zobel. Sample selection for dictionary-based corpus compression. In *Proceedings of the 34th ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, 1137–1138, 2011.
- C. Hoobin, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. In *Proceedings of the VLDB Endowment (PVLDB)*, 5(3):265–273, 2011.
- T. Gagie, C. Hoobin, and S. J. Puglisi. Block graphs in practice. In *Proceedings of the 2nd International Conference on Algorithms for Big Data (ICABD)*, 30–36, 2014.
- C. Hoobin, T. Kind, C. Boucher, and S. J. Puglisi. Fast and efficient compression of high-throughput sequencing reads. In *Proceedings of the 6th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics (BCB)*, 352–334, 2015.

This work was supported by the Australian Research Council and the Information Storage, Analysis and Retrieval Group (ISAR) at RMIT University.

The thesis was typeset using the LaTeX 2$_\varepsilon$ document preparation system.

Contents

List of Figures

List of Tables

Abstract

When is comes to managing large volumes of data, general-purpose compressors such as gzip are ubiquitous. They are fast, practical and available on every modern platform from standard desktops to mobile devices. These tools exploit local redundancy in a text using a fixed-size sliding window. This window is usually very small relative to the text, however, in principle it can be as large as available memory. The window acts as a dictionary. Compression is achieved by replacing substrings with pointers to previous occurrences found in the dictionary. This type of algorithm becomes problematic when dealing with collections that are larger than physical memory, as it fails to capture any non-local redundancy, that is, repetition that occurs outside of its search window. With rapid growth in the already enormous amount of data we store and process there is a pressing need for improving compression effectiveness, reducing both storage requirements and decompression costs. However, many systems still use general-purpose compression tools on large highly repetitive data collections.

In this thesis we focus on addressing this issue. We explore compression in a variety of domains where large volumes of data need to be stored and accessed, and general-purpose compression tools are cannon. First we discuss our work on web corpus compression, then we discuss the implementation of a practical index for repetitive texts that gives strong theoretical bounds in terms of size and access, and finally, we discuss our work on compression of high-throughput sequencing reads. We show that in all cases, our new methods improve on current techniques in both run-time and compression effectiveness, and provide important functionality such as fast decoding and random access.

CHAPTER 1

Introduction

We are witness to massive growth in the size of digital archives across all domains, from natural language texts, to archived web crawls and biological databases. This rapid growth presents unique challenges in many areas of computer science from data storage and maintenance to efficient indexing, search and retrieval. Compression plays a crucial role providing many benefits [Ziviani et al. 2000]. A compressed text takes less space, reducing the cost of storage, transmission, and improving bandwidth across all levels of the memory hierarchy. There are many desirable properties in a compression algorithm: the amount of compression it achieves, the time and space requirements for compression and decompression, and the ability to provide random access, that is, being able to decode from arbitrary positions in a compressed text. This leads to a variety of applications such as, document retrieval, query biased snippet generation [Tsegay et al. 2009], and pattern matching directly in compressed text [Manber 1997, de Moura et al. 2000].

We measure compression effectiveness by a *compression ratio*, which is the size of a compressed file as a fraction of the original text size. In most cases we consider efficient compression and decompression speed to be mutually exclusive. For example, an Information Retrieval system will compress a collection once and access it many times. Here, more of an emphasis is placed on efficient decoding and random access to its compressed text. As the collection is only compressed once concessions can be made during encoding, for example, using slower or more memory-hungry approaches to improve compression. On the other hand, if a collection is to be archived it is not crucial that either compression or decompression is fast, only that it is compressed as efficiently as possible.

There exists a wealth of literature providing solutions for efficient storage and retrieval of text collections [Bell et al. 1990, Witten et al. 1999, Baeza-Yates and Ribeiro-Neto 1999]. A classical approach for compression of natural language texts is to use a semi-static word-based model [Moffat 1989, Horspool and Cormack 1992, Zobel and Moffat 1995b, Ziviani et al. 2000] combined with a bit-oriented Huffman code [Huffman 1952] in which words are assigned codes based on their probability distribution. de Moura et al. [2000] and Brisaboa et al. [2007a] extend this idea to use byte-oriented codes, observing a minimal effect on compression size and a significant improvement in decompression speed. The main disadvantage of this model is that a mapping of symbol to code-word must be maintained during both compression and decompression. This is a significant drawback when compressing larger collections as the mapping dominates the rest of the encoding. Moffat et al. [1997] observe that a mapping grows almost linearly in the size of the text due to the inclusion of spelling mistakes, new acronyms, and junk text. A further drawback is that the definition of a symbol or word becomes problematic for non-English texts, for example, Chinese texts, where sentences are written without explicit word boundaries.

A practical alternative it to use an LZ77-based algorithm. Presented by Ziv and Lempel [1977] in their seminal paper dating back almost 40 years, LZ77 has spawned a large family of algorithms offering a variety of trade-offs during compression and decompression [Salomon 2004]. An LZ-style algorithm provides reasonable compression, fast decoding, and forms the base of many popular general purpose compression tools such as gzip, zip, 7zip, and xz, which in one form or another can be found on almost every computing device from desktops and servers to mobile devices. Conventional LZ-style compression methods exploit *local* redundancy in a text by encoding their input relative to a sliding window of previously encoded substrings. This window is usually small or at least bound by physical memory, and, as a consequence it does not accurately capture any *global* redundancy present in collection. This becomes problematic when compressing specific collections, such as sets of whole genomes or DNA sequencing reads, which are known to be highly repetitive but where redundancy is usually non-local [Mäkinen et al. 2010, Deorowicz et al. 2013]. A further disadvantage is that traditional LZ methods do not directly support random access. Decoding must always start from the beginning of a compressed file. There are a variety of solutions for this problem, such as adding synchronization points to the encoding [Witten et al. 1999] or partitioning a collection into fixed sized blocks and compressing them separately [Ferragina and Manzini 2010], however, these tricks lead to an undesirable trade-off between compression effectiveness and decoding speed.

A self-index is a data structure that represents a text and provides efficient random access and pattern matching in space close to that of the compressed text. Because of this, the index can actually replace the text [Navarro and Mäkinen 2007]. Such indexes are usually based on compressed suffix arrays (CSA) [Sadakane 2003], or the Burrows Wheeler Transform (BWT) of a text [Ferragina and Manzini 2005], and there are a number of efficient implementations that work well in practice [Ferragina et al. 2009]. Navarro [2004] present an LZ78-based self index [Ziv and Lempel 1978], however, it is was shown to be not effective for compressing large repetitive collections. Mäkinen et al. [2010] note the suitability of LZ77 for compression of highly repetitive collections and comment that

"LZ77 has defied for years its adaptation to a self-index form. Thus, there is a wide margin of opportunity for such a development." The first practical LZ77-based index was presented by Kreft and Navarro [2010]. They describe LZ-End, an LZ77-style parsing technique and accompanying data structure that achieves compression similar to LZ77-based compressors and supports fast random access. To construct a self-index, a suffix array [Manber 1997] and other uncompressed data structures are usually required to be built on a text. This limits the utility of self-indexes to text collections that fit in available memory, as the suffix array can require space up to eight times the size of a text [Puglisi et al. 2007].

In recent work, Ferragina et al. [2012] describe a disk-based method for computing the BWT of a text directly without the need of a suffix array, however, constructing an index from the BWT still assumes that it is resident in memory, as the BWT is a permutation of a text.

Grammar-based compression is another group of algorithms that provides efficient compression of highly repetitive collections. Here, a text is replaced with a small context-free grammar (CFG) which is later used to rebuild the text. A grammar can represent a text that is exponentially larger than itself. Generating an optimal grammar for a text is considered impractical [Charikar et al. 2005], however, most grammar-based compressors compute an approximation taking various heuristic approaches. Some examples of grammar-based compressors are LZ78 [Ziv and Lempel 1978], Sequitur [Nevill-Manning et al. 1994], XRay [Cannane and Williams 2000], and Repair [Larsson and Moffat 1999]. This approach is capable of identifying and exploiting global redundancy throughout a text, however, like self-indexes, grammar compressors are hindered by large memory requirements during construction. Maruyama et al. [2012; 2013] give an online grammar-based compressor, FOLCA, that works in relatively small space, however, it still assumes that a text can fit in memory.

As we have seen, many existing approaches to text compression have inherent limitations when dealing with collections that are significantly larger than physical memory. For example, most existing or off-the-shelf LZ-based compressors can not exploit non-local duplication in a text due to the limited size of their dictionary, while algorithmic approaches such as a self-index are bound by memory constraints, particularly at construction time. General-purpose compressors such as gzip are used everywhere, especially when distributing large text collections. Such wide adoption is primarily due to accessibility and ease of use. These tools provide a reasonable trade-off between compression effectiveness and efficient decoding speed. With such rapid growth in digital archives and known limitations of existing practical approaches there is a increasing need to investigate and improve the efficiency and effectiveness of such tools.

This is the primary motivation behind the research and contributions in this thesis. We present algorithms and data-structures for compression of large text collections that are capable of identifying and exploiting non-local redundancy and also provide important functionality such as efficient decoding and fast random access.

1.1 Key Contributions

Next, we provide a summary of the main contributions of this thesis.

Compression of large scale web collections. Archived web crawls contain a very high level of redundancy, for example, many pages will contain the same boilerplate markup or shared scripts. There could be mirrored sites fetched from different domains or shared news articles. Over the years we have observed a dramatic increase in the size and content of web collections from our own academic resources such as GOV2[1] and Clueweb,[2] to public archives such as the Internet Archive[3] and the Common Crawl.[4]

In Chapter 3 we describe a compression scheme that builds a representative sample of a collection. This sample is then used as a dictionary in an LZ-like encoding of the whole collection. First, we outline a novel dictionary generation technique that successfully captures non-local redundancy throughout a collection. We show that the dictionary can be as small as 0.1% of the overall collection size, which can easily fit in physical memory, and still provide effective compression and fast random access. We describe a number of coding techniques that offer various trade-offs during compression and decoding. We then empirically evaluate our scheme by simulating a document retrieval system and demonstrate that it provides superior compression and significantly faster decoding throughput and random access than current state-of-the-art baselines. Additionally, we show that compression is still effective in a dynamic environment, that is, where a collection is regularly updated with new documents.

Although our compression scheme is highly effective we noticed that a large percentage of the dictionary was unused during encoding, almost 30% on average. Moreover, there was a strong skew in the samples that were used, and, even among these, there was redundancy as some samples contained repeated material. In Chapter 4 we explore this issue and describe two techniques to eliminate redundancy throughout a dictionary. The first algorithm is to be used before compression in a pre-processing stage, where long repeated substrings are identified and removed from the dictionary. Then, we describe a more principled approach where we compute usage statistics of a dictionary during an encoding, then eliminate unused and redundant content to create a new compact dictionary. This is used to re-encode the collection, repeating this process until compression degrades or we reach a desired dictionary size or acceptable level of redundancy. We show

[1]http://ir.dcs.gla.ac.uk/test_collections/gov2-summary.htm
[2]http://lemurproject.org/clueweb09/
[3]https://archive.org/
[4]http://commoncrawl.org/

that both algorithms successfully remove redundancy throughout the dictionary with a minimal effect on compression. We observed that we can reduce the dictionary to half its size and achieve almost identical compression. Furthermore, we can generate a finely tuned 100 MB dictionary that provides superior compression than all practical baselines on a 426 GB document collection. These methods give us explicit control of memory during run-time which can be useful in restricted environments, for example, a virtual machine instance or a lightweight device. Furthermore, this space saving can be used to append more useful content to a dictionary in order to improve compression.

A practical compressed index with fast random access. Many compressed indexes are designed for efficient pattern matching operations such as returning the number of occurrences or locations of a pattern in a text. Most indexes do provide random access, however, they are not as efficient in practice. LZ-End by Kreft and Navarro [2010] achieves exactly the opposite. It was designed primarily for efficient random access and performs well in practice, however, it has poor worst-case bounds for compression and random access time. Grammar-based indexes by Rytter [2003] and Bille et al. [2011] give stronger theoretical bounds, however, they are not practical, as they contain unwieldy constant factors.

In Chapter 5 we give a practical implementation of the block graph data structure by Gagie et al. [2011], an LZ-style compressed index that supports efficient random access. We show block graphs to be competitive in both theory and practice. First, we give an overview of the data structure detailing how to traverse the index and extract random substrings. Then, we outline a practical implementation of a block graph and describe in detail how to navigate and represent each component of a block graph compactly. We show that on several standard repetitive collections our implementation provides better compression and faster random access than LZ-End and is competitive in space with general-purpose compressors.

Compression of high-throughput sequencing reads. Advances in high-throughput sequencing technology have dramatically reduced the time and cost for an individual sequencing experiment, acting as a catalyst for a massive growth in genomic databanks, such as the NCBI Nucleotide database[5] and the Sequence Read Archive (SRA),[6] which are expected to double in size every ten months for the next decade [Cochrane et al. 2013]. With a number of ambitious projects on the horizon, such as the UK-10K project[7] and those outlined by the International Cancer Genome Sequencing Consortium [Hudson et al. 2010], both aiming to sequence tens of thousands of genomes are placing a significant burden on databanks in the near future. Such an increase in growth of biological

[5]http://www.ncbi.nlm.nih.gov/nucleotide
[6]http://www.ncbi.nlm.nih.gov/Traces/sra/
[7]http://www.uk10k.org/

data is creating many unique and costly challenges, from maintenance and storage to the development of algorithms that can scale to such massive volumes of data.

Efficient compression of biological data requires methods that are quite different to that of natural language texts in order to exploit redundancy throughout a collection, and popular general-purpose compressors struggle to improve over a naive static codes. In Chapter 6 we present two novel algorithms for compression of large real-world high-throughput sequencing read collections. First, we present Faust, a scan-based LZ-style algorithm capable of scaling to large real-world sequencing experiments. We introduce an efficient coding scheme to represent a read in terms of another previously seen similar read. Then, we present Afin, a second stage compression algorithm which performs a reordering of a Faust encoding to further exploit the high levels of redundancy throughout a collection. We empirically evaluate both algorithms against current state-of-the-art methods and general-purpose compression tools on a collection of reads from a large real-world sequencing experiment and find that both new methods perform efficiently in practice. We show that Faust is competitive in compression performance to BEETL a state-of-the-art read compression scheme by Cox et al. [2012], encoding in half the time and providing significantly faster decoding, 17 minutes compared to 40 hours on average. Then, we demonstrate that Afin achieves competitive compression and decoding compared to a number of current state-of-the-art baselines.

1.2 Thesis Structure

The remainder of this thesis is structured as follows.

Chapter 2 gives an overview of the related concepts covered throughout this thesis, from information theory to text compression, text indexing and operations on succinct data structures.

Chapter 3 presents an efficient compression scheme for large repetitive text collections. Our experiments focus on web crawls, however, in future chapters, we demonstrate that it provides practical compression on a variety of real-world text collections.

Chapter 4 outlines approaches for redundancy elimination in sampled dictionaries described in the previous chapter. We demonstrate that we can effectively remove redundant and unused components from a dictionary, reducing its size significantly with little to no effect on compression compared to original approaches.

Chapter 5 introduces the first practical implementation of a block graph data structure proposed by Gagie et al. [2011] which is specifically designed for compression of large repetitive collections. We demonstrate that it is competitive in both theory and practice, providing competitive compression and superior random access capability.

Chapter 6 presents two novel algorithms for compression of high-throughput sequencing read data. We demonstrate that these algorithms are capable of compressing large real-world collections, providing superior compression and decoding speeds compared to state-of-the-art approaches.

Chapter 7 concludes and outlines directions for future work.

CHAPTER 2

Background

In this chapter we give an overview of text compression methods describing fundamental algorithms, data structures and associated work in the context of the contributions in this thesis. We begin in Section 2.1 by defining standard symbols and notation to represent properties of strings. In Section 2.2 we discuss text indexing and its applications, from pattern matching to text compression. We detail two classic text indexes, the suffix tree and the suffix array, then we briefly give an overview of succinct data structures and compressed full-text indexes. Finally, in Section 2.3 we discuss text compression, focusing on adaptive dictionary-based modeling, which is a fundamental building block for the work of this thesis.

2.1 Preliminaries

A string, S, of length n, is defined as a finite sequence of symbols (or characters) derived from an alphabet Σ of size σ such that, $S[1..n] = S[1]S[2]..S[n-1]S[n]$. The empty string, of length 0, is denoted ε. The alphabet defines the set of unique symbols that can occur in a string. Throughout this thesis we will be using a number of fixed alphabets, specifically, a binary alphabet $\Sigma = \{0, 1\}$, DNA nucleotides, $\Sigma = \{a, c, g, t\}$, and integer alphabets, $\Sigma = \{1, 2, .., \sigma - 1, \sigma\}$, such as extended ASCII, $\Sigma = \{0, 1, .., 255\}$.

Interval notation is used to identify specific characters and substrings of S. A square bracket denotes a closed interval, such that it includes the characters at each end point, and a curved bracket is used to exclude them. For example, given the string, $S =$ sassafras, $n = 9$, $\Sigma = \{a, f, r, s\}$ and $\sigma = 4$. $S[5]$ corresponds to the character a at index 5. $S[7..9]$, corresponds to the substring ras, while $S[7..9)$ corresponds to ra. The ith prefix of S is written as $S[1..i]$, such that, $S[1..i] = S[1]S[2]..S[i]$. Likewise, the jth suffix of S is written as $S[j..n]$ with length $n - j + 1$, such that, $S[j..n] = S[j]S[j + 1]..S[n]$. The first four prefixes of S are s, sa, sas and sass. The first four suffixes of S are sassafras, assafras, ssafras and safras. All logarithms are of base 2 unless otherwise specified.

2.2 Text Indexing

Pattern matching plays a fundamental role in computer science with many practical applications such as the search functionality in a text editor and shell tools, for example, awk and grep. Many of the core algorithmic problems in fields such as Information Retrieval and Bioinformatics boil down to pattern matching problems. The definition of the exact pattern matching problem consists of locating one or more occurrences of a pattern, P of length m, in a string S of length n. In general, the string is usually much larger than the pattern, that is, $m \ll n$. When n is small there are a number of efficient online solutions that can be used. These run linear in the size of the text, such as the classic textbook algorithms by Knuth et al. [1977] and Boyer and Moore [1977]. Both algorithms efficiently scan the text from the beginning to the end reporting pattern occurrences. When n is sufficiently large and a text is to be searched many times, scanning is no longer practical. In such cases it is useful to construct an index on the text offline, such as a suffix tree or suffix array in order to speed up the pattern matching process. Such indexes have been described as having myriad virtues [Apostolico 1985], providing exact pattern matching capabilities in time proportional to m, and giving solutions to a wide range of other string processing problems from approximate pattern matching, where errors are allowed in the pattern and/or string, to text compression. We refer the reader to Gusfield [1997] for further reading.

The fundamental operations supported by an index data structure are count, locate and extract.

count(P) returns the number of occurrences of a pattern P in a text T.

locate(P) returns the positions of each occurrence of a pattern P in a text T.

extract(i, j) returns the substring $T[i..j]$.

This functionality is used as a building block for more advanced operations such as approximate pattern matching.

Next we discuss two fundamental data structures for text indexing, the suffix tree and the suffix array. Both play an important role as the basis for many compression algorithms and compressed data structures [Navarro and Mäkinen 2007]. Furthermore, we make extensive use of the suffix array throughout the body of work in this thesis.

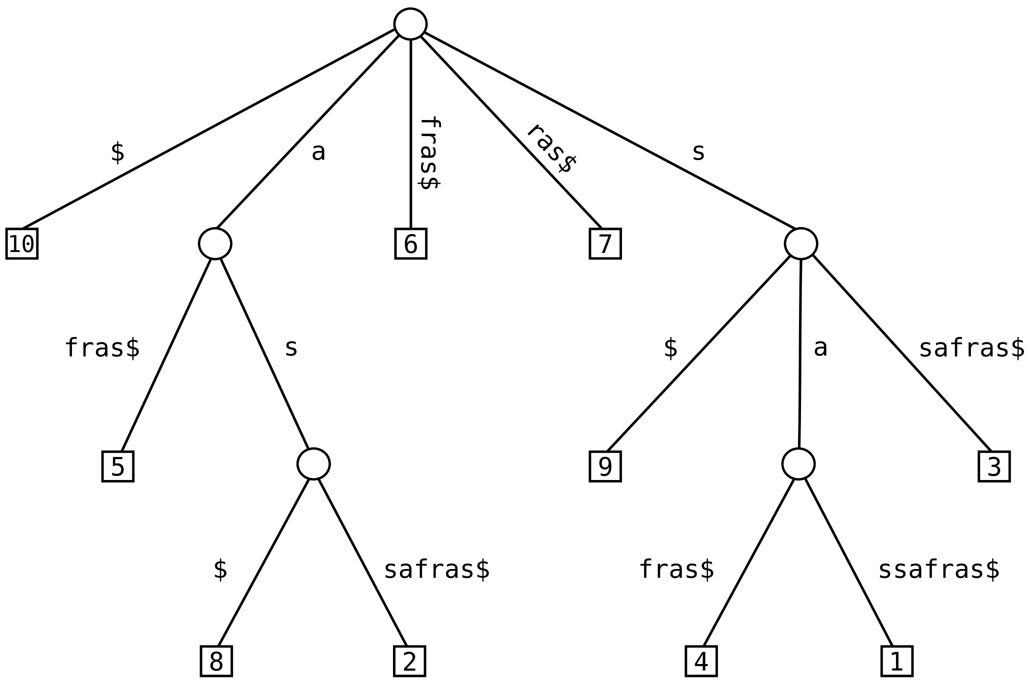


Figure 2.1: Suffix Tree for the string sassafras$

2.2.1 Suffix Tree

A *trie* [Fredkin 1960] is a tree-based data structure built on the characters of a set of strings. Each edge is labeled with a single character from one or more of the strings that it represents. The edges between a node and each of its child nodes must have distinct labels. Internal nodes represent a distinct prefix from one or more of the strings in the set. The path from the root node to a leaf node corresponds to a complete string. Inserting a new string into a trie and searching for a pattern can be performed in linear time in the size of the string to be inserted or searched for.

A *suffix trie* of a string, S, is a trie that is built on S's set of suffixes. The basic idea behind a *suffix tree* is to collapse unary paths in the suffix trie. Morrison [1968] proposed the idea of combining unary edge labels for general tries. This was independently proposed by Weiner [1973] in the context of the suffix tree. A suffix tree contains n leaf nodes, each corresponding to the unique suffixes of S. The leaf nodes act as pointers to the beginning of its corresponding suffix. The suffix tree for the string sassafras$ is shown in Figure 2.1. Note that we append a $ to the string and consider this character to be smaller than every other character in its alphabet, Σ. This is ensure that each suffix is *prefix free*, that is, it prevents a suffix from acting as a prefix to any other suffixes in the tree. We use $ for technical convenience, however in practice a unique termination symbol can be avoided. Each edge is represented as a pair of indexes into S, corresponding to the beginning and end position of the substring that it represents. For example, the right-most edge in Figure 2.1 corresponds to $(4, 10)$, which represents the suffix safras$.

Searching for a pattern can be performed in $\mathcal{O}(mlog\sigma + occ)$ time, where m is the pattern length and occ is the number of occurrences of the pattern in S. The $log\sigma$ factor is due to the need to choose the appropriate edge (of $\mathcal{O}(\sigma)$ possible edges) during each step of the tree traversal. As an example, say we want to search for the pattern, $P =$ sa on the suffix tree in Figure 2.1. Beginning at the root node we traverse down the node via the edge label s. Its right child, a leaf node, corresponds to the suffix ssafras$. Its middle child contains the edge label a which matches with the current position in the pattern, so we follow the edge to its child node. At this point we have successfully matched the pattern sa in the suffix tree. From here we traverse the nodes rooted at our current position. Each leaf we encounter will correspond to an occurrence of the pattern sa. In this case the pattern is found at index 1 and 4 in S. As an alternative example, say we are searching for the pattern asf. This time we follow the a edge from the root node. Here we can move down its right edge, corresponding to the substring as. At this point there are two leaf nodes, however, neither match against f, so we have determined that the pattern asf does not occur in S.

A suffix tree can be computed in linear time [Weiner 1973, McCreight 1976, Ukkonen 1995, Farach 1997] and can be stored in $\mathcal{O}(n \log n)$ bits. In practice suffix trees are rarely used on large texts due to large constant factors dominating the size of the data structure. Kurtz [1999] observed that even the most efficient suffix tree can take up to 10 times the size of its text and, in the worst case, can be a large as 20 or more. There are many solutions to this problem, most notably the suffix array, which will be discussed next.

2.2.2 Suffix Array

The suffix array [Gonnet et al. 1992, Manber and Myers 1993] was designed to reduce the space requirements of the suffix tree. It is a much simpler data structure that can achieve functionality similar to that of a suffix tree in significantly less space. The suffix array of a string S, of length n is simply a permutation of the integers 1 to n such that each integer corresponding to the suffix $S[i..n]$ is sorted in lexicographical order. That is, $S[\mathrm{SA}[i]..n] < S[\mathrm{SA}[i+1]..n]$, for $1 \leq i < n$.

A suffix array can be stored in $n \log n$ bits, and so has the same space complexity of a suffix tree, though with a much smaller constant of proportionality. In practice it is much smaller, at $4n$ or $8n$ bytes depending on the width of the integer used to represent each suffix pointer and the size of the text being indexed. A suffix array can be computed from a suffix tree in linear time, however it is more efficient in terms of time and space to construct it directly. Manber and Myers [1993] originally described an algorithm to construct a suffix array in $\mathcal{O}(n \log n)$ time. There are now a number of efficient linear-time solutions [Ko and Aluru 2005, Kärkkäinen and Sanders 2003, Nong et al. 2009; 2011], see Puglisi et al. [2007] for a survey of a wide range of suffix array construction algorithms. The suffix array for the string $S =$ sassafras$ is shown in Figure 2.2. To see the relationship

i	S[i]	SA[i]	LCP[i]	S[SA[i]..n]
1	s	10	-	`$`
2	a	5	0	`afras$`
3	s	8	1	`as$`
4	s	2	2	`assafras$`
5	a	6	0	`fras$`
6	f	7	0	`ras$`
7	r	9	0	`s$`
8	a	4	1	`safras$`
9	s	1	2	`sassafras$`
10	$	3	1	`ssafras$`

Figure 2.2: The arrays S, SA, LCP and corresponding suffixes for the string sassafras$.

between the suffix array and the suffix tree observe that the suffix array directly maps to an in-order traversal of leaf nodes in the suffix tree from Figure 2.1 – assuming that the nodes in the suffix tree are arranged in lexicographical order. Moreover, there exists a range in the suffix array, SA[$lb..rb$], that maps directly to each node in its suffix tree. For example, in Figure 2.1, the range SA[2..4] maps to the node pointed to by the edge labeled a. Furthermore, the range SA[7..10] maps to the node pointed to by the s edge. Notice that the leaf nodes in this range are 9, 4, 1 and 3, which are identical to the corresponding values in the suffix array range.

Searching for a pattern can be performed in $\mathcal{O}(m \log n)$ time using only the suffix array and the text. The key to searching in a suffix array is to observe that all occurrences of a pattern will be adjacent to each other, as the suffixes are sorted lexicographically. That is, the a search will return a range in the suffix array, SA[$lb..rb$], that contains each position in the text where the pattern occurs. The left and right bounds of this range can be computed by a binary search over the array, performing $\mathcal{O}(m)$ character comparisons at each step. The number of occurrences of a pattern can then be found in constant time by returning the value $rb - lb + 1$. We can list the locations of each occurrence in $\mathcal{O}(occ)$ time by iterating through the values in the range SA[$lb..rb$]. Manber and Myers [1993] described how to avoid these $\mathcal{O}(m)$ comparisons during each binary search to reduce the run-time to $\mathcal{O}(m + \log n)$, however, this comes at the cost of two auxiliary arrays computed from the longest common prefix (LCP) array.

Let the function *lcp* return the length of the longest common prefix between strings. The LCP array represents the longest common prefix of adjacent suffixes in a suffix array. That is, LCP[i] $= lcp(S[\mathrm{SA}[i-1]..n], S[\mathrm{SA}[i]..n])$ for $1 < i \leq n$, and LCP[1] $= \emptyset$. The LCP array is shown in Figure 2.2. As an example, LCP[9] = 2, as the suffixes $S[\mathrm{SA}[9]..n]$ = safras$ and $S[\mathrm{SA}[8]..n]$ = sassafras$ share a common prefix of sa. The LCP array can be computed in linear time [Kasai et al. 2006, Kärkkäinen et al. 2009, Fischer 2011] and combined with a suffix array can be used to replicate most of the functionality of a suffix tree, but not all. It is possible to fully replicate the functionality of a suffix tree by creating an *enhanced suffix array* [Abouelhoda et al. 2004], this is achieved by including the LCP array and a number of auxiliary arrays which are used to describe the structure of the suffix tree it represents.

2.2.3 Succinct Data Structures

Succinct data structures are designed to have the same functionality as conventional data structures, but using as little space as possible. Jacobson [1989] observed that we can represent the structure of a tree or graph as a bitvector and that it is possible to simulate traversal of these data structures using a number of constant time operations over the bitvector. This has led to a variety of applications, including compressed text indexes [Navarro and Mäkinen 2007], succinct trees [Jacobson 1989, Munro and Raman 2001, Benoit et al. 2005, Barbay et al. 2007], graphs [Claude and Navarro 2007, Brisaboa et al. 2009] and binary relations [Barbay et al. 2007]. Jacobson [1989] identified three fundamental operations on bitvectors that are used as the basis for more complex functionality of succinct data structures. Given a bitvector B, a position i, and a bit b (either 0 or 1) we define

$\mathsf{access}(B, i)$ return the bit value at position i in B.

$\mathsf{rank_b}(B, i)$ returns the number of occurrences of b before position i in B.

$\mathsf{select_b}(B, i)$ returns the position of the ith b in B.

There is an interesting symmetry between rank and select, namely

$$\mathsf{rank_1}(B, \mathsf{select_1}(B, i)) = \mathsf{select_1}(B, \mathsf{rank_1}(B, i)) = i.$$

Many succinct operations can be solved in terms of each other. For example, access can be solved in terms of rank

$$\mathsf{access}(B, i) = \mathsf{rank_1}(B, i+1) - \mathsf{rank_1}(B, i) \quad \mathsf{for}\ i \in [1, n),$$

and $\mathsf{rank_0}$ can be solved in terms of $\mathsf{rank_1}$

$$\mathsf{rank_0}(B, i) = i - \mathsf{rank_1}(B, i).$$

Moreover, select can be solved in terms of rank, although, this typically comes with a logarithmic time penalty as a binary search is required over the rank structure. Jacobson [1989] gives a data structure which adds $o(n)$ bits overhead on a bitvector and provides constant time rank and logarithmic time select operations. Clark [1996] and Munro [1996] later improved select to run in constant time. The main idea behind these data structures it to maintain a hierarchical sampling of cumulative rank counts at regularly spaced intervals across a bitvector. A rank operation is reduced to querying the sampled blocks in constant time before using a population count (*popcnt*), which counts the number of 1 bits in a machine word. Key to the efficiency of these data structures is the speed at which *popcnt* is performed. Basic methods use a lookup table of pre-computed population counts. Another approach is to use broadword techniques such as Knuth [2007]'s sideways addition rule, which is known to work well in practice [Vigna 2008]. As *popcnt* is such a widely used operation it was recently included as a machine instruction POPCNT which was introduced along side the SEE4.2 instruction set and vastly improves run-time.[1]

Raman et al. [2002] describe a compressed bitvector representation that theoretically gives constant time rank and select operations, however, in practice it runs in $\mathcal{O}(s)$ time, where s is a specified sampling rate. Claude and Navarro [2009] use a fixed sampling rate of 15 by using a pre-computed lookup table. For larger sampling rates a lookup table is no longer practical, as its space requirements are too large. Navarro and Providel [2012] provide a solution for larger sampling rates by removing the lookup table altogether and manually encoding/decoding block offsets on-the-fly. Okanohara and Sadakane [2007b] present a number of compressed bitmap representations giving varied trade-offs in terms of size and speed of operations, for example, their sdarray compactly represents sparse bitvectors and provides efficient rank and select operations which is almost as fast as an uncompressed bitvector, however, it is only effective when the bitvector is very sparse, for example, densities below 5%. Kärkkäinen et al. [2014a] describe a hybrid scheme that represents a bitvector by dividing it into blocks and separately encoding each with one of a variety of techniques. This method is particularly effective when the distribution of bits throughout a bitvector is irregular and each block can be compressed effectively by alternative methods.

In the case of arbitrary sequences, that is, when Σ is no longer a binary alphabet, rank and select operations can be solved using a wavelet tree [Grossi et al. 2003]. A wavelet tree decomposes an alphabet into a balanced binary tree of depth $\log \sigma$, successively halving Σ at each node until reducing the leaf nodes, each of which will correspond to a single character. A wavelet tree stores n bits at each level, giving a total upper bound of $n\lceil \log \sigma \rceil$ bits. The tree can be explicitly compressed by representing the bitvectors in compressed form [Raman et al. 2002] or implicitly compressed by replacing the balanced tree with the Huffman tree of the sequence [Mäkinen and Navarro 2007]. Wavelet trees have been extensively studied over the last decade and have found uses in many domains such as compressed text indexes, which are discussed next. See Navarro [2014] for a recent review of the wide variety of applications of wavelet trees.

[1]POPCNT is not actually considered part of SSE4.2, however, it was introduced at the same time. In fact, POPCNT and LZCNT have their own dedicated CPUID bits to indicate support.

2.2.4 Compressed Full-Text Indexes

The fundamental issue with classic text indexes such as the suffix tree and suffix array is their space requirements relative to the size of the input text. Although the space complexity for both data structures is theoretically linear in the length of the input string, they both come with large constants, resulting in data structures that are significantly larger than the text. Furthermore, to support pattern matching, the text is required along with the data structure itself. In recent years there have been a number of solutions to this problem in the form of compressed full-text indexes, also called self indexes. These indexes provide the functionality of a classic text index in space close to that of the compressed text, and, as a result, can actually replace the text. See Navarro and Mäkinen [2007] for a comprehensive overview of the field. There are many variations of self-indexes all making use of the succinct data structures described in Section 2.2.3. Most build a compressed representation of a suffix tree [Sadakane 2007] or suffix array [Sadakane 2003, Grossi and Vitter 2005]. There are a number of compressed self-index based on a Lempel-Ziv family of algorithms, originally based on LZ78 [Navarro 2004, Ferragina and Manzini 2005, Arroyuelo et al. 2006], and more recently on LZ77 [Kreft and Navarro 2010, Gagie et al. 2012; 2014]. Indexes based the Burrows Wheeler Transform (BWT) [Burrows and Wheeler 1994] of a text is a very active area of research, for example, the large family of FM-indexes [Ferragina and Manzini 2000, Ferragina et al. 2004, Mäkinen and Navarro 2005, Kärkkäinen and Puglisi 2011], are heavily used in Bioinformatics [Langmead et al. 2009, Simpson and Durbin 2010; 2012].

2.3 Text Compression

The core idea behind compression is to represent data in less space than its original representation, reducing the cost in terms of time and resources to transmit and/or store the data, for example, over a physical medium such as a network, or levels of a systems memory hierarchy. Algorithms are divided into two distinct categories; *lossless* and *lossy* methods. In *lossless* compression the aim is to encode a text without the loss of any information, that is, the original text can be retrieved verbatim from its compressed representation. On the other hand, *lossy* compression allows for a certain loss of accuracy during encoding. This approach is suitable for data that is already an approximation, for example, data converted from an analog source such as an image, or an audio or video signal. The compression methods discussed throughout this thesis will be *lossless*.

The fundamental idea behind text compression is to take a stream of symbols and convert them into codes. An effective compression algorithm will output codes in space smaller than that of its original symbols. Note that the notion of a symbol is abstract, it could be a character, word, sentence, phrase or arbitrary substring.

There is a lower bound on the number of bits required to encode a symbol. Shannon

[1948] defines the information content or *self-information* of a symbol with a pre-defined probability distribution as

$$I(p) = \log(1/p_i). \tag{2.1}$$

Self-information represents a lower bound on the number of bits an ideal compressor requires to represent a random symbol. The *entropy* of a message, S, is the average of the information content for each individual symbol, that is

$$H(S) = \sum_{i=1}^{n} p_i \; I(p). \tag{2.2}$$

Manzini [2001] formalized this concept in terms of a finite string. For a string S of length n and an alphabet Σ, the 0-order empirical entropy of a string S is defined as

$$H_0(S) = \sum_{c \in \Sigma, n_c > 0} \frac{n_c}{n} \log \frac{n}{n_c}, \tag{2.3}$$

where n_c is the number of occurrences of character c in S. The 0-order empirical entropy is the average number of bits an ideal compressor which keeps statistics of single symbols uses to encode each symbol. The idea here is that symbols with low probability have high information content, conversely, symbols with high probability have a low information content and require fewer bits to represent. nH_0 defines a lower bound on the number of bits required to represent a complete string. For example, the 0-order empirical entropy of the string S = `sassafras`, $n = 9$, $n_a = 3$, $n_f = 1$, $n_r = 1$, $n_s = 4$ is

$$H_0(S) = \frac{3}{9} \log \frac{9}{3} + \frac{1}{9} \log 9 + \frac{1}{9} \log 9 + \frac{4}{9} \log \frac{9}{4} = 1.752.$$

The average length of a codeword for the string S using a 0-order ideal compressor is 1.752 bits and can represent S in $nH_0(S) = 15.768$ bits.

A 0-order statistical model maintains a probability distribution of independent symbol occurrences. More advanced methods for representing symbol probabilities consider the context in which each symbol appears, for example, k symbols proceeding it, on the assumption that knowledge these contexts will lead to a more effective probability distribution. This is especially prevalent when compressing English text. For example, if 1-order symbol statistics are computed at a character level the letter u would be highly likely to occur after the letter q. Considering symbols as words, intuitively, given the word *suffix* it would be highly likely that the next word encountered is *array* or *tree* and extremely unlikely that it would be *tambourine*. Formally, the kth-order empirical entropy of a string S is defined as

$$H_k(S) = \frac{1}{n} \sum_{\alpha \in \Sigma^k, S_\alpha \neq \epsilon} |S_\alpha| H_0(S_\alpha), \tag{2.4}$$

where S_α is a collection of symbols from S comprised of all the symbols that are followed by each context (substring) α of length k in S. It always holds that $H_{k+1} \leq H_k \leq H_0 \leq \log \sigma$.

The kth-order empirical entropy of the string $S = \texttt{sassafras}$, $n = 9$, $n_a = 3$, $n_f = 1$, $n_r = 1$, $n_s = 4$, for $k > 0$ is

$$\begin{aligned}
H_1(S) &= \frac{1}{9}(3H_0(S_a) + H_0(S_f) + H_0(S_r) + 3H_0(S_s)) \\
&= \frac{1}{9}(3H_0(sfs) + H_0(r) + H_0(a) + 3H_0(asa)) \\
&= 0.611.
\end{aligned}$$

$$\begin{aligned}
H_2(S) &= \frac{1}{9}(H_0(S_{af}) + H_0(S_{as}) + H_0(S_{fr}) + H_0(S_{ra}) + 2H_0(S_{sa}) + H_0(S_{ss})) \\
&= \frac{1}{9}(H_0(r) + H_0(s) + H_0(a) + H_0(s) + 2H_0(sf) + H_0(a)) \\
&= 0.222.
\end{aligned}$$

$$\begin{aligned}
H_3(S) &= \frac{1}{9}(H_0(S_{afr}) + H_0(S_{ass}) + H_0(S_{fra}) + H_0(S_{saf}) + H_0(S_{sas}) + H_0(S_{ssa})) \\
&= \frac{1}{9}(H_0(a) + H_0(a) + H_0(s) + H_0(r) + H_0(s) + H_0(f)) \\
&= 0.
\end{aligned}$$

Note that higher order models offer lower average entropy per symbol as we have a more effective model of S. Theoretically, by $k = 3$, we are sure which symbol will occur next.

2.3.1 Modeling

Rissanen and Langdon [1981] categorize text compression into two distinct parts: *modeling* and *coding*. In this section we discuss modeling and we follow with a discussion of coding techniques in the next. A model is essentially a collection of information about an input such as statistical probabilities of symbol occurrences, or a description of other types of repetitiveness in the data. In order to provide effective compression it is critical that the model provide an accurate representation of its input. There are a wide range of modeling techniques used in text compression [Bell et al. 1989]. A 0-order statistical model maintains a probability distribution of individual symbol occurrences. Advanced statistical models use higher order probability distributions and are efficient in practice, for example, the Prediction by Partial Matching (PPM) algorithm of Cleary and Witten [1984].

Coding is the process of representing symbols using information from the model. The core idea is to map short codewords to frequently occurring symbols, and longer codewords to rarer symbols. Statistical models provide information to a coder which in turn maps symbols to variable length codewords. An alternative approach is dictionary modeling. Here substrings are replaced with codewords. The model maintains a dictionary comprised of a set of strings. A text is compressed in terms of phrases in the dictionary and the encoding is simply pointers referencing the dictionary.

There are many approaches to predict probabilities in a model. A *static* model is a fixed mapping between symbol and codeword. The model does not depend on the input symbols at all. Both the encoder and decoder share the model so there is no need to communicate it with the encoding. Static models are useful when properties of the input are known in advance. For example, compressing Human DNA using a static code for each of the four neuclotides, $\Sigma = \{a, c, g, t\}$, $a = 00, c = 01, g = 10, t = 11$ is efficient in practice because each symbol occurs with almost equal probability.

A *semi-static* model computes symbol probabilities off-line in an initial pass of the text. Compression is performed in a second pass using information gained from the model. If a fixed length code is used to represent each symbol a static and semi-static model can provide random access to the text, that is, direct access to the compressed data at any point in an encoding. Random access to compressed data is a desirable aspect of many compression systems, for example, information retrieval systems or document databases where specific documents or snippets of text are extracted from large compressed files. The main disadvantage of static and semi-static models is that the dictionary needs to be transmitted with the encoding to the decoder. Compressing large collections with a semi-static model can be problematic, as the vocabulary can grow significantly larger than physical memory. In our own experiments a parsing of Clueweb09 Category A, a 15TB English text web crawl [2] generated a 13GB uncompressed vocabulary. Most notably, close to 50% of the lexicon was comprised of non-word symbols that occurred only once throughout the collection. Such hurdles to scalability have not been reported before in literature due to the relatively small collection sizes used in experiments. For example, Turpin et al. [2007] use a semi-static model for document compression, but the collections they used were less than 100GB. In general, overall compression achieved by a semi-static approach is limited by its insensitivity to any global repetitive properties of the collection. The best case for these methods is a reduction of the text size to around 20% of its original size not including the space required for the model.

Adaptive models avoid the explicit storage and transmission of the model with an encoding. Both compression and decompression begin with no information about its input or some predetermined state. They progressively learn an effective model by adapting to knowledge of previously encountered symbols. This is an online approach as it requires only one pass over the text, which is useful if the text is significantly larger than memory and a semi-static model is not be feasible. A disadvantage of adaptive models is the extra cost of maintaining and updating the model during both compression and decompression.

[2]http://lemurproject.org/clueweb09

Dictionary Based Models

In this section we give an overview of adaptive dictionary-based models, which act as the basis for the contributions in this theses.

LZ77 [Ziv and Lempel 1977] An LZ77 encoder examines an input sequence through a window that consists of two components; a *dictionary* that contains a portion of recently encoded sequences and a *look-ahead* buffer, which contains the next portion of the text to be encoded. The dictionary is built from previously occurring substrings in a text. It is typically implemented as a fixed size sliding window, however, it can be allowed to extend back to the beginning of a text. The look-ahead buffer can also be fixed length or unbounded giving a space time trade-off between window size and compression effectiveness. Compression involves finding the longest substring in the dictionary that matches the current prefix in the look-ahead buffer. When a match is found in the dictionary, for example, a substring of length i, the model notifies the coder and slides the window $i + 1$ positions across the text. This model does not directly support random access to the compressed text. In fact, due the to adaptive nature of the dictionary, decoding must commence at the beginning of the text or at specific synchronization points coded in the output [Witten et al. 1999].

A traditional LZ77 encoder outputs a triple (p, l, c), where p is an offset into the dictionary from the current position in the look-ahead buffer, l is the length of the longest substring matching in both buffers, and c is the character in the look-ahead buffer that directly follows the match. This triple is commonly known as an *LZ factor* or *phrase*. The set of LZ factors for the entire text is the *LZ factorization* or *LZ parse*. If the current character to encode is not found in the dictionary we output $(0, 0, c)$, identifying c as a literal character.

An example of traditional LZ77 factorization using a fixed size sliding window of four characters on the string $S =$ abaababaabaabaw is shown in Figure 2.3. At the top of the figure we show the various stages of the LZ77 factorization including the (p, l, c) factors. Observe how the window slides across the text, advancing after each factor. Underlined characters correspond to a match in the dictionary and look-ahead buffer or a single literal character. Gray characters in the look-ahead buffer correspond to the c element of each computed factor. At the bottom of the figure we show the textual representation of the LZ factorization. Note that the position values in Figure 2.3 are relative values. That is, they correspond to an offset into the dictionary from the current position in the text.

There is a wide range of LZ77-based algorithms (see Salomon [2004] for a detailed overview). They tend to fall into three main classes; restrictions on window size, methods of substring selection in the dictionary, and methods of coding LZ factors. All aspects give varying compromises between speed, memory and compression effectiveness. The larger the window the slower it will be to search for factors and more memory that will be

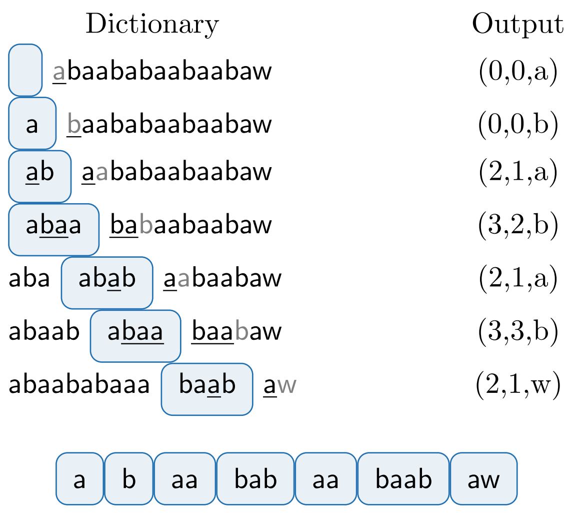


Figure 2.3: LZ77 factorization of the string abaababaabaabaw using a sliding dictionary window of four characters. On the left shows the position of the window and the look-ahead buffer during an encoding. The right shows each computed LZ factor (p, l, c) where p is the relative position or offset from the beginning of the look-ahead buffer, l is the length of the factor and c is the current symbol at the beginning of the look-ahead buffer.

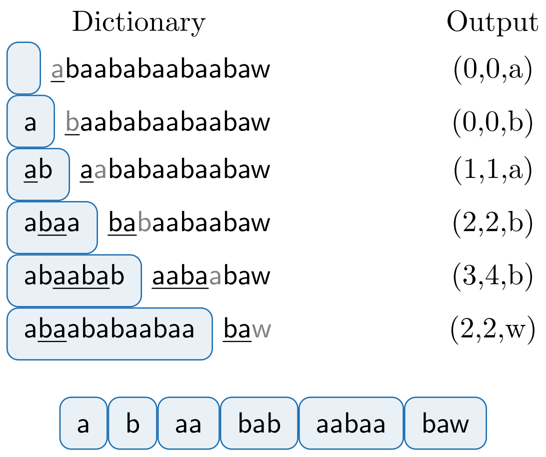


Figure 2.4: LZ77 factorization of the string abaababaabaabaw using an unbounded dictionary. On the left shows the growth of the unbounded dictionary during an encoding. The right shows each computed LZ factor (p, l, c) where p is the absolute position or offset from the beginning of text, l is the length of the factor and c is the current symbol at the beginning of the look-ahead buffer.

consumed, however, the dictionary will have a greater selection of strings to select from, and in turn compression should improve. On the other hand, using a smaller window may reduce run-time and use significantly less memory, but at the cost of worse compression, as the dictionary does not represent the text very well.

An example of LZ77 factorization using an unbounded window is shown in Figure 2.4. Note that each factor's position value is now absolute, that is, it corresponds to an index in the uncompressed text and not an offset from its current position. At the top of the diagram we see the incremental stages of the factorization. At the bottom we show the textual representation of the LZ factorization. In this example the string is factorized with one less factor compared with using a small four character window. While this is not a remarkable improvement, on larger texts the difference can be significant.

There are many methods for selecting phrases in a dictionary. When the dictionary is small (for example zlib typically uses a dictionary of less than 32KB), the most effective method is to hash all possible strings and to select the longest match. As the dictionary fits comfortably in higher levels of cache (L2 cache sizes on a current CPU range in megabytes), using more advanced methods would incur additional costs. When using a larger window it is better to create an index over the dictionary to improve selection time such as a multi-level hash [Sadakane 2000], a suffix tree [Gusfield 1997], or a suffix array [Chen et al. 2008, Goto and Bannai 2013, Kärkkäinen et al. 2013a;b; 2014b]. Phrase selection has an effect on both compressed size and decoding speed. For example, the final factor in Figure 2.4, $(2, 2, w)$, which corresponds to the substring ba, occurs in four distinct positions in S. Selecting the substring closest to the current position in the stream will improve cache locality and reduce the encoding size if a fixed window is used by being able to encode smaller position offset values.

Traditional LZ77 codes a factor in $\lceil \log D \rceil + \lceil \log L \rceil + \lceil \log \sigma \rceil$ bits, where D is the length of the dictionary window and L is the length of the look-ahead buffer. Most practical general-purpose LZ77 implementations use statistical coding techniques, for example, DEFLATE/zlib/gzip uses a Huffman code and LZMA/LZMA2/7zip/xz uses range coding. These methods are discussed later in this chapter. We experiment with factor coding in our work on semi-static dictionary compression in Chapter 3.

A desirable aspect of most adaptive dictionary-based models is their fast decoding throughput. LZ77 decoding is simple and effective. The decoder maintains a window in an identical manner to the encoder, however, it does not build any complex data structures over the dictionary. For each factor it read,s it simply copies phrases from the dictionary to its output stream.

LZSS [Storer and Szymanski 1982] If there are a large proportion of rarely occurring characters in a string, the LZ77 parse just discussed will output many wasteful $(0, 0, c)$ factors. The most common variation of LZ77 that is used by many general-purpose compressors is LZSS by Storer and Szymanski [1982]. The main contribution of LZSS was to represent LZ factors as a double (p, l), identifying that the character field is potentially wasteful and can be represented more efficiently. Another improvement was to include character literals in the output stream when it is more expensive to code the factor. This was achieved by adding a flag bit before each output to identify if the next entry is a factor or a literal character. A variation of LZSS is to encode characters with the double $(0, c)$

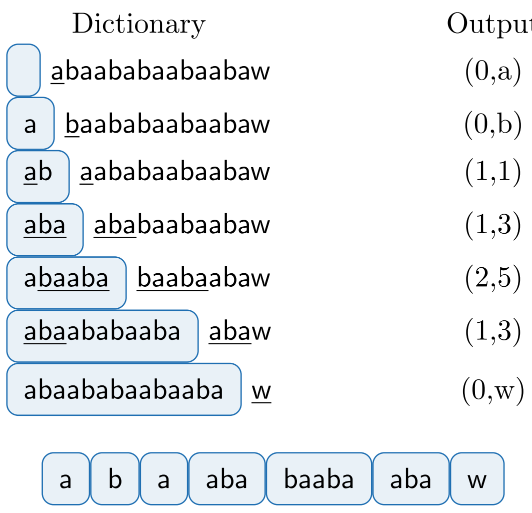


Figure 2.5: LZSS-style factorization of string abaababaabaabaw using an unbounded dictionary. On the left shows the growth of the unbounded dictionary during an encoding. The right shows each computed LZ factor (p, l) where p is the absolute position or offset from the beginning of text and l is the length of the factor.

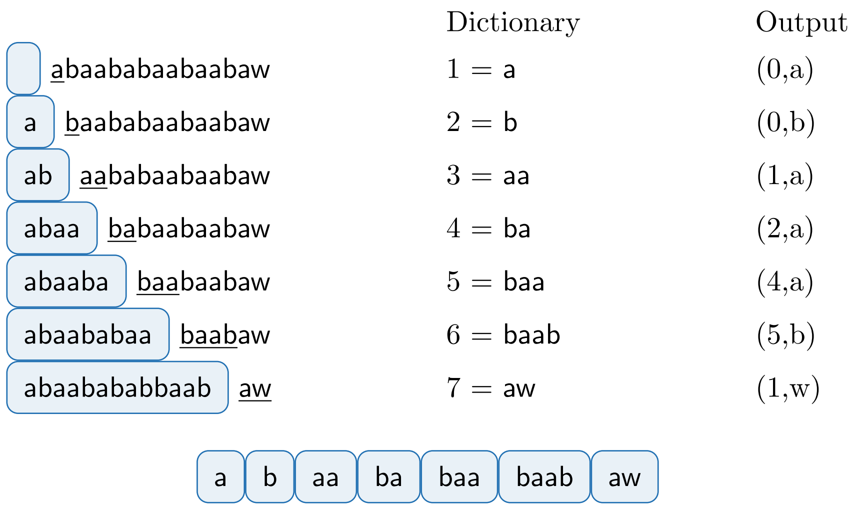


Figure 2.6: LZ78 factorization of string abaababaabaabaw using an unbounded dictionary. The left depicts the position of the look-ahead buffer. The middle contains each dictionary entry and the right shows the (i, c) pairs where i is an index to a phrase in the dictionary and c is the next character in the text.

as large modern dictionaries force this case to be an extremely rare occurrence, saving 1 bit per factor, which can be significant when compressing larger texts. An example of LZSS factorization using an unbounded dictionary is shown in Figure 2.5. Bell [1986] expand LZSS, by encoding position offsets with a variable-bit code and length values with an Elias-γ code.

LZ78 [Ziv and Lempel 1978] Bounded-window LZ77-based algorithms make an implicit assumption that redundancy in a text is local, in which repetitions occur close to each other. If the dictionary window is not sufficiently large enough to capture redundancy in a text, compression will be poor.

Ziv and Lempel [1978] propose an alternative approach where an explicit dictionary of phrases is used instead of a sliding window over previously seen text. Each factor is coded as a double (i, c) where i is an index or identifier to an existing phrase in the dictionary and c is the next character to encode in the text. The concatenation of the existing phrase and character forms a new entry in the dictionary. When a character does not exist in the dictionary it is identified by the double $(0, c)$ where 0 is treated as a special index that does not point to a phrase, in other words, the 0 acts as a flag. The LZ78 factorization for the string $S = \mathsf{abaababaabaabaw}$ is shown in Figure 2.6. Each line corresponds to a single step during the factorization. On the left is the current position in the text being processed. An underlined substring corresponds to an phrase match in the dictionary. The middle column is the index/phrase value that is added to the dictionary during each step. The right column is the output of the encoder. At the bottom of the figure we show the textual representation of the LZ factorization.

Encoding is faster than LZ77 as there is no need to search for matching substrings in the dictionary. Dictionary lookup can be solved efficiently with the use of a trie, however, there is a time penalty during decoding as the dictionary has to be built and maintained. As each factor is comprised of an existing phrase and a new character, the decoder simply adds each new entry to the dictionary as it is executes. A disadvantage of having an explicit dictionary is that is continuously grows. Moreover, phrases in dictionary might not actually be used again. In practice a dictionary can not grow infinitely. There are a number of possible approaches to mitigate the issue, such as to simply stop adding phrases when it reaches a predefined size. Another method is to reset the dictionary and start from an empty state. For this to work the encoder must write a special reset code to the output stream. When the decoder encounters the symbol it will reset its dictionary.

LZW [Welch 1984] Similar to the improvements made by LZSS, Welch [1984] proposed a technique for removing the character element from a LZ78 double (i, c) and to simply output phrase indexes from the dictionary. An example of LZW factorization for the string $S = \mathsf{abaababaabaabaw}$ is outlined in Figure 2.7. Initially, all symbols of the strings alphabet are added to the dictionary. Each unique symbol in S is added to the dictionary and assigned an index, in this case $\Sigma = \{a, b, w\}$ and $1 = \mathsf{a}$, $2 = \mathsf{b}$ and $3 = \mathsf{w}$. Encoding works just like LZ78. We continually read symbols from the look-ahead buffer until we create a phrase that we have not seen before. Then, we output the code for the previous known phrase and assign a code for the new unknown phrase. For example, in Figure 2.7 we begin with the symbol a. This phrase exists in the dictionary so we concatenate it with

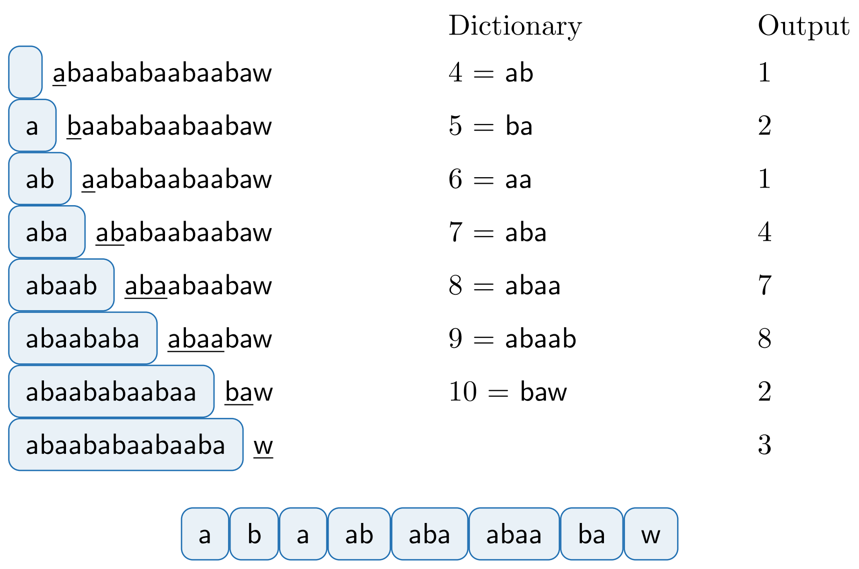


Figure 2.7: LZW factorization of string abaababaabaabaw with an initial dictionary mapping for the alphabet $\Sigma = \{a, b, w\}$ of 1 = a, 2 = b and 3 = w.

the next symbol in the string, giving the phrase ab. As ab is not found in the dictionary we output the code, 1 = a, add 4 = ab into the dictionary, and advance in the string treating the last symbol as the new phrase. The process is repeated until we reach the end of the string. Decoding initially populates the dictionary with each symbol in the phrase alphabet and then proceeds in the same manner as LZ78.

2.3.2 Coding

In this section we cover a number of common coding techniques, from statistical Huffman and arithmetic coding to static bit-, byte- and word-oriented codes.

Statistical Coding

In this section we briefly cover the two most important although quite different statistical coding techniques, Huffman coding and Arithmetic coding. Huffman or minimum redundancy coding is usually faster than arithmetic coding, however, arithmetic coding is capable of achieving significantly better compression. The effectiveness of statistical coding methods hinge on the accuracy of the probability distribution given by the model. If the model fails to accurately represent its input, compression will be poor.

Huffman Code [Huffman 1952] Huffman coding is a method for computing an optimal minimum redundancy code for a set of symbols given their probability distribution. In its most basic form the core of the algorithm is the Huffman tree. This tree is used to assign prefix-free codewords to each symbol given knowledge of the symbol's probability distribution. A prefix-free code implies that no code word is a prefix of another codeword. This is necessary in order to determine where one code stops and another begins.

Constructing a Huffman tree is conceptually straightforward. For each symbol we create a leaf node corresponding to its probability. The two least frequent symbols are combined into a subtree with its root node containing the sum of their probabilities and added back to the set of nodes. This process is repeated, merging symbols and subtrees and combining their probabilities, until there is a single node remaining, which is the root of the tree. The path from the root to a leaf node corresponds to a unique codeword for a symbol. Moving down the tree to a left child appends a 0 to the code and moving the right child appends a 1. Efficient techniques for generating minimum redundancy codes avoid computing the Huffman tree altogether using table based methods, which vastly improved decoding [Moffat and Turpin 1997; 1998]. Huffman coding with a static or semi-static model requires the symbol probabilities to be transmitted with the encoding, which can be costly, however, coding can also be performed with an adaptive model [Cormack and Horspool 1984, Lu and Gough 1993].

Traditionally, codewords were assigned to symbols at a character level by a bit-oriented code. This generally leads to poor compression of natural language texts to around 60% of their original size [Moffat and Turpin 2002]. Using words as symbols leads to much better compression, as the word distribution of natural language texts is much more biased than the character distribution [Ziviani et al. 2000]. A word-based Huffman code can compress typical natural language texts to nearly 25% [Moffat 1989, Witten et al. 1999].

de Moura et al. [2000] describe two byte-oriented coding techniques, Plain Huffman (PH) and Tagged Huffman (TH). These schemes provide much faster encoding and decoding speeds at a cost of slightly reduced compression performance (5% to 10%) compared to bit-oriented codes. Operating at a byte level eliminates the need for expensive bit manipulations required in traditional Huffman coding. These codes support random access as each byte represents a codeword boundary. In Tagged Huffman codes a flag bit is reserved in each byte to signal the start of a codeword. This allows for fast compressed pattern matching: a pattern can be encoded with the same model and searched for directly in the compressed text.

Brisaboa et al. [2007a] discuss End Tagged Dense Codes (ETDC), an improvement to Tagged Huffman coding, by modifying the flag bit to symbolize the end of a codeword. This reduces the requirement to build a Huffman tree to ensure each symbol is a valid prefix code. Dense codes provide a useful space trade-off to Tagged Huffman codes. They are simpler to implement and are faster at compression and decompression. Like the other Huffman approaches, it is necessary to build an explicit dictionary of symbols. There is a family of dense codes described in literature that can be used for corpus compression. In later work, Brisaboa et al. [2007b] describe Pair-Based and Phrase-Based

End-Tagged Dense Codes (PETDC and PhETDC), two extensions to ETDC that use symbols of a higher order. In PETDC symbols can be either words or pairs of words. In PhETDC symbols are considered words or phrases of words. It is reported that PETDC can reduce a text by 70%, and PhETDC by 77%, outperforming all current zero-order word-based semi-static compressors [Brisaboa et al. 2007b]. A Dynamic End Tagged Dense Code (DETDC) [Brisaboa et al. 2008] is a dynamic version of ETDC, where the model is transmitted along with the encoding, much like an adaptive compression algorithm. A Dynamic Lightweight End Tagged Dense Code (DLETDC) [Brisaboa et al. 2010] is a modification to this scheme that reduces the cost of transmitting the model.

Arithmetic Code [Rissanen 1976; 1979] A limitation of Huffman coding is that each codeword for a symbol must approximate $\log 1/p$ with an integral number of bits. As a consequence the minimum codeword length for a symbol in a Huffman code is 1 bit. If a symbol had a 33% probability of occurring next then its optimal code length is $\log 1/0.33 = 1.58$ bits, however, a Huffman code would at best use a 2 bit codeword. Furthermore, if a symbol had a 95% probability of occurrence it should be coded in close to 0.074 bits. A Huffman code will be approximately 13 times larger. Arithmetic coding is effective when dealing with such highly skewed probability distributions. Rather than replacing symbols with separate codewords, arithmetic coding encodes an entire string as a single real number represented as a binary fraction selected in the interval $[0, 1)$, in space very close to entropy.

Given a set of symbols, a statistical model and an initial interval $[0, 1)$, each symbol is assigned a sub-interval proportional to its probability distribution. When a symbol is encoded the interval is reduced to the sub-interval it corresponds to. The updated interval is now divided into sub-intervals based on the static or adaptive symbol probabilities from the model. Once the last symbol has been processed the value that represents the entire encoding is a real number selected from the final interval.

Arithmetic coding is most useful when coupled with a high-order adaptive compression model and is especially effective on large alphabets [Witten et al. 1987, Moffat et al. 1998]. If the model has fixed probabilities, that is, a static or semi-static model is used, an arithmetic coder will run considerably slower than a Huffman coder [Moffat and Turpin 1997]. Range Coding [Nigel and Martin 1979] is a form of arithmetic coding where encoding is performed across alternative bases, for example, bytes rather than bits.

Integer Coding

Integer coding is useful in the situation where it is problematic or undesirable to compute a model of the input, such as when the input is too large to process in memory or when the source alphabet is unbounded. This section describes methods to compress arbitrary positive integers without reliance on a statistical model. The basic idea is to let both the encoder and decoder process integers and codewords independently.

Unary Code The unary code of an integer x is a sequence of x 1 bits followed by a single 0 bit. Alternatively, this could be represented as a sequence of x 0 bits followed by a single 1 bit. For example, the integer 7 would be encoded as 11111110 or 00000001. Decoding is straightforward: read from the input stream one bit at a time counting the number of bits until a change. In the case where x is assumed to be a positive non-zero value, that is, $x > 0$, the integer can be coded as a sequence of $(x - 1)$ 1 bits, followed by a single 0 bit. Using this method the integer 7 would be encoded as 1111110. Unary codes are useful when encoding small values, however, they quickly become ineffective when representing larger numbers.

Elias Gamma Code [Elias 1975] An Elias γ-code represents a positive integer $x > 0$, as a concatenation of two separate codes: a prefix and a binary suffix. The prefix is encoded as a unary code of the value $1 + \lfloor \log x \rfloor$ and the binary suffix is encoded as the value $x - 2^{\lfloor \log x \rfloor}$ in $\lfloor \log x \rfloor$ bits, for a total of $1 + 2\lfloor \log x \rfloor$ bits. For example, the γ-code of the integer 7 is derived as follows. The unary prefix is, $1 + \lfloor \log 7 \rfloor = 3$, which corresponds to the code 110. The binary suffix is, $7 - 2^{\lfloor \log 7 \rfloor} = 3$, so we encode the value 3 in $\lfloor \log 7 \rfloor = 2$ bits, that is, 11. Concatenating both values results in γ-code, 11011. To decode a γ-code first we decode the unary prefix, x_p, then the next $x_p - 1$ bits are read which represents the binary suffix code, x_b. The final value can be computed as $x = 2^{x_p - 1} + x_b$.

Elias Delta Code [Elias 1975] An Elias δ-code is similar to a γ-code. It separates a code into two components, a prefix and a binary suffix, however, its prefix code is encoded as a γ-code instead of unary. For example, the δ-code for integer 7 is derived as follows. First the prefix value, $1 + \lfloor \log 7 \rfloor = 3$, is encoded as a γ-code, 101. The binary suffix remains the same as its γ representation, that is, $7 - 2^{\lfloor \log 7 \rfloor} = 3$ is encoded in $\lfloor \log 7 \rfloor = 2$ bits, resulting in 11. Concatenating the prefix and binary suffix gives the δ-code, 10100. A δ-code for an integer x can be encoded in $1 + 2\lfloor \log \log 2x \rfloor + \lfloor \log x \rfloor$ bits. Decoding operates in the same manner as γ decoding.

δ-codes are efficient at representing larger integer values, however, for small values a γ-code will be shorter. As an example, the δ-code for the value 4 is encoded in 4 bits, where as the γ-code takes 3 bits. Conversely, the δ-code for 1,000,000 takes 28 bits compared to the γ-code which takes 39 bits [Witten et al. 1999].

Golomb Code [Golomb 1966] For a positive integer $x > 0$ and a parameter b, a Golomb code is represented as two parts: first a unary code of a quotient, $q = 1 + \lfloor (x-1)/b \rfloor$ and second, the binary representation of its remainder $r = x - qb - 1$. Depending on the value of b, the remainder r may require $\lfloor \log b \rfloor$ or $\lceil \log b \rceil$ bits. Golomb codes are generally more space efficient than Elias codes and they are faster to decode if an appropriate b value is selected. Witten et al. [1999] suggest $b = 0.69 \cdot \bar{x}$ where $\bar{x}$ is the average of the integer values to be coded.

A Rice code [Rice and Plaunt 1971] is a variation of a Golomb code where b is restricted to powers of 2. This improves encoding and decoding speed as masks and bit shifts can be used to generate codes. A further advantage is that the remainder r will always stored in $\lceil \log b \rceil$ bits. The disadvantage of a fixed b value is that compression could be less effective than traditional Golomb codes as the optimal value for b might not be close to a power of 2.

Variable Byte Codes Variable byte codes, also known as Vbyte, Varint, or nibble codes are a useful family of codes that trade space efficiency for fast processing. They are used extensively across a variety of applications from relational database and information retrieval systems [Scholer et al. 2002, Zobel and Moffat 2006] and serialization protocols. Operating at a byte-level results in significantly faster encoding and decoding throughput compared to bit-level codes.

A variable byte code represents a positive integer x in one or more bytes. For each byte a single bit, usually the most significant bit, is used as an identifier to tell the decoder to stop or continue decoding the current integer value. The remaining lower bits of the byte are used to store the binary representation of an integer, seven bits at a time. We can store the integer values $0 \leq x < 2^7$ in one byte, $2^7 \leq x < 2^{14}$ in two bytes, and so on. Variable byte coding uses $\lfloor \log_{128}(x) \rfloor + 1$ bytes to represent an integer x.

For example, the variable byte code for the value 42 is $\underline{0}0101010$. The underlined 0 bit tells the decoder to stop. To encode the value 142 we require two bytes, $\underline{1}0001110$ $\underline{0}0000001$. The first byte contains a continue bit and the first seven bits of the integer. The second byte contains a stop bit and the remaining bits of the integer. Note that to construct the final value the continue codes must be removed and each subsequent byte must be shifted into its correct position. In this case, the second byte is shifted 7 bits to the left, resulting in 10001110.

A nibble code is a generalization of variable byte codes where the size of the bits used to store the code is parameterized. For example, reducing the size of the block to a 4 bit nibble, or to increase it to 16 bits. A nibble code uses $\lfloor \log_R(x) \rfloor + 1$ where $R = 2^{n-1}$ and n is the size of the nibble in bits.

Simple9 [Anh and Moffat 2005] Simple9 is a word-aligned integer code. This scheme is particularly useful for compressing large arrays of small integer values, for example, a gapped posting list from an inverted file, and is efficient in practice. The basic idea is to pack groups of integers into 32-bit words. For each 32-bit word, 4 high bits are used as a selector and the remaining 28 bits are used to store integers. The value of the selector is used to determine how the codes are represented. In Simple9 there are 9 different coding schemes, for example, 28 groups of 1-bit integers or 9 groups of 3-bit integers (containing one wasted bit), see Anh and Moffat [2005] for further detail. [Zhang et al. 2008] identified that 9 coding schemes in a 4 bit selector is wasteful, furthermore, some coding schemes contained unused bits. They proposed Simple16 an extension to Simple9 that specifies 16 efficient coding scheme to avoid redundant bits. Recently Anh and Moffat [2010] extended their work to use 64-bit words and observed a dramatic improvement when targeting x86_64 architecture.

2.4 Summary

In this chapter we gave an overview of the core concepts, algorithms and data structures for text compression. This began with a discussion of the pattern matching problem, text indexing, and details of two classic data structures, the suffix tree and suffix array, both of which have utility not only in pattern matching, but also for text compression and a wide variety other problems. This was followed by a brief discussion of compact data structures and compressed full-text indexes. We then discussed text compression focusing primarily on LZ-based adaptive dictionary methods, which forms a foundation for the work throughout the rest of this thesis. Finally, we discussed coding methods. First we described two statistical coding techniques, Huffman coding and arithmetic coding, which are slow in practice, but achieve compression close to empirical entropy given an accurate model. We then covered integer coding techniques which are useful when we do not have much information about the input or when the source alphabet is unbounded.

CHAPTER 3

Efficient Storage and Retrieval of Web Collections

Storage of digital collections is arguably one of the most challenging problems of the information age [Berman 2008]. Compression plays a central role and is a fundamental component of any information retrieval system [Witten et al. 1999, Ziviani et al. 2000, Manning et al. 2008, Büttcher et al. 2010, Croft et al. 2010]. Compression improves both search and retrieval by reducing the effect of disk-seek time and read latency, thus increasing bandwidth between levels of the memory hierarchy [Zobel and Moffat 1995a, Scholer et al. 2002, Büttcher and Clarke 2007]. In the context of text retrieval, a compression algorithm first must maintain a compact representation of the collection. Second, it must provide fast random access to specific documents of the collection for retrieval and post-processing tasks, including batch tasks, such as indexing and processing, or query-biased snippet generation [Tombros and Sanderson 1998, Turpin et al. 2007, Tsegay et al. 2009]. Generally, the time it takes to decompress a document is far more important than the initial compression time, as a document will usually be encoded once but decoded many times. However, the compression algorithm must at least be practical and scalable, that is, it must be capable of compressing collections orders of magnitude larger than primary memory in reasonable time and simultaneously provide efficient compression. With the recent rapid growth in digital collections, corpus compression is as important a challenge as ever.

A standard approach to document compression is to store groups of documents in fixed-size blocks, and then compress each block with a general purpose compression library, such as ZLIB. This approach implies a classical trade-off between space and time. Using a small block means that there is less data available for the compressor to learn about the redundancy present, and thus compression is less effective. If a larger block is used retrieval speed is compromised, as half of the block must be decoded on average for access to an individual document.

Table 3.1: An illustration of a fixed size adaptive dictionary compressor failing to capture non-local redundancy. The window of size 4 is contained in the box. The text to compress begins directly after the window. An example of global redundancy is highlighted in grey.

. . . abaababaabaababbbaabbbbabb $\boxed{\texttt{abba}}$ abaababaabaaba . . .

Conventional adaptive compression methods exploit local redundancy in a text by encoding its input relative to a sliding window of previously encoded substrings. This window is usually small or at least bound by primary memory and, as a consequence, it does not accurately capture any global redundancy present in a collection. An illustration of this behavior is shown in Table 3.1. A stream of text is in the process of being compressed using a fixed size sliding window of size 4. This window currently contains the substring abba. The text that we are to compress, the look-ahead buffer, begins directly to the right of the window and currently begins with the substring abaa. The next factor to be computed would be the pair (0,2), denoting that a shared substring occurs in look-ahead buffer and the window at position 0, of length 2, corresponding to the substring ab. Because of the restricted window size, adaptive algorithms fail to detect that a longer substring in the look-ahead buffer abaababaabaaba, which is much larger than the sliding window, has already occurred in the text (highlighted in grey). While this is somewhat of a trivial example, consider if the window was now 4 GB, and that the document collection was 400 GB. Even with a window of this size a great deal of duplication may fall outside of it. Given the disparity in size of the collection with respect to the dictionary window this is entirely plausible, for example, a web crawl could store multiple copies of a large website, that is, mirrored sites that are hosted on different domains. Furthermore, a number of news sites could report exactly the same article from a content distributor such as the Associated Press. Another example of global redundancy in web content is pages that share similar boilerplate style sheet markup or javascript. If such redundancy is stored in separate blocks before compression or placed outsize of an adaptive compressors search window, compression will not be as effective. Where data that can be sorted in a way such that similar documents are adjacent, for example sorting by URL [Ferragina and Manzini 2010], existing block-oriented methods can yield better compression at the cost of extremely slow retrieval.

In this chapter we propose a novel yet straightforward solution to exploit non-local redundancy. We build a representative sample of the collection and use it as a dictionary in an LZ-like encoding of the rest of the collection – where each document is compressed relative to the dictionary. The dictionary size is a parameter, but, as we show compression is effective when the dictionary only occupies a small fraction of current desktop memory. We demonstrate that using a dictionary as small as 0.1% of the collection size our

Table 3.2: An example of the relative Lempel-Ziv factorization given a dictionary (top), a string to encode (middle), its factors (position/length pairs), and, their corresponding substrings (bottom).

	1	2	3	4	5	6	7	8
$d[i]$	c	a	b	b	a	a	b	b

... b b a a n c a b b b b a a b c a ...

(3, 4) (n, 0) (1, 4) (3, 5) (1, 2) $\rightarrow$ (bbaa) (n) (cabb) (bbaab) (ca)

algorithm provides much better compression than existing approaches as well as achieving dramatically faster decompression and random access speed. The specific focus of our compression scheme is on methods that provide fast decompression for use in batch processing tasks but also allow for reasonably efficient random access to arbitrary documents in the compressed collection. In Section 3.1, we introduce our compression scheme. Then, in Section 3.1.3 we propose a simple yet highly effective method for generating a dictionary that accurately represents a large document collection and successfully captures much of its global repetitiveness. In Section 3.1.5 we discuss practical methods for compression of the LZ factors out method produces. We empirically evaluate our approach in Section 3.3 against current standard block oriented baselines and conclude in Section 3.5.

3.1 Compression

We now present our compression scheme. The algorithm resembles a traditional LZ77 factorization, where strings are encoded in terms of previously occurring substrings. However, we perform an LZ factorization against a pre-defined set of sub-strings or dictionary. We call this a relative Lempel-Ziv factorization (RLZ). In the next section we formally describe the algorithm, then we discuss dictionary generation techniques and practical methods for representing factors, which altogether provide efficient compression and effective document retrieval.

3.1.1 Relative Lempel-Ziv Factorization

At the core of our compression scheme is relative Lempel-Ziv factorization [Ziv and Merhav 1993, Kuruppu et al. 2010]. Let $x = x[1..n]$ be a string of length n, and a dictionary $d = d[1..m]$ be a string of length m, where $m \leq n$. The relative Lempel-Ziv factorization of x with respect to d, is a set of z substrings, $x = w_1 w_2..w_z$, such that each substring w_j, $j \in 1..z$, is either:

1. The longest factor, i.e., substring, of d starting at the current position in x; or
2. a single character c in x, that does not occur d.

Each factor w_j is represented as a pair (p_j, l_j), where p_j specifies an offset to a position in the dictionary d and l_j denotes the length of the factor in d. When a character c in x does not occur in d we use a special pair representation where the position field, p_j, stores the missing character and its length value, l_j, is set to 0 indicating that there was no match. As an example, the relative Lempel-Ziv factorization of the string, bbaancabbbbaabca with respect to a dictionary cabbaabba is shown in Figure 3.2. Five factors are computed. The first factor, (3, 4), corresponds to the substring bbaa, at offset 3 and length 4 in d. The second factor is (n, 0), as the character n does not exist in the dictionary. Then follows (1, 4), the substring cabb beginning at offset 1, and finally, the factors, (3,5) and (1,2) corresponding to the substrings bbaab and ca at offsets 3 and 1 respectively.

3.1.2 General Overview

Our compression scheme operates as follows.

1. We construct a dictionary, d, of total length m characters, by concatenating a selective sampling of substrings from documents in a collection. The size of m is dictated by the user and/or the available primary memory. Dictionary generation is discussed in Section 3.1.3.
2. For each document, x, in the collection we factorize x relative to d into factors (or pairs) denoted (p, l), and encode each pair efficiently. Section 3.1.5 outlines a number practical compressed pair representations offering different space-time trade-offs.
3. We store a document map which provides the position on disk of each encoded document. This component is common to all large scale document retrieval systems.
4. To access a desired document we first locate the beginning of the document and number of compressed factors using the document map, and then decode the (p, l) pairs, translating each factor into text via the dictionary, d.

Random access is achieved as the dictionary is no longer adaptive, and can be made small enough to be held in memory. Two of the most important aspects to consider during encoding are dictionary generation and pair representation. For effective compression the dictionary must capture the overall structure of the collection – representing its globally repetitive properties. Furthermore, each pair must be encoded in a compact form and support fast decoding and random access. In the next sections we describe a simple yet highly effective dictionary generation technique then examine efficient methods for representing factors.

3.1.3 Dictionary Generation

Before we perform a relative LZ factorization we must first construct a dictionary. It is critical that the contents of the dictionary represent the complete collection reasonably well. The goal is to capture global repetition across the collection that semi-static and adaptive algorithms fail to detect, either due to their block-oriented nature or limitations on window size. This poses a significant challenge as we aim to compress collections that are much larger than the physical memory of a typical server. A naive approach such as processing the collection and recording the most frequently occurring n-grams or substrings would be unfeasible as space usage would rapidly exceed memory. Substrings and statistics could be stored in a disk based index, however, we would face similar memory issues and there would be a significant impact on run time.

We found the following approach to be highly effective. We treat a collection as a single string and extract evenly spaced samples (substrings) across the collection. For a collection string, $x = x[1..n]$ of length n, we wish to generate a dictionary, $d = d[1..m]$ of length m, using samples of length s. That is, we take m/s samples at positions $0, n/(m/s), 2n(m/s), ...$ We expect that if a collection is comprised of similar documents, for example, a web crawl, any sufficiently frequent material in a collection would be captured during this sampling process and should generate an effective representation of a collection. In Section 3.3 we shall see in that this technique does indeed generate a very effective dictionary for typical web data.

An undesirable aspect of this sampling method is that the dictionary is highly likely to contain redundant information, that is, many samples will share similar content or substrings. We explore and address this issue in the next chapter.

3.1.4 Compression Algorithm

We can compute the RLZ factorization in $O(n \log m)$ time and $O(m)$ words of memory, using a variation of CSP2 for traditional LZ77 factorization [Chen et al. 2008]. The main idea is to construct the suffix array of the dictionary and use it as an index to parse each input document into factors. As described in Section 2.2.2, the suffix array $\mathrm{SA}[1..n]$ of a text $x = x[1..n]$ is an array of pointers to all the suffixes of x arranged in lexicographic order, such that $x[\mathrm{SA}[i]..n] < x[\mathrm{SA}[i+1]..n]$. For every substring of x, $x[i..j]$, there exists an interval in SA_x, $SA_d[lb..rb]$, such that $\mathrm{SA}_x[lb], \mathrm{SA}_x[lb+1], \mathrm{SA}_x[..], \mathrm{SA}_x[rb]$ contains positions to every occurrence of $x[i..j]$ in x. As the suffixes are ordered lexicographically the interval boundaries lb and rb can be calculated with successive binary searches. The

Table 3.3: An example demonstrating the Refine function where we are searching for the current prefix of x in dictionary d. To achieve this we refine the range $SA_d[lb..rb]$ such that the longest prefix of x is found in d. In this example we output the factor (3, 4), corresponding to a match in the dictionary at position 3 and length 4.

i	1	2	3	4	5	6	7	8	9
d[i]	c	a	**b**	**b**	**a**	**a**	b	b	a
SA$_d$	9	4	8	6	2	3	7	5	1

i	d[SA$_d$[i]] .. d[m]								
1	a								
2	a	a	b	b	a				
3	a	b	b	a					
4	a	b	b	a	a	b	b	a	
5	**b**	a							
6	**b**	a	a	b	b	a			
7	**b**	**b**	a						
8	**b**	**b**	**a**	**a**	b	b	a		
9	c	a	b	b	a	a	b	b	a

j	1	2	3	4	5	6	7	8	9
x[j]	**b**	**b**	**a**	**a**	n	c	a	b	b
lb	5	7	8	8	-1				
rb	8	8	8	8	-1				

pseudo-code in Algorithm 1 outlines the factorization procedure. Given a string, x, and a dictionary, d, the function RLZEncode computes a factorization of x relative to d. This is achieved with successive calls to Factor, where we use the suffix array of the dictionary, SA$_d$, for efficient matching. To facilitate document retrieval we stop factorization each time we encounter a document boundary, returning the current (pos, len) pair. Furthermore, we maintain a document map indicating the offsets of each document boundary in the encoding, which is used to provide random document access.

Refine calibrates the left and right boundary of suffix array, SA$_d[lb..rb]$, such that the suffixes of length l in the interval between these bounds matches the current prefix in x, $x[i..i + l]$. The length of the match increases with each successful call to Refine. As the suffix array is ordered lexicographically, each boundary or edge of the interval can be calculated using a binary search. An example of the factorization process is demonstrated in Table 3.3. The dictionary, d = cabbaabba, and input sequence, x = bbaancabbbbbaabca, from Table 3.2 are used. Initially, we compute the suffix array of d, SA$_d$. Factorization begins from the first position in the input sequence. We call Factor$(1, x, d)$. The first call to Refine returns the interval $(5, 8)$. The first character of the suffixes between this interval match the first character in x. The second call to Refine returns the interval $(7, 8)$. The suffixes in this interval match the first two characters in x. The longest match is 4 characters and occurs in the interval $(8, 8)$. The value of $\mathrm{SA}_d[8]$ contains the suffix position in d where the match occurred, in this case 3. The length of the match is 4, so Factor will return the pair $(3, 4)$ and we call factor with the new offset Factor$(5, x, d)$.

Note that when we break from the while loop on line 5 of the function Factor there will be $rb - lb + 1$ occurrences of the current shared factor. That is, each index value in the range $lb..rb$ will correspond to a position in d that matches the current prefix of the input stream of length $j - i$. In the example above there was only one result, however, in practice this range will be larger and one must consider what is the most appropriate index in d to select. In our implementation we opted to always select the left most index, $\mathrm{SA}_d[lb]$, however, considering the size of the dictionary one could avoid costly lower level cache misses by selecting the index closest to the position of the previously encoded factor.

Algorithm 1 RLZEncode performs a relative LZ factorization of the string x with respect to d. The output is a set of (pos,len) pairs.

```
1: function RLZEncode($x$, $d$)
2:     $i \leftarrow 1$
3:     while $i \leq len(x)$ do
4:         $(pos, len) \leftarrow$ Factor($i$, $x$, $d$)
5:         output $(pos, len)$
6:         if $len = 0$ then
7:             $i \leftarrow i + 1$
8:         else
9:             $i \leftarrow i + len$
1: function Factor($i$, $x$, $d$)
2:     $lb \leftarrow 1$
3:     $rb \leftarrow len(d)$
4:     $j \leftarrow i$
5:     while $j \leq len(x)$ do
6:         if $lb = rb$ and $d[SA_d[lb] + j - i] \neq x[j]$ then
7:             – The current character in $d$ does not match $x[j]$
8:             – so we can no longer refine the interval in $SA_d$
9:             break
10:        $(lb, rb) \leftarrow$ Refine($lb$, $rb$, $j - i$, $x[j]$)
11:        if $(lb, rb)$ is no longer a valid interval then
12:            break
13:        $j \leftarrow j + 1$
14:        if x[j] is at a document boundary then
15:            break
16:    if $j = i$ then return $(x[j], 0)$
17:    else return $(SA_d[lb], j - i)$
```

Table 3.4: Average factor length and percentage of unused bytes in an RLZ dictionary for varied dictionary and sample sizes built on a 426 GB GOV2 corpus.

Size (GB)	Samp. (KB)	Avg.Fact.	Unused (%)
2.0	0.5	46.01	39.62
2.0	1.0	46.83	24.28
2.0	2.0	46.77	28.10
2.0	5.0	46.09	20.65
1.0	0.5	41.30	36.00
1.0	1.0	41.80	31.38
1.0	2.0	41.62	25.66
1.0	5.0	40.93	17.84
0.5	0.5	37.07	32.91
0.5	1.0	37.35	28.64
0.5	2.0	37.15	23.65
0.5	5.0	36.45	16.20

Table 3.5: Average factor length and percentage of unused bytes in an RLZ dictionary for varied dictionary and sample sizes built on a 256 GB Wikipedia corpus.

Size (GB)	Samp. (KB)	Avg.Fact.	Unused (%)
2.0	0.5	38.70	27.34
2.0	1.0	39.11	21.33
2.0	2.0	39.13	17.29
2.0	5.0	38.97	12.22
1.0	0.5	34.54	23.72
1.0	1.0	34.85	18.52
1.0	2.0	34.81	13.99
1.0	5.0	34.63	9.56
0.5	0.5	31.05	21.15
0.5	1.0	31.22	15.83
0.5	2.0	31.17	11.53
0.5	5.0	30.96	7.41

3.1.5 Pair Representation

Efficient encoding of the (p, l) pairs is a critical component of the compression scheme, for which we explored several approaches. In practice we encode the two components of a pair separately. We observed that the position values had no significant skew in distribution to exploit using common compression methods. That is, in each pair the p values appear to be spread randomly across the dictionary, and are therefore difficult to compress. We represent each p value as a single unsigned 32-bit integer and concentrated on finding an efficient encoding for the length elements (the l values). Results in Table 3.4 and Table 3.5 show the average factor length recorded for varied combinations of dictionary size and sampling size across two document collections. Note that the average factor length remained relatively stable across all runs, ranging from 30 to 40 characters. Furthermore, we observed that a significant percentage of length values in an encoding were always less than 100, and usually no greater than the sample size used to generate the dictionary. This is illustrated in Figure 3.1, which plots histograms of encoded length values for a factorization of the GOV2 collection using a 0.5 GB dictionary and varying sample periods. Note that factor lengths are restricted to the length of the sample used when generating each dictionary. This can be observed in Figure 3.1 where each dictionary sampling plots a vertical line of points at its sample length. Note that the dictionary where 512 byte samples were used there are a number of points where the length values are greater than its sample length. Examining this group of factors showed that these samples were concatenated groups of white space or junk characters.

Observe that, irrespective of the sample period the bulk of length values remain small. In light of this we used a variable byte (Vbyte) code [Williams and Zobel 1999, de Moura et al. 2000] to encode length values, which provides a reasonable trade-off between compression and decoding speed [Scholer et al. 2002, Trotman 2003]. Using Vbyte, the majority of length values are encoded in just a single byte. Representing both position and length values as byte oriented codes provides a great advantage during decoding – as costly mask and shift operations required by bit oriented codes are avoided.

Closer inspection of the position values revealed that while the distribution of the p values across the entire collection was rather flat, applying a general-purpose compressor (ZLIB) to the p values for each document separately gave a significant boost to compression, suggesting that the p values within each document can be quite skewed. This effect is manifested by substrings that are repeated within a file, but not present in the dictionary. These substrings get factorized into the same set of pairs, which are then repeated in the document's RLZ factorization. Applying a local compressor to the pairs captures these local repetitions and improves overall compression. We observed the same phenomenon in the length values. That is, they contained higher-order patterns at a document level.

3.1.6 Dynamic Document Databases

A further virtue of our method is its application in a dynamic environment where documents are appended to the collection over time. Due to the nature of our sampling process, as long as additional documents maintain similar characteristics to the initial collection there will be little impact on compression effectiveness.

If per-document compression degrades below a specific threshold there are several ways to compensate. If there is no constraint on memory, we can sample the new documents and append them to the dictionary. This method avoids an expensive re-encoding process as the previous pair codes are still valid. The suffix array will need to be recomputed in order to include the new samples during factorization, however, this takes

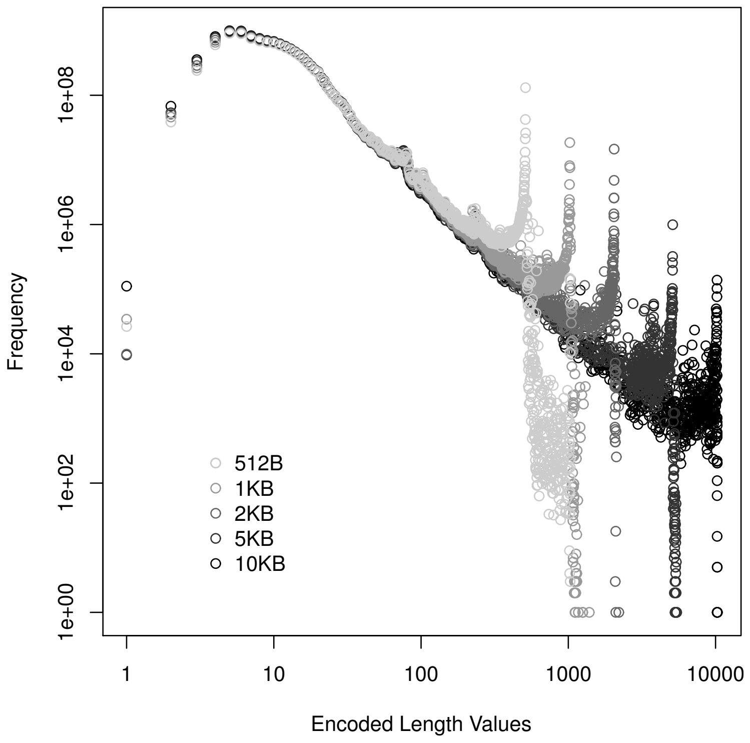


Figure 3.1: Frequency histogram of length values in an RLZ encoding of the 426 GB GOV2 corpus using a 0.5 GB dictionary and varied sample periods.

relatively little time. If there are constraints on memory, the dictionary can be regenerated taking the additional documents into consideration. This invalidates the original encoding, and, as a consequence, the collection will need to be compressed again.

3.2 Decompression

The decompression algorithm is extremely efficient. No auxiliary data structures are required, only the compressed file and its dictionary. If random document retrieval is desired a document map is also required, however, its size is trivial compared to both the encoding and dictionary. We give the pseudo-code for decoding in Algorithm 2. We decode the collection a single pair at a time. If the length component of the pair is set to 0 then the position value contains a single character that is not found in the dictionary. Otherwise, the position value corresponds to an offset in d for which we output the substring $d[pos..pos + len)$. To extend the algorithm to support random document retrieval we add an initial seek operation, which moves to the position in the compressed collection of the first pair of the desired document and then begin decoding. As the whole dictionary is resident in memory and no other work is required (for example, maintaining an adaptive dictionary and needing to recompute it at regular intervals or including sequence points) decoding is tremendously fast.

Algorithm 2 RLZDecode decodes a relative Lempel-Ziv encoding e with respect to a dictionary d. Pairs are decoded on at a time and correspond to substrings in d.

```
1: function RLZDecode($e$, $d$)
2:     for ($pos$, $len$) $\in e$ do
3:         if $len = 0$ then
4:             output $pos$
5:         else
6:             output $d[pos \mathrel{..} pos + len)$
```

3.3 Experiments

In this section we empirically evaluate RLZ, comparing it to five baseline algorithms that are commonly used in document storage and retrieval systems all offering a variety of space-time trade-offs.

3.3.1 Method

To simulate various aspects of a document retrieval system, two access patterns were used throughout our experiments. First, we used a sequential list of 100,000 document IDs to simulate requests from large-scale batch processing systems. Second, to simulate the typical behavior of a document retrieval system we generated a list of 100,000 document IDs from the ranked output of real queries into a search engine. Each collection was indexed using the Zettair search engine,[1] then queried using default settings and a log sourced from topics 20,001 to 60,000 from the 2009 Million Query Track.[2] The top 20 document IDs for each query were concatenated to a list and capped at 100,000.

3.3.2 Systems Tested

The first baseline is simply a raw concatenation of uncompressed documents with a map specifying offsets to each document location. The next group of baselines use a standard block oriented approach where documents are grouped into fixed-size blocks and compressed with a general purpose adaptive algorithm. We use ZLIB[3] and LZMA[4] as these were the two best systems reported in the extensive study by Ferragina and Manzini [2010]. On the other side of the adaptive spectrum we include Snappy[5] and LZ4,[6] both byte-oriented adaptive compressors, which are commonly used in massively distributed storage systems, offering efficient decoding at the cost of compression effectiveness. Block

[1]http://www.seg.rmit.edu.au/zettair
[2]http://trec.nist.gov/data/million.query09.html
[3]http://www.zlib.net
[4]http://www.7-zip.org/sdk.html
[5]http://code.google.com/p/snappy/
[6]https://code.google.com/p/lz4/

sizes for all baselines begin as a single document per block, identified as 0.0 MB, and increase in size from 0.1 MB, 0.2 MB, 0.5 MB to 1.0 MB.

RLZ runs are identified by their dictionary size and the (position, length) coding schemes that were used to compress each document. Methods used were Z, ZLIB with Z_BEST_COMPRESSION enabled, V, variable byte coding and U, unsigned 32 bit integers. Dictionary sizes used in the evaluation section were 0.5 GB, 1.0 GB and 2.0 GB. Unless stated otherwise, all RLZ dictionaries were generated from 1 KB samples. To evaluate the performance of our method in a dynamic environment we simulate collection update behavior by generating dictionaries from fixed prefixes of a collection. We then use these dictionaries to compress the complete collection, observing any impact on compression.

3.3.3 Test Collections

Two document collections were used. TREC GOV2 is a 426 GB web crawl of the .gov top level domain in 2004. This consists of roughly 25 million documents, with an average document size of 18 KB. The second collection is a 256 GB English Wikipedia snapshot sourced from Clueweb09,[7] consisting of approximately 6 million documents and an average document size of 45 KB. Experiments were conducted on both collections sorted in natural web crawl order.

3.3.4 Environment

All document retrieval experiments were conducted on an Intel Xeon 3.0 GHz processor with 4 GB of main memory. The disk was a Seagate Scandisk II, 1Tb, 7200 RPM, with 32 MB cache. The operating system was Red Hat Enterprise Linux Server release 5.5 (Tikanga), running Linux kernel version 2.6.18. The compiler used was GCC 4.1.2 with full optimizations. All time results were recorded as wall clock time. As the compressed collections used for evaluation were significantly larger than internal memory it is important to account for disk seek and read latency as they are the dominant cost in document retrieval. We ensured each collection was the only one present on the disk for each run, to eliminate disk position bias. Caches were dropped between each run with `sync && echo 3 > /proc/sys/vm/drop_caches`. No other processes were running during each experiment. We define compression ratio as a percentage of the encoded output against the original collection size.

[7] http://lemurproject.org/clueweb09

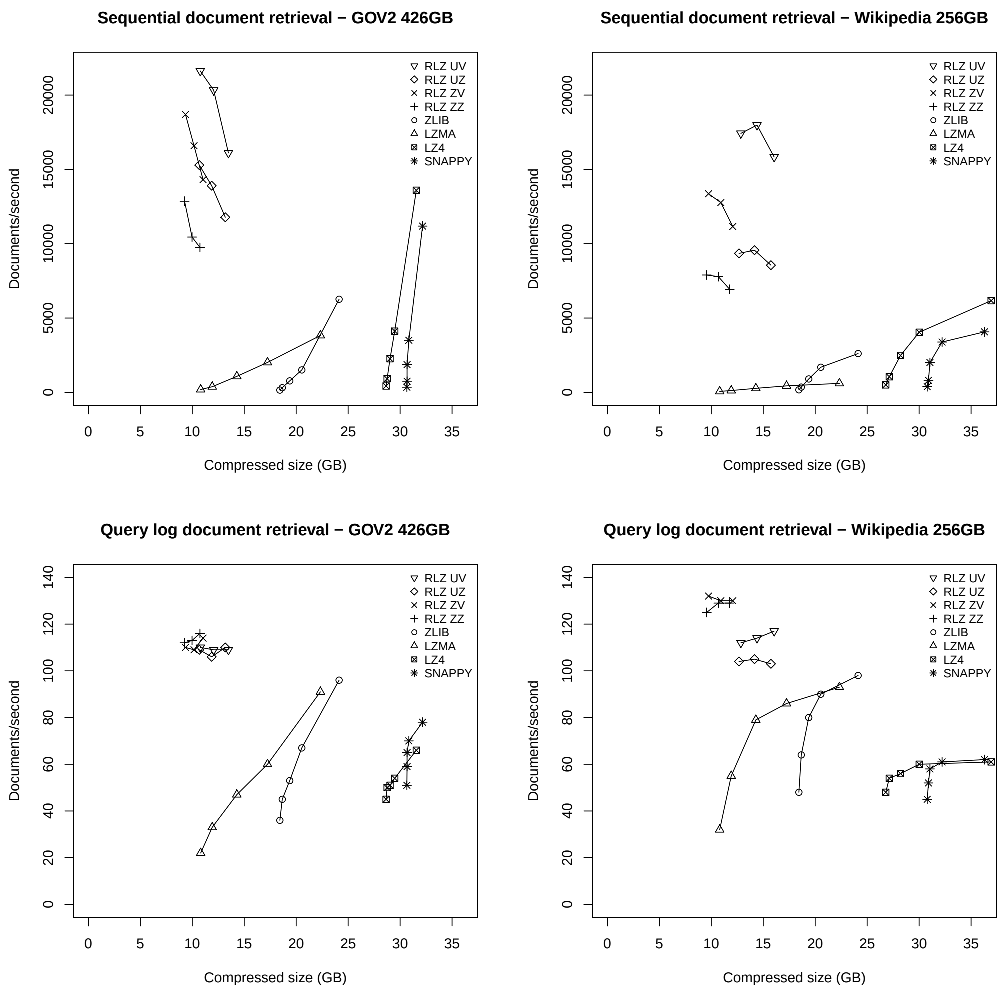


Figure 3.2: Compressed size against documents retrieved per second for sequential (top) and query log (bottom) document retrieval requests on GOV2 (left) and Wikipedia (right) collections for varied RLZ pair combinations and baseline block sizes. RLZ runs used 0.5, 1.0 and 2.0 GB dictionaries. Baseline runs used 0.0, 0.1 MB, 0.2 MB, 0.5 MB and 1.0 MB blocks.

3.4 Discussion

Compression statistics and document retrieval times for RLZ and block-oriented baselines are shown in Figure 3.2 and Tables 3.6 to 3.9. RLZ clearly outperforms all baselines in terms of time and space for both sequential and query-log document request scenarios. Comparing cases with similar memory requirements and compression effectiveness, for sequential access our RLZ approaches a thousand times the speed of the competitor methods. Excepting cases where the compression achieved by the competitor methods is particularly poor, the sequential speed of RLZ is generally at least ten times greater, and the random-access speed is always better by a significant margin. LZ4 and Snappy compressing single documents per block gave competitive sequential decoding speeds compared to RLZ's slowest coding scheme, ZZ, however, both achieved significantly worse compression results, a difference of 20% compression effectiveness on average.

The effectiveness of RLZ compression validates our dictionary sampling hypothesis that we are capturing global repetitive properties of a collection that the baselines cannot detect. A key factor attributing to the performance of RLZ decoding speed is that the dictionary is static and present in memory. Decoding can start immediately. The compressed baselines incur a penalty initializing a new dictionary for each document request. The baselines are subject to the further penalty of having to decode at least half a block on average to retrieve a document. Ordered document requests have much faster decoding rates due to sequential disk access. UV pair coding was the fastest method due to its cheap decoding procedure. ZZ was the slowest method, but it was still faster than all baselines and achieves excellent overall compression at 9.26% using a 2.0 GB dictionary.

Query log requests were much slower than sequential requests due to latency during disk operations. Focusing on the compressed baselines, the fastest throughput was achieved by each baseline implementation where single documents were encoded in each block. This was expected because there was no additional overhead when decoding a document. At the same time, the single document methods were the largest of the block-oriented encodings as there was less redundancy to exploit. This mirrors results reported by Ferragina and Manzini [2010]. Figure 3.2 clearly outlines the differences between the two types of baseline compressors. ZLIB and LZMA, both implementing bit-oriented adaptive compression schemes achieve better compression at the cost of decoding throughput. Furthermore, LZ4 and Snappy, which use byte-oriented adaptive schemes sacrifice compression effectiveness for improved sequential decoding speeds. Although LZ4 and Snappy clearly give a significant improvement in sequential decoding throughput compared to the other two general-purpose baselines (ZLIB and LZMA), random access speeds were actually slower. Compressing documents in larger blocks, for example, 0.5 MB and 1.0 MB gave speeds similar to that of ZLIB and LZMA, however, on smaller blocks we observed that they achieved much slower speeds, up to 30 documents per second less on average. After close examination of the each algorithms reference implementation it was found that LZ4 and Snappy both incur a large initialization penalty before decoding starts compared to ZLIB and LZMA.

A larger dictionary was beneficial for ordered document requests on both collections, but there was no clear benefit to the use of a larger dictionary for query-log document requests. All RLZ methods had consistent access speeds, averaging over 100 documents per second. ZZ and ZV pair coding methods ran slightly faster on the Wikipedia collection. We attribute this to Wikipedia's average document size being much larger, and ZLIB being able to compress the pairs more effectively.

Results in Table 3.10 demonstrate that our algorithm responds well in a dynamic environment, where new documents are added to the collection. In our simulation we generated 1.0 GB dictionaries from 90% to 1% prefixes of GOV2 and Wikipedia. We observed less than 1% difference in compression relative to the original dictionary. Indeed, the loss when using a dictionary from a 1% prefix of Wikipedia was only a 1.35% reduction in compression effectiveness. This demonstrates that RLZ should provide a highly robust compression method in the presence of dynamic updates to a document database system.

3.5 Summary

In this chapter we described an efficient compression scheme capable of scaling to large real-world text collections. RLZ provides both highly effective compression and fast sequential decoding and random access to individual documents. We proposed a dictionary generation technique which although simple, successfully captures global repetitive properties of a collection and provides excellent overall compression. We empirically demonstrated that our algorithm can dramatically outperform state-of-the-art block-oriented techniques, primarily because it is able to capture global repetition in large collections, which block-oriented techniques and general-purpose adaptive compressors inherently miss. We also demonstrated that RLZ works well in a dynamic environment where the collection is regularly updated. An additional virtue of RLZ is its scalability: it is lightweight at compression time, both in principle and in practice. An undesirable side effect of the sampling technique is a high percentage of redundancy exists throughout a dictionary, especially on highly repetitive collections. In the next chapter we explore this issue and outline methods of redundancy elimination in a RLZ dictionary at various stages of compression.

Table 3.6: Sequential and Query-log retrieval speed in documents per second on a 426 GB GOV2 corpus for varied combinations of RLZ dictionaries sizes and position–length codes.

Size (GB)	Pos–Len	Enc. (%)	Sequential	Query Log
2.0	ZZ	9.26	12,857	112
1.0	ZZ	9.98	10,449	113
0.5	ZZ	10.74	9,752	116
2.0	ZV	9.35	18,694	110
1.0	ZV	10.17	16,591	109
0.5	ZV	11.04	14,310	114
2.0	UZ	10.68	15,288	109
1.0	UZ	11.87	13,902	106
0.5	UZ	13.18	11,779	110
2.0	UV	10.77	21,622	110
1.0	UV	12.06	20,327	109
0.5	UV	13.48	16,107	109

Table 3.7: Sequential and Query-log retrieval speed in documents per second on a 426 GB GOV2 corpus for baseline ASCII and blocked LZ files.

Alg.	Block (MB)	Enc. (%)	Sequential	Query Log
ascii	-	100.00	8,982	28
ZLIB	0.0	24.13	6,263	96
ZLIB	0.1	20.54	1,509	67
ZLIB	0.2	19.38	773	53
ZLIB	0.5	18.66	313	45
ZLIB	1.0	18.43	153	36
LZMA	0.0	22.33	1,490	91
LZMA	0.1	17.24	338	60
LZMA	0.2	14.29	180	47
LZMA	0.5	11.92	78	33
LZMA	1.0	10.81	41	22
LZ4	0.0	31.56	13,595	66
LZ4	0.1	29.48	4,118	54
LZ4	0.2	29.03	2,262	51
LZ4	0.5	28.75	918	50
LZ4	1.0	28.65	427	45
Snappy	0.0	32.16	11,185	78
Snappy	0.1	30.84	3,504	70
Snappy	0.2	30.67	1,868	65
Snappy	0.5	30.67	749	59
Snappy	1.0	30.65	346	51

Table 3.8: Sequential and Query-log retrieval speed in documents per second on a 256 GB Wikipedia corpus for varied combinations of RLZ dictionaries sizes and position–length codes.

Size (GB)	Pos–Len	Enc. (%)	Sequential	Query Log
2.0	ZZ	9.56	7,898	125
1.0	ZZ	10.68	7,786	129
0.5	ZZ	11.77	6,932	129
2.0	ZV	9.74	13,360	132
1.0	ZV	10.92	12,766	130
0.5	ZV	12.07	11,156	130
2.0	UZ	12.67	9,351	104
1.0	UZ	14.16	9,563	105
0.5	UZ	15.74	8,557	103
2.0	UV	12.85	17,422	112
1.0	UV	14.40	17,979	114
0.5	UV	16.05	15,834	117

Table 3.9: Sequential and Query-log retrieval speed in documents per second on a 256 GB Wikipedia corpus for baseline ASCII and blocked LZ files.

Alg.	Block (MB)	Enc. (%)	Sequential	Query Log
ascii	-	100.00	2,093	50
ZLIB	0.0	24.13	2,610	98
ZLIB	0.1	20.54	1,690	90
ZLIB	0.2	19.38	902	80
ZLIB	0.5	18.66	355	64
ZLIB	1.0	18.43	172	48
LZMA	0.0	22.33	604	93
LZMA	0.1	17.24	437	86
LZMA	0.2	14.29	271	79
LZMA	0.5	11.92	123	55
LZMA	1.0	10.81	65	32
LZ4	0.0	36.93	6,168	61
LZ4	0.1	30.01	4,044	60
LZ4	0.2	28.21	2,485	56
LZ4	0.5	27.13	1,049	54
LZ4	1.0	26.79	502	48
Snappy	0.0	36.29	4,070	62
Snappy	0.1	32.21	3,386	61
Snappy	0.2	31.04	2,008	58
Snappy	0.5	30.90	813	52
Snappy	1.0	30.78	381	45

Table 3.10: Simulating a dynamic document database by compressing 426 GB GOV2 and 265 GB Wikipedia corpus using ZZ pair codes relative to 1 GB dictionaries built from varied prefixes of the collection.

Prefix %	GOV2 Enc. %	WP Enc. %
100.0	9.89	10.68
90.0	9.92	10.70
80.0	10.41	10.73
70.0	10.79	10.76
60.0	10.88	10.89
50.0	11.10	10.11
40.0	11.17	11.11
30.0	11.35	11.25
20.0	11.38	11.37
10.0	11.40	11.64
1.0	12.04	11.04

CHAPTER 4

Sample Selection for Dictionary Based Corpus Compression

In the previous chapter we presented a simple yet highly effective sampling technique to a generate dictionary which is suitable for efficient large-scale corpus compression. Key to the effectiveness of the method is that the dictionary forms a representative sample of the collection. The aim is to capture the global repetition across a collection that adaptive compression algorithms do not detect. To create a dictionary of size m we consider the collection as a single concatenated string of length n and take samples of lengths s at evenly-spaced intervals. That is, we take m/s samples from $n/(m/s)$ evenly spaced locations throughout the collection, on the assumption, which our experiments have confirmed, that any sufficiently frequent material in the document collection is likely to be captured in this process.

Although our sampling method successfully captures non-local duplication and provides excellent compression, there is a high volume of redundant content found throughout the dictionary. Indeed, in experiments we observed that a significant percentage of each dictionary was unused, almost 30% on average. Furthermore, there was a strong skew in the samples that were used, and even among these, there was redundancy as some samples contained repeated material. Figure 4.1 plots dictionary sample usage sorted by frequency when compressing the 256 GB Wikipedia collection with 0.5 GB, 1.0 GB and 2.0 GB dictionaries and 1 KB sample sizes. For each dictionary there was a very small set of about 200 frequently used samples that were accessed close to six million times on average. Interestingly, the content of these samples were comprised of incomprehensible strings of whitespace, HTML markup and junk text. On the other hand, the vast majority of dictionary samples were rarely used, and many were completely untouched, especially with larger dictionaries. For example, the 0.5 GB, 1.0 GB and 2.0 GB dictionaries used in Figure 4.1 had 20 KB, 50 KB and 150 KB completely unused samples respectively.

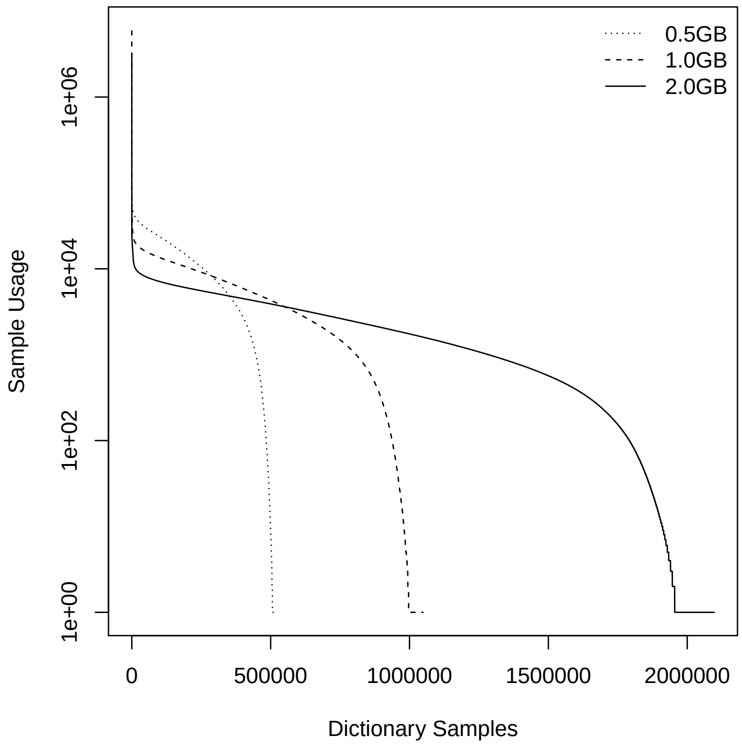


Figure 4.1: Dictionary sample usage (sorted by frequency) using 0.5 GB, 1.0 GB and 2.0 GB RLZ dictionaries with 1 KB samples on a 256 GB Wikipedia corpus.

In this chapter we present methods to identify and remove redundancy throughout a sampled dictionary. This gives the opportunity to make a number of improvements to the compression scheme during encoding and decoding stages. First we can replace unnecessary and redundant content with more appropriate samples that will improve compression effectiveness. On the other hand, such methods can be used to reduce the overall memory footprint of the compressor to facilitate use on memory-constrained devices such as phones and tablets. A further advantage of a smaller dictionary is that it improves decoding efficiency as the dictionary will exhibit improved behavior across all levels of the cache hierarchy. In Section 4.1 we outline a pre-processing method that removes long repetitive substrings from a dictionary before compression. Then, in Section 4.2 we discuss alternative methods where redundant samples and characters are removed in a post-processing stage after compression. In Section 4.3 we evaluate our methods and demonstrate that we can reduce a dictionary by 50% or more – making it less than 0.1% of the overall collection size – while having no significant effect on compression effectiveness. Finally, in Section 4.4 we conclude.

4.1 Pre-processing

As a consequence of the sampling method vast quantities of redundant information can be captured during the dictionary generation process. An example of such redundancy can be found in the Wikipedia collection that we have been using throughout our experiments. Each document is prefixed with a Web Archive Header (WARC).[1] This header provides metadata specific to the web crawl, for example, a unique document identifier and a time stamp indicating when a document was fetched. In our experiments we observed that a 2 GB dictionary using 1 KB sampling on the 256 GB Wikipedia collection sampled WARC header elements approximately half a million times. While this redundancy only equates to a small component of the dictionary, it serves as an example of global redundancy in a collection that is only needed to be stored once – and not half a million times.

Such redundancy can be removed in a pre-processing step, that is, before compression is performed using a variation of the pre-compression technique proposed by Bentley and McIlroy [2001], (BMI). They describe an LZ-style algorithm where substring matches are restricted to a minimum length. Long and possibly distant repetitive substrings are identified and replaced with a position/length code directly in the text. This is implemented by inserting a unique escape code in the text, then, efficiently encoding the position and length values with a byte-oriented code, for example, Vbyte. The output is then passed to a second stage general purpose compression algorithm such as ZLIB which encodes short and closer repetitive substrings. Naturally decoding is also a two-step process. That is, an encoding must initially be uncompressed with a general purpose algorithm, then each embedded position/length code needs to be identified and expanded in a second pass.

This method is effective when the input is highly redundant, such as collections of natural language texts or clustered documents, and was recently used in the compression scheme outlined in BigTable [Chang et al. 2008], a massively distributed storage system. The algorithm restricts substring matches to a minimum length b. That is, all matches with a length less than b are ignored. BMI hashes non-overlapping substrings of length b, that is, $x[1..b], x[b+1..2b], x[2b+1..3b]$, etc., and then scans the text searching for matches. We make a slight modification to the algorithm where we hash all overlapping substrings, that is, $x[1..b], x[2..b+1], x[3..b+2]$, and so on. This ensures that we detect all possible matches and not a smaller subset. The algorithm is used to detect redundancy throughout a sampled dictionary, however, when we identify a repetitive substring we simply remove it from the dictionary and continue processing. In Section 4.3 we show that this method is highly effective at reducing redundancy throughout a sampled dictionary and only has a minimal effect on compression.

Examples of this encoding scheme using various restricted match lengths on the string acaacacaacaacaca are shown in Table 4.1. The middle column displays the output of BMI

[1]http://www.digitalpreservation.gov/formats/fdd/fdd000236.shtml

Algorithm 3 PreProcess-BMI accepts a text x of length n and removes all repeated substrings in x with a length of at least b.

```
1: function PREPROCESS-BMI($x$, $n$, $b$)
2:     $i \leftarrow 0$
3:     initialize hash of $x[1..b]$
4:     for $j \leftarrow b$ to n do
5:         store $(hash, j - b)$
6:         update hash to include $x[j]$ and exclude $x[j - b]$
7:         if $j - b + 1 < i$ then
8:             continue
9:         $(pos, len) \leftarrow$ FindMatch(hash)
10:        if $len > b$ then
11:            output $x[i..pos)$
12:            $i \leftarrow pos + len$
```

Table 4.1: Example output of BMI and PreProcess-BMI for a variety of match lengths, b, on the string acaacacaacaacaca. The middle column displays the original BMI output (left) and each pair's corresponding substring (right). PreProcess-BMI output is shown in the right column where all repetitive substrings larger than b have been removed.

b	**BMI output**			**Output**
1	ac(0,1)(0,3)(1,5)(3,5)	$\rightarrow$	ac a aca caaca acaca	ac
2	aca(0,3)(4,2)(0,8)	$\rightarrow$	aca aca ca acaacaca	aca
4	acaac(0,6)(3,5)	$\rightarrow$	acaac acaaca acaca	acaac
8	acaacaca(0,8)	$\rightarrow$	acaacaca acaacaca	acaacaca

(left) and each pairs corresponding substring (right). The column on the right displays our desired output, where all repetitive substrings of length greater than or equal to b have been removed. Note that a pair can be self-referential. An example of this is shown in Table 4.1 where the minimum match length is restricted to 4. The first pair returned is (0,6). This indicates that a match of length 6 was found at index 0. Note that at this point the algorithm has only processed 5 characters. The sixth character is referenced from the actual match. That is, once we have exhausted the current input buffer, in this case the substring abaab at index 4, we wrap around to the beginning of the substring and resume matching characters. This process continues until we reach the desired match length. The pseudo-code for our variation of BMI is outlined in Algorithm 3. A rolling hash [Karp and Rabin 1987] is used to represent all overlapping substrings of length b in a string x of length n, storing $n - b + 1$ values in a table for fast lookup. On line 2 we initialize a pointer, i, indicating how much of the string we have processed. Line 4 begins a scan through the string searching for matching substrings of length b. Line 7 prevents matching in string that we have already processed. If the function FindMatch identifies a matching substring, we greedily expand the match forward and backward in the string.

Algorithm 4 PostProcess-IH accepts a RLZ encoding e, its dictionary d which used a block size of b for sampling. The function returns a new dictionary where half of the least-frequently used samples have been removed.

```
function PostProcess-IH($e$, $d$, $b$)
    initialize new dictionary $d'$ to null
    initialize sample usage array $C[0..|d|/b]$ to zero
    for each $(pos, len)$ factor in e do
        if $len = 0$ then
            continue
        for $i \leftarrow pos$ to $pos + len - 1$ do
            $C[i/b] \leftarrow C[i/b] + 1$
            $i \leftarrow i + b$
    sort $C$ by decreasing frequency
    for $i \leftarrow 0$ to $(|d|/b)/2$ do
        $d' \leftarrow d' + d[C[i]]$
    return $d'$
```

For example, pre-compression of the string acacacacacacacac with match length restricted to 2 would output $ac(0, 14)$, as forward expansion would match to the end of the string. On completion we compress the collection once more relative to the new, smaller and less redundant dictionary.

4.2 Post-processing

In this section we consider redundancy elimination from a dictionary after the collection has been compressed.

Our first approach, *iterative halving*, is defined as follows. We decode the collection and compute usage statistics for each dictionary sample. For each RLZ pair we increment a counter corresponding to the sample that the factor occurred in. Once the pairs have been processed we sort the samples by frequency in descending order and generate a new dictionary, half the size of its original, comprised of the most frequently used samples. Then, we compress the collection relative to the newly created dictionary. This process can be repeated until we reach a desired dictionary size or compression degrades to a preset threshold. Such an approach gives explicit control of memory use and should maximize compression effectiveness for a given number of samples. The pseudo-code for the iterative halving algorithm, PostProcess-IH, is outlined in Algorithm 4. Note that if a dictionary d of length $|d|$ used sample blocks of length b, there will be $|d|/b$ sample counters, see lines 3 and 11. Furthermore, it is possible that a factor is located across a number of samples, see lines 7 to 9.

An alternative approach is to remove redundancy at a byte level. This time, during decoding we record statistics of the individual usage of characters in the dictionary. More

Algorithm 5 PostProcess-B accepts a RLZ encoding e and its dictionary d. The function returns a new dictionary all unused bytes have been removed.

```
1: function PostProcess-B($e$, $d$)
2:     initialize new dictionary $d'$ to null
3:     initialize byte usage bitvector $BV[0..|d|]$ to zero
4:     for each $(pos, len)$ factor in e do
5:         if $len = 0$ then
6:             continue
7:         for $i \leftarrow pos$ to $pos + len - 1$ do
8:             $BV[i] \leftarrow 1$
9:     for $i \leftarrow 0$ to $|d|$ do
10:        if $BV[i] = 0$ then
11:            continue
12:        $d' \leftarrow d' + d[i]$
13:    return $d'$
```

precisely, we store a bitvector – one bit for each character in the dictionary – and set a bit to 1 if its corresponding dictionary character was used during decompression. We can then generate a new dictionary removing all unused characters, which are identified by still having a 0 bit by the end of the decoding process. Pseudo-code for the byte-level redundancy removal, PostProcess-B, is outlined in Algorithm 5.

Unlike iterative halving we do not need to re-encode the collection relative to the newly computed dictionary, instead, we can avoid the significant computational step by translating each RLZ position value to its new position using rank operations on the bitvector that was used for redundant byte removal. That is, for any position value from the original encoding, *oldpos*, we can compute its position in the new dictionary with:

$$newpos = \mathsf{rank}_1(bv, oldpos)$$

An example of this procedure is illustrated in Table 4.2 where we factorize the input string *naabcbcbc* relative to the dictionary $d = aabcanad$. After factorization we compute a bitvector, BV, where each 1 bit corresponds to a used character during encoding. The new dictionary, d', is generated by removing all unused characters from d. Then, we translate each of the original factor's position values using a rank operation on BV. For example, the first factor in the encoding occurs at position 5 in d. The rank at position 5 in BV is 3, that is, $\mathsf{rank}_1(bv, 5) = 3$, therefore, the factor is translated to (3,2), as the substring *na* now occurs at position 3 in d'. The complete translation of the encoding relative to the new dictionary d' is shown on the final line. Pseudo-code for the extended algorithm, PostProcess-BE, which includes translating the original encoding relative to the newly computed dictionary is described in Algorithm 6. Note that the ComputeRank function generates a data structure that answers rank calls on a bitvector in constant time. This can be in compressed or uncompressed form.

Table 4.2: Removing redundancy at a character level from a dictionary d (top) after compression in a post-processing step. A new dictionary is computed d' (middle) and the factorization is translated to match factors in the new dictionary (bottom).

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
d[i]	a	a	b	c	a	n	a	d
bv[i]	0	1	1	1	0	1	1	0
$\mathbf{rank_1}$[i]	0	0	1	2	3	3	4	5

... n a a b c b c b c ...

(5, 2) (1, 3) (2, 2) (2, 2) $\rightarrow$ (na) (abc) (bc) (bc)

	0	**1**	**2**	**3**	**4**
d'[i]	a	b	c	n	a

(3, 2) (0, 3) (1, 2) (1, 2) $\rightarrow$ (na) (abc) (bc) (bc)

A drawback of removing byte level redundancy is that the dictionary can no longer be used in an iterative halving run as there is no clear way to identify sample boundaries.

4.3 Experiments

The same experimental environment was used as described in the previous chapter. Pre- and post-processing schemes were implemented and evaluated on both GOV2 and Wikipedia document collections. All compression results reported used the ZZ encoding scheme described in Section 3.1.5 and 1 KB uniform sampling.

Pre-processed dictionary results are shown in Tables 4.3 and 4.4. Initial sampled dictionaries sizes were 0.5 GB, 1.0 GB, 2.0 GB and 4.0 GB. Each dictionary was passed through our pre-processing algorithm using varied minimum match block lengths. Given a raw dictionary and a block length each table reports the reduced pre-processed dictionary size, its compression effectiveness and the percentage of bytes which remained unused once it was used in an encoding.

Note that the original dictionaries contain a very high percentage of unused bytes at 30% and 20% on average for GOV2 and Wikipedia respectively. Setting a small minimal match length of 64 bytes reduced each dictionary by approximately a factor of three and only contributed to a 2% reduction in compression on average. Larger minimal match

Algorithm 6 PostProcess-BE accepts a RLZ encoding e and its dictionary d. The function returns a new dictionary where all unused bytes have been removed and a new RLZ encoding where each position value has been updated to valid positions in the new dictionary.

```
1: function PostProcess-BE($e$, $d$)
2:     initialize new encoding $e'$ to null
3:     initialize new dictionary $d'$ to null
4:     initialize byte usage bitvector $BV[0..|d|]$ to zero
5:     for each $(pos, len)$ factor in e do
6:         if $len = 0$ then
7:             continue
8:         for $i \leftarrow pos$ to $pos + len - 1$ do
9:             $BV[i] \leftarrow 1$
10:    for $i \leftarrow 0$ to $|d|$ do
11:        if $BV[i] = 0$ then
12:            continue
13:        $d' \leftarrow d' + d[i]$
14:    $rank_1 \leftarrow \mathsf{ComputeRank}(BV)$
15:    for each $(pos, len)$ factor in e do
16:        if $len > 0$ then
17:            $pos \leftarrow \mathsf{rank_1}(pos)$
18:        $e' \leftarrow e' + (pos, len)$
19:    return $(e', d')$
```

length values used on 2.0 GB and 4.0 GB dictionaries reduced their size by close to 50% and had an even smaller effect on compression effectiveness, 0.5% on average. All pre-processed dictionaries contained significantly less redundant information than their initial counterparts, with a smaller minimum match length achieving the lowest percentages of unused content. However, this comes at the price of worse compression, as the dictionary is smaller. All compression results reported by pre-processed dictionaries were significantly better than each baseline that was outlined in Tables 3.7 and 3.9 from the previous chapter. A minimal match length of 512 bytes reduced a 4.0 GB dictionary from GOV2 and Wikipedia by half and compressed the collection to 8.90% and 8.39% respectively.

Iterative halving results are reported in Tables 4.5 and 4.6. Each of the initial sampled dictionaries were halved in size until it reached 100 MB in size. As can be seen, halving dictionary size led to a small increase in encoding size, less than 1% per iteration. Even at a ten-fold reduction in dictionary size, compression was still better than all reported practical baselines outlined in the previous chapter.

Across both collections halving each dictionary in size using PostProcess-IH consistently reported better compression than its equivalent unprocessed dictionary. For example, a 4.0 GB sampled dictionary reduced to 2.0 GB compressed GOV2 to 8.67% where as a raw sampled 2.0 GB dictionary compressed GOV2 to 9.26%.

Just like the pre-processing results, compression is always most effective starting with a larger sampling of the collection, then tuning the dictionary size by eliminating redundancy. Observe that all 100 MB dictionaries computed by iterative halving, which is 0.02% and 0.04% the size of GOV2 and Wikipedia collections respectively, compress both collections to 13% on average — a result significantly better than all reported baselines and with such a restricted dictionary size. This compression result is also similar to the UZ pair encoding results reported in the previous chapter, which sacrifices compression effectiveness to dramatically improve decoding and random access time. As we are selecting the most frequently used samples during each iteration, unused dictionary percentages rapidly drop, essentially halving during each step with GOV2 dictionaries and even faster on Wikipedia, dropping to an average of 6% by the first iteration and slowly improving during subsequent iterations.

4.4 Summary

In this chapter, we explored techniques to remove redundant components of a sampled dictionary used for corpus compression. First, we described a pre-processing technique where long repetitive substrings are removed from a dictionary before compression is performed. We demonstrated that this method works well in practice, successfully removing large quantities of redundant substrings from a dictionary with no discernible effect on compression effectiveness. Next, we examined dictionary usage after compression. By computing usage statistics during decompression at a sample and character level we can make a more informed decision about which parts of the dictionary can be removed. First we described a sample-based technique where we make multiple passes of a collection, each time reducing dictionary size by half and keeping the most frequently used samples. We find that by the first iteration, while halving the overall size dictionary, has a minimal effect on compression, 0.2% on average. Furthermore, we showed that we can iteratively reduce the dictionary down to 100 MB of its original size and still maintain superior compression compared to each of the blocked baselines reported in the previous chapter. Finally, we described an alternative post-processing technique where unused characters are removed from a dictionary. We outlined a method to translate an existing encoding position values to point to its new position in the dictionary (where unused characters have been removed) without the need to re-encode the collection.

Results in this section lead to some interesting questions regarding the best method to generate a dictionary for corpus compression. We have shown that selective sampling, although a simple technique, provides efficient compression. Moreover, a finely tuned 100 MB dictionary achieves superior compression than all practical baselines. A small dictionary could be suitable for light-weight devices or provide the opportunity to add more effective samples to the dictionary — as we have seen, larger sampled dictionaries compress our two text collections to under 10% of their original size. One possibility would be to identify areas in a collection that compressed poorly during an encoding, then, once we have removed all redundant samples from the dictionary we can and add new samples from the poorly compressed areas, repeating the process for a number of iterations. An ideal solution however would generate an efficient dictionary before compression. We leave this as a problem for future work.

Table 4.3: Compression results and percentage of unused dictionary bytes for pre-processed dictionaries with varied dictionary size and minimal match lengths on the 426 GB GOV2 corpus.

Orig. (GB)	New. (GB)	Block (B)	Enc. (%)	Unused (%)
0.5	-	-	10.74	32.91
0.5	0.3	512	10.86	12.39
0.5	0.2	256	11.06	5.72
0.5	0.2	128	11.41	2.31
0.5	0.1	64	12.40	1.40
1.0	-	-	9.98	36.00
1.0	0.6	512	10.13	12.38
1.0	0.5	256	10.36	5.17
1.0	0.4	128	10.76	2.42
1.0	0.3	64	11.89	1.61
2.0	-	-	9.26	39.62
2.0	1.2	512	9.43	7.32
2.0	1.0	256	9.68	5.77
2.0	0.8	128	10.12	2.61
2.0	0.6	64	11.39	1.87
4.0	-	-	8.53	37.43
4.0	2.0	512	8.98	13.82
4.0	1.9	256	9.12	5.96
4.0	1.6	128	9.48	2.84
4.0	1.2	64	10.85	2.16

Table 4.4: Compression results and percentage of unused dictionary bytes for pre-processed dictionaries with varied dictionary size and minimal match lengths on the 256 GB Wikipedia corpus.

Orig. (GB)	New. (GB)	Block (B)	Enc. (%)	Unused (%)
0.5	-	-	11.77	21.15
0.5	0.4	512	11.86	5.25
0.5	0.3	256	12.01	3.55
0.5	0.2	128	13.38	2.90
0.5	0.1	64	13.26	2.79
1.0	-	-	9.89	23.72
1.0	0.8	512	10.80	6.52
1.0	0.6	256	10.99	4.38
1.0	0.5	128	11.31	3.69
1.0	0.3	64	12.33	3.42
2.0	-	-	9.06	27.34
2.0	1.5	512	9.56	6.58
2.0	1.2	256	9.95	5.31
2.0	1.0	128	10.34	4.64
2.0	0.6	64	11.52	4.21
4.0	-	-	7.71	19.95
4.0	2.0	512	8.39	7.39
4.0	2.0	256	8.64	5.28
4.0	1.8	128	8.62	5.21
4.0	1.2	64	10.22	4.30

Table 4.5: Compression results and percentage of unused dictionary bytes for post-processed dictionaries using iterative halving on the 426 GB GOV2 corpus.

Orig. (GB)	New. (GB)	Enc. (%)	Unused (%)
0.5	-	10.74	32.91
0.5	0.2	11.13	2.51
0.5	0.1	12.45	0.49
1.0	-	9.98	36.00
1.0	0.5	10.23	2.91
1.0	0.2	11.74	0.56
1.0	0.1	12.37	0.49
2.0	-	9.26	39.62
2.0	1.0	9.39	3.69
2.0	0.5	11.03	0.60
2.0	0.2	11.65	0.59
2.0	0.1	12.44	0.38
4.0	-	8.53	37.43
4.0	2.0	8.67	4.56
4.0	1.0	10.32	0.72
4.0	0.5	10.93	0.67
4.0	0.2	11.74	0.51
4.0	0.1	12.32	0.42

Table 4.6: Compression results and percentage of unused dictionary bytes for post-processed dictionaries using iterative halving on the 256 GB Wikipedia corpus.

Orig. (GB)	New. (GB)	Enc. (%)	Unused (%)
0.5	-	11.77	21.15
0.5	0.2	12.41	2.38
0.5	0.1	13.06	1.20
1.0	-	10.68	23.72
1.0	0.5	11.03	3.25
1.0	0.2	12.17	1.51
1.0	0.1	13.09	1.31
2.0	-	9.56	27.34
2.0	1.0	9.93	4.65
2.0	0.5	10.96	2.12
2.0	0.2	12.03	1.76
2.0	0.1	13.15	0.94
4.0	-	7.71	19.95
4.0	2.0	8.31	4.12
4.0	1.0	9.37	1.67
4.0	0.5	10.55	1.23
4.0	0.2	11.70	0.73
4.0	0.1	12.59	0.71

CHAPTER 5

Efficient Implementation of the Block Graph Data Structure

In Chapter 3 we presented a technique for compressing large, highly repetitive text collections that performs well in practice. Its effectiveness hinges on the dictionary representing the global repetitive properties of the collection. Because a sampling of the collection is used to generate the dictionary, if repetition throughout the collection is not uniform, or the sampling just happens to provide a poor representation of the collection, compression will suffer. As a consequence, theoretical analysis on the algorithm is difficult. In this chapter we approach the same problem – text compression and fast random access – by constructing an index that is competitive in both theory and practice.

Indexing highly repetitive texts to achieve fast random access has been studied extensively in recent years, see Grossi [2013] for a survey of the field. There are many approaches to the problem, such as LZ78 [Sadakane and Grossi 2006, Arroyuelo et al. 2012], the BWT [Ferragina and Venturini 2007] and grammar-based compression [Rytter 2003, Charikar et al. 2005, Bille et al. 2011, Maruyama et al. 2012; 2013]. Grammar-based compressors, such as Rytter [2003] and Charikar et al. [2005] give strong theoretical guarantees, and yet, there exists no feasible method to implement the algorithms in practice. A practical implementation of OLCA by Maruyama et al. [2012], a grammar-based algorithm exists, however, it does not support random access. Recently, Maruyama et al. [2013] proposed FOLCA, another grammar-based algorithm, and found that substring extraction was almost twice as slow as their baselines. In light of this, it was noted by Sirén et al. [2008] that algorithms based on LZ77 [Ziv and Lempel 1977] are better suited for compression of highly repetitive texts. Recently Kreft and Navarro [2010] introduced a variant of LZ77 called LZ-End and a supporting data structure that works well in practice but lacks good worst-case bounds for both compression and random access.

In this chapter we outline a practical implementation of the block graph by Gagie et al. [2011], an LZ-style data structure that supports fast random access in practice, but also

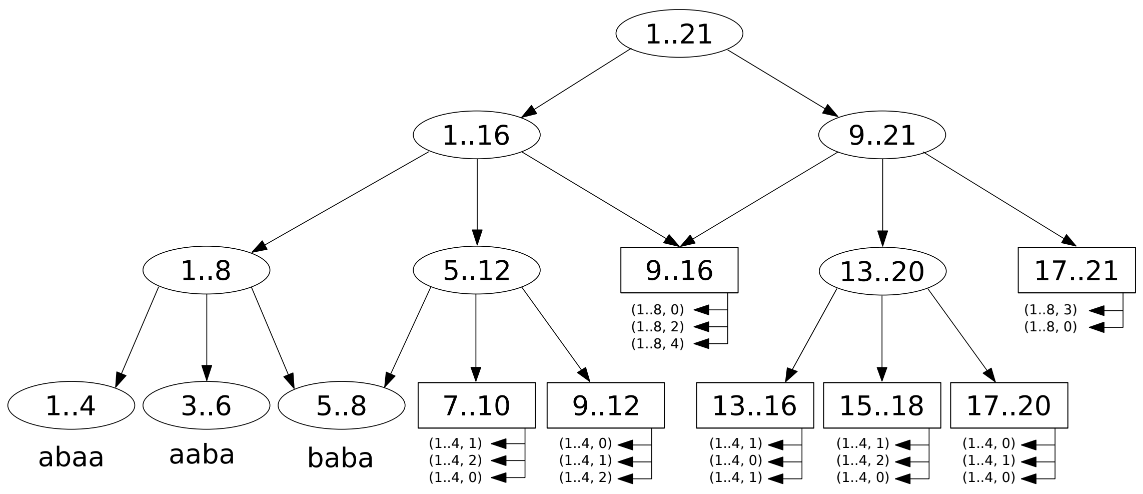


Figure 5.1: The block graph for the eighth Fibonacci string, abaababaabaababaababa, truncated at depth 3. Internal nodes are represented as ovals. Leaf nodes are represented as rectangles and their child pointers are a pair (n, o), where n is the internal node at the same depth of its parent where the child's block first occurred at offset o.

has strong theoretical bounds for compression and random access. We compare the block graphs against the current state-of-the-art methods, LZ-End by Kreft and Navarro [2010], OLCA, a variation of RLZ from Chapter 3 and adaptive general-purpose compressors.

The rest of the chapter is organized as follows. In Section 5.1 we give an overview of the block graph data structure. This is followed a high level discussion covering traversal of the block graph for substring extraction. In Section 5.2 we outline a practical implementation of a block graph and describe in detail how to navigate and represent its distinct components compactly. We empirically evaluate our implementation in Section 5.3 against a number of state-of-the-art indexes that provide fast random access. In Section 5.4 we discuss our results and conclude in Section 5.5.

5.1 Block Graph

A block graph is a directed acyclic graph (DAG) built on a string $x[1..n]$. The general structure of a block graph is best described visually. In Figure 5.1 we show an illustration of a block graph for the eighth Fibonacci string, abaababaabaababaababa. Each node, $(i..j)$, maps to a substring, $x[i..j]$, which we call the nodes block. The root, or source node, corresponds to the complete text, $x[1..n]$. A node, v, can have up to three children, representing the first, middle and last half of its substring. Note that a node's middle child corresponds to the overlap between its first and last siblings. This implies that a node can have two parents. For example, the node (5..8) in Figure 5.1 acts as the right child of (1..8) and the left child of (5..12). This overlap plays an important role during construction when assigning leaf node pointers, which will be discussed in detail later.

Consider a text of length $n = 2^h$ for some value h. For simplicity we use a length that is a power of two, however, in practice this is not a requirement. If n is not a power of 2 we append blank characters until it is. Once the block graph has been constructed we remove all redundant nodes, that is, nodes comprised entirely of blanks, and adjust any remaining nodes with blocks, $(i..j)$ where $j > n$ to $(i..n)$. For example, the block graph in Figure 5.1 represents a string of length 21. The text would have initially been padded to $x[1..32]$ and redundant nodes trimmed during construction. Furthermore, its block (9..21) would have originally represented the substring (9..24) and was truncated to (9..21).

We consider the root node to be at depth 0 and the block graph to have a maximum possible depth of $t = 2^{\lceil \log n \rceil}$. We can reduce the size of the block graph by truncating it at a depth where storing three pointers takes less space than storing a block of characters explicitly. A node at depth d will have a block size, $b = 2^{t-d}$, corresponding to the substring $x[i..i + b]$. For each node we add pointers to three children, $x[i..i + b/2)$, $x[i + b/4..i + 3b/4)$ and, $x[i + b/2..i..b)$, creating them if necessary, as a node may already exist due to child pointer overlap. If a node's block is the first occurrence of its substring in x we mark it as an *internal node*. If a nodes is not unique, that is, its block has previously occurred in x we identify it as a *leaf node*. In Figure 5.1, internal nodes are represented as ovals and leaf nodes are rectangles.

The main operation during block graph construction is assigning leaf nodes and updating their pointers. Say we have established that a leaf node is a block that has already occurred in x. We update its children to point to an offset in an internal node at the same level as its parent. This is one reason for the overlapping blocks, as the child at a leaf node will be fully contained in an internal node at its parents depth. We represent a leaf node pointer as a pair (n, o), where n corresponds to the node that contains the first occurrence of a child's block at offset o. For example, the block (17..21) in Figure 5.1, corresponds to $x[17..21] =$ ababa, which first occurs at $x[4..8]$. We turn it into a leaf node by updating its child pointers (17..20) and (19..21), corresponding to the blocks $x[17..20] = abab$ and $x[19..21] = aba$, which first occur in positions 4 and 1 in x respectively. Therefore, we replace (17..21)'s pointer to (17..20) by a pointer to (1..8) and the offset 3, and replace its pointer to (19..21) by another pointer to (1..8) and the offset 0.

5.1.1 Extracting a Single Character

Extracting a single character from a block graph is straightforward. We begin at the root node and descend through the graph via nodes that contain $x[i]$. If there are two child nodes that contain the specific index we are looking for we make an arbitrary choice, in our implementation we always select the left-most path. If we descend to a leaf node, u such that $x[i]$ is the j^{th} character in u's block we follow one of its pointers to an internal node and adjust the extract index by the pointers offset. That is, if u stores a pointer to internal node v and offset c, we follow u's pointer to v and extract the $(j+c)^{th}$ character in v's block. Finally, when we reach the depth where internal nodes store raw text in-place of node pointers we return $x[i]$.

For example, if we wanted to extract the 17^{th} character from the block graph shown in Figure 5.1 we could take the following path. Beginning at the root node we descend to (9..21) as it is the only child from the root that contains $x[17]$. From (9..21) we can follow its middle or right child as they both contain $x[17]$. If we descend to its middle child, (13..20), once more, we are presented with the choice to follow either (13..20)'s middle or right child. Suppose we select its middle child again, the leaf node (15..18). All three children of (15..18) point to an offset in an internal node at same depth of its parent. Its left child (15..16) corresponding to the substring $x[15..16]$ = ba has been replaced by a pointer to (1..4) at offset 1, signifying that the first occurrence of this substring is located at $x[2..3]$. Similarly, its middle child (16..17), $x[16..17]$ = aa has been replaced by a pointer to (1..4) at offset 2, indicating that its first occurrence it located at $x[3..4]$. Finally, the right child, (17..18), $x[17..18]$ = ab has been replaced by a pointer to (1..4) at offset 0, corresponding to $x[1..2]$. Suppose we follow the middle pointer, (16..17). Since we were going to extract the second character from (16..17) and the middle pointer takes us to (1..4) at offset 2, we end up extracting the fourth character in (1..4). This node is an internal node at the block graph's truncated depth, so it stores the substring $x[1..4]$ = abaa. Finally, we extract the fourth character from 1..4, $x[4]$ = a, which is equivalent to $x[17]$.

5.1.2 Extracting a Substring

Extracting a substring from x is slightly more complicated. The procedure is similar to extracting a single character, however, as we descend through the graph, once the substring interval length, $j-i+1$ is longer than half of the current block length we need to split the query in two.

There are a number of scenarios to consider when extracting a substring from a block graph. If the current node u is at the lowest depth of the graph, that is, where internal nodes store text instead of pointers, we simply return the required substring $x[i..j]$. Otherwise, we are at an internal or a leaf node and need to determine if the requested interval $x[i..j]$ is fully contained in one of its children. If so we descend to the appropriate child and continue, otherwise, we need to split the interval into two or three sub-intervals. As an example, consider the function extract(u, i, j), where u is a node in a block graph and (i, j) represent the range of characters to extract in u's block. Say the current state of an extract call traversing the block graph from Figure 5.1 is extract$(1..8, 1, 8)$. That is, we are at node (1..8) and want to extract the substring $x[1..8]$. As the interval is not fully contained in one of u's children we have to split the query. This could be achieved with two extract calls, for example, extract$(1..4, 1, 4)$ and extract$(5..8, 5, 8)$, or three, for example, extract$(1..4, 1, 3)$, extract$(3..6, 4, 5)$ and extract$(5..8, 6, 8)$. Although it is possible to split into three sub-intervals it is never ideal, and, due to the overlap in child nodes the same outcome can always be achieved by partitioning into two sub-intervals. In doing so we avoid the cost of an extra function call.

For a more complete example, say we want to extract the substring, $x[6..10]$ of length 5 from the block graph in Figure 5.1. We begin at the root node and call, extract$(1..21, 6, 10)$. We move to its left child by calling, extract$(1..16, 6, 10)$, then its middle child, extract$(5..12, 6, 10)$. Here the node $(5..12)$ is of length 8. As our query is of length 5, which is greater than half the length of the current block, we need to split the query into two sub-intervals. In this case we call, extract$(5..8, 6, 8)$, which corresponds to a text block, so we extract the text aba. For the second sub-interval we call, extract$(7..10, 9, 10)$. $(7..10)$ is a leaf node. We want to access is right child, corresponding to the block $(9..10)$. The child pointer directs us to the internal node $(1..4)$ at offset 0. We move to $(1..4)$, adjusting the interval to account for the offset, which is 0 in this case, by calling extract$(1..4, 1, 2)$. As $(1..4)$ is a text node we extract the substring, ab. Concatenating the result of the two sub-intervals gives abaab, which corresponds to the substring, $x[6..10]$.

5.1.3 Time and Space Complexity

Gagie et al. [2011] show that the block graph contains $\mathcal{O}(z \log n)$ nodes, where z is the number of phrases in the LZ77 parse of a text, therefore requires $\mathcal{O}(z \log n)$ words of space.

The maximum depth of a block graph is $\log n$. For each node we either descend to one of its children or follow a leaf pointer to an internal node in constant time. Therefore, we can extract a single character $x[i]$ in $\mathcal{O}(\log n)$ time.

Substring extraction proceeds in exactly the same manner as above, that is, descending through the block graph to each node in constant time, until m, the length of the substring to extract, is more than half the size of the current block. In Section 5.1.2 we show that a node at depth d, with a block length of $b = 2^{\lceil \log n \rceil - d}$, and a substring query of length $b/2 < m \leq b$, can always partitioned into two auxiliary extract calls, for which each call continues descending through the graph in constant time. Summing across each level we can extract a substring in $\mathcal{O}(\log n + m)$ time.

Gagie et al. [2011] note that we can remove the top d levels of the block graph and reduce the space and access time. For example if $d = \log z$, then we store a total of $\mathcal{O}(z \log n \log(n/z))$ bits and need only $\mathcal{O}(\log(n/z))$ time for access. However, as we discuss in Section 5.4, this would give a negligible improvement in practice, as the top d levels represent a very small component of the block graph.

5.2 Implementation

In this section we describe an implementation of a block graph which is efficient in practice. The main idea is to represent the shape of the graph (the internal nodes and their pointers) using bitvectors and operations from succinct data structures, and to carefully allocate space for the leaf nodes depending on their distance from the root. Below we make use of two familiar operations for bitvectors: rank and select. Given a bitvector B, a position i, and a type of bit b (either 0 or 1), $\mathsf{rank}_\mathsf{b}(B, i)$ returns the number of occurrences of b before position i in B and $\mathsf{select}_\mathsf{b}(B, i)$ returns the position of the i^{th} b in B. Efficient data structures supporting these operations have been extensively studied, [Okanohara and Sadakane 2007a, Raman et al. 2007, Kärkkäinen et al. 2014a], we give an overview of this topic in Section 2.2.3.

Recall that each level of the block graph consists of a number of nodes, either internal nodes, or leaves. Let B_d be a bitvector which indicates whether the i^{th} node (from the left) at depth d is a leaf, $B_d[i] = 0$, or an internal node, $B_d[i] = 1$. We define another bitvector R_d, where $R_d[i] = 1$ if and only if $B_d[i] = 1$ and $B_d[i + 1] = 1$ for $i < n - 1$. That is, we mark a 1 bit for each instance of two adjacent internal nodes in B_d, otherwise $R_d[i] = 0$. Let L_d be an array that holds leaf nodes at depth d. The structure of a leaf node is discussed below. Finally, let T be the concatenation of the textual representations, that is, the corresponding substrings of all internal nodes at the truncated depth, d'. As adjacent text blocks share $2^{\log n - d' - 1}$ characters, we concatenate only the last half of a new adjacent block to T. Non-adjacent blocks are fully concatenated. We utilize bitvector R_d at this level so that we can extract the correct substrings; however, we mark $R_d[i] = 1$ if the i^{th} node at the truncated depth is a text block.

Table 5.1 gives the bitvectors B and R, and text block T for the block graph in Figure 5.1. Note that there is no need to store B_0 and R_0 as there is only ever one root node. Furthermore, at the truncated depth there are three adjacent internal nodes of length 4. Instead of storing the concatenation of the three blocks abaa, aaba and baba, we only store the last half of each adjacent block, that is abaa, ba and ba, resulting in the string $\mathsf{abaababa}$.

5.2.1 Navigating the Block Graph

The main operation is to traverse from an internal node to one of its three children. Say we are currently at the j^{th} internal node at depth d of the block graph, that is, we are at $B_d[i]$, where $i = \mathsf{select}_1(B_d, j)$. Each internal node has three children. If these children were independent then locating the left child of the current node would simply be three times the node's position on its level, that is $3j = 3 \cdot \mathsf{rank}_1(B_d, i)$. However, in a block graph, adjacent internal nodes share exactly one child, so we correct for this by subtracting the number of adjacent internal nodes at this depth prior to the current node — this is given by $\mathsf{rank}_1(R_d, i)$. To find the position corresponding to the left child of a node in B_{d+1} we compute $\mathsf{leftchild}(B_d, i) = 3 \cdot \mathsf{rank}_1(B_d, i) - \mathsf{rank}_1(R_d, i)$. Note that we are using zero indexing for B, R and T arrays, as it simplifies calculations.

Given the address of the left child, it is easy to find the center or right child by adding

Table 5.1: A succinct representation of the block graph from Figure 5.1. One bits in B_d bitvectors represent internal nodes at depth d. One bits in R_d bitvectors represent runs of adjacent internal nodes at depth d. The array T contains the text at the truncated depth.

i	01234567
B_1	11
R_1	10
B_2	11010
R_2	10000
B_3	11100000
R_3	11000000
T	abaababa

1 or 2, respectively. If $B_d[i] = 0$ then we are at a leaf node, and its leaf information is at $L_d[\mathsf{rank}_0(B_d, i)]$. Once we reach the truncated depth we access the text of an internal node by computing its offset in T. The length of a block at the truncated depth d' is $b' = 2^{\log n - d'}$. To compute a block's offset in T we first compute its index assuming that all text blocks were fully concatenated, $\mathsf{rank}_1(B_d, i) \cdot b'$, then we account for any overlapping adjacent text blocks of length $b'/2$. Therefore, $\mathsf{rank}_1(B_d, i) \cdot b' - \mathsf{rank}_1(R_d, i) \cdot b'/2$ corresponds to the index of the i^{th} text block in T.

For example, to extract the 17^{th} character using the bitvectors in Table 5.1 we will follow the same path outlined in Section 5.1.1, that is, (9..21), (13..20), (15..18), (16..17), then (1..4). Starting at depth 1 the bitvector B_1 states that there are two internal nodes, corresponding to (1..16) and (9..21) at index 0 and 1 respectively. We want to find the position in B_2 of (9..21)'s middle child, (13..20). The position of (9..21)'s left child is $3 \cdot \mathsf{rank}_1(B_1, 2) - \mathsf{rank}_1(R_1, 2) = 3 \cdot 1 - 1 = 2$. Then we add one to get the index of its middle child, at position 3. That is, $B_2[3]$ corresponds to the node (13..20). We compute $\mathsf{access}(B_2, 3) = 1$, so it is an internal node. We want to follow its middle child again to get to (15..18). We compute $3 \cdot \mathsf{rank}_1(B_2, 3) - \mathsf{rank}_1(R_2, 3) = 3 \cdot 2 - 1 = 5$, then add 1 to get the index of its middle child, 6. Continuing, $B_3[6]$ corresponds to the node (15..18). $\mathsf{access}(B_3, 6) = 0$, so it is a leaf node. We lookup the position and offset for (15..18)'s middle child, (16..17) in the L_3 calling $L_3[\mathsf{rank}_0(B_3, 6)]$, which returns (0,2) corresponding to the node (1..4) and offset 2. This time $\mathsf{access}(B_3, 0) = 1$ which means it is a text node as we are at the truncated depth of the block graph. We required the second character in (16..17), so we need the index of the second character plus the leaf offset in (1..4), which is index 3. We know that (1..4) is the first block at this depth, therefore, its text block will begin at $T[0]$, however, in the general case, to compute the index of a text block in T, we would use $\mathsf{rank}_1(B_3, 0) \cdot 4 - \mathsf{rank}_1(R_d, 0) \cdot 2 = 0 \cdot 2 - 0 \cdot 2 = 0$, where 4 is the length of a block at the truncated depth and 2 is the length of overlap between adjacent text blocks. Finally, we have an offset into T and an index in (1..4) of the character we are after, we return $T[3]$.

Substring extraction operates in a similar manner. Note from Section 5.1.2 that care must be taken to reduce the number of extract partitions when descending through the graph.

5.2.2 Representing Leaf Nodes

In a block graph, leaves point to internal nodes. For each leaf we store two values, the position of the destination node on the current level, and an offset in the destination node pointing to the beginning of the leaf block. Note that we do not need to store the depth of the destination node. It is, by definition, on the level above the leaf pointer, and we know this by keeping keep track of the depth at each step in a traversal. To improve compression we store leaf pointer and offsets in two separate arrays.

At depth d there are no more than $2^{d+1} - 1$ possible nodes, so we can store each pointer in $\lceil \log(2^{d+1} - 1) \rceil$ bits. However, if we record the number of nodes at each depth we can reduce number of bits required for the pointer array at the cost of an extra rank operation. For example, at depth 2 in Figure 5.1 there are 5 nodes — 3 internal nodes and 2 leaf nodes. A leaf node will only point to an internal node at the same depth so we can reduce the number of bits required for each leaf pointer at depth 2 from $\lceil \log(2^{2+1} - 1) \rceil$ = 3 bits, to $\lceil \log(3) \rceil = 2$ bits. To fetch the index of the i^{th} internal node at level d we call $\mathsf{rank}_0(B_d, i)$.

For the offsets array we observe that the length of a node at depth d is $b = 2^{\lceil \log n \rceil - d}$, and a child pointer node represents a block of length $b/2$. A leaf node can have an offset value in the range $0 \leq i \leq b/2$, so we can store each offset in $\lceil \log(b/2 + 1) \rceil$ bits.

In Section 5.4 we examine the three components of a block graph, its bitvectors, leaf blocks and text blocks. We find that the bitvectors represent a very small percentage of a block graph and the dominant cost is storing the leaf nodes. In our practical implementation we store leaf pointers in $\lceil \log(2^{d+1} - 1) \rceil$ bits, however, a more compact representation may be of interest when indexing larger collections.

5.2.3 Constructing the Block Graph

Construction of a block graph is straightforward. We build the graph in a top-down manner, that is, we begin at the root node and update node pointers a level at a time, truncating the graph at a specified block length. For each node at the current level we initially check if it is a leaf node, if so, we assign it leaf pointers and offsets and continue. Otherwise, the node is an internal node and we link them to their child internal nodes, creating them if necessary. We then continue the procedure, operating on the nodes we created at the next depth.

The computational bottleneck during construction is determining if a node is a leaf node, that is, if its block has already occurred in the input text. Once we have detected a leaf node we can use the same operation with a smaller block to assign leaf node pointers. To perform this efficiently we compute the suffix array of the input text and use its pattern matching capability we described Section 2.2.2.

Our implementation uses a suffix array. If the input text is too large to store an index in memory, we use a disk-based variant of the doubling algorithm [Arge et al. 1997, Crauser and Ferragina 2002] described by Dementiev et al. [2008], to compute the leaf blocks at each level before we build the block graph.

5.3 Experiments

We implemented the block graph as described in Section 5.2 with a reference implementation available online[1] and evaluated its compression effectiveness on texts from the Pizza-Chili Repetitive Corpus,[2] a standard test bed for data structures designed for repetitive strings. All three categories of texts were used, ranging from highly compressible *artificial* texts, such as the 41^{st} Fibonacci string, to *pseudo-real* texts that were generated by artificially adding repetitiveness to existing collections, and *real* texts such as collections of Wikipedia articles, source code and DNA.

We evaluate our implementation against the LZ-End data structure by Kreft and Navarro [2010], OLCA by Maruyama et al. [2012], and two general-purpose compressors gzip and xz. We were unable to test against FOLCA, a more recent data structure by Maruyama et al. [2013] as their source code is not available. However, their experiments show that, apart from requiring fewer resources during construction, that decompression and random access is twice as slow on average compared to LZ-End. Finally, we compare against an implementation of RLZ from Chapter 3 that was altered to support data without explicit document headers.

LZ-End and OLCA were run with default arguments. gzip and xz were run with their highest compression setting -9. RLZ used a dictionary generated from 1 KB uniform sampling. Each dictionary was generated to be 2% of each test collection in size. Throughout our experiments we tested block graphs with varied truncated depths such that the smallest blocks were 4, 8, 16, 32 and 64 bytes. Note that OLCA, gzip and xz provide compression only, not random access, and are included as reference points for achievable compression. We did not include self-indexes in our experiments such as the LZ-Index by Navarro [2004] or RLCSA by Sirén et al. [2008] as experiments by Ferragina

[1]http://www.github.com/choobin/block-graph
[2]http://pizzachili.dcc.uchile.cl/repcorpus.html

et al. [2009] and Kreft and Navarro [2010] both show that they achieve slightly worse compression and slower extraction speeds compared to LZ-End.

Random access was evaluated on Block graphs, LZ-End and RLZ extracting substrings of varied lengths from a single character to 2^{18} characters in length. Each run of extractions was performed across 10,000 randomly-generated queries. Experiments were conducted on an Intel Core i7-2600 3.4 GHz processor with 8 GB of main memory, running Linux 3.3.4; code was compiled with GCC version 4.7.0 targeting x86_64 with full optimizations. Caches were dropped between runs with `sync && echo 1 > /proc/sys/vm/drop_caches`. Time values were reported using wall clock time.

5.4 Discussion

Compression results are presented in Tables 5.2 and 5.3. We report compressed space in MB and time in seconds respectively. Regarding the two general-purpose compressors, gzip provided the worst compression overall, specifically on the artificial and pseudo-real collections. gzip on real collections was somewhat competitive, but still worst overall with the exception of two collections, Escherichia_coli and influenza, which it actually compressed better than OLCA, LZ-End and Block graphs. As expected, gzip had the fastest compression time across all texts due to the nature of its algorithm. xz provides superior compression on all pseudo-real and real texts, with significantly better results than all other algorithms. xz was also competitive in run-time compared to OLCA and Block graphs and faster than LZ-End. It is important to note that while xz provides superior compression and gzip gives the fastest decoding speeds that they both do not support random access.

RLZ and OLCA compression results were very similar, preforming slightly worse than LZ-End and Block graph's and better than gzip in terms of size. Furthermore, compression time was four times faster than LZ-End and twice as fast as Block graphs on pseudo-real and real texts. We observed no significant degradation in compression effectiveness for Block graph up to truncated block size of 32. This is expected as three pointers on a 64 bit machine is 24 bytes, and block lengths are always powers of two, so best compression should be achieved between block a length of 16 and 32. Compression degraded using a truncated block length of 64 by 5% on average, however, it provides random access speeds an order of magnitude faster than other Block graphs and LZ-End. LZ-End's compression time performance was the worst overall, being four times slower compressing artificial collections and twice as slow on real texts compared to Block graphs.

Figure 5.2 gives an overview of the relative sizes of each Block graph component, its bitvectors, leaf block and text blocks on real texts using truncated block lengths of 4, 8, 16 and 32. The first key observation is that for all truncated block lengths the size of the bitvectors is insignificant compared to the leaf and text blocks. That is,

Table 5.2: Size in MB compressing artificial (top), pseudo-real (middle) and real (bottom) text collections from the Repetitive Corpus with baseline general-purpose compressors, RLZ, OLCA, LZ-End, and Block graphs. Block graphs were truncated at text length 4, 8, 16, 32 and 64. RLZ used a dictionary 2% of each collection size and 1 KB uniform sampling.

Collection	**ASCII**	**GZIP**	**XZ**	**RLZ**	**OLCA**	**LZ-End**	**Bg4**	**Bg8**	**Bg16**	**Bg32**	**Bg64**
fib41	267	1.17	0.19	0.94	0.00237	0.00154	0.00225	0.00219	0.00248	0.00212	0.00214
rs.13	216	1.09	0.13	0.77	0.00277	0.00238	0.00253	0.00248	0.00242	0.00244	0.00245
tm29	268	1.42	0.45	0.93	0.00316	0.00274	0.00247	0.00241	0.00236	0.00234	0.00239
dblp.xml.00001.1	100	18.35	0.15	0.37	1.05	1.23	1.66	1.38	1.31	1.39	1.50
dblp.xml.00001.2	100	18.51	0.15	0.44	1.07	1.28	1.65	1.38	1.31	1.39	1.50
dblp.xml.0001.1	100	18.39	0.18	0.45	2.20	1.70	4.09	3.68	3.60	3.89	4.65
dblp.xml.0001.2	100	19.85	0.18	1.42	2.24	2.16	4.07	3.66	3.59	3.87	4.62
dna.001.1	100	28.48	0.52	1.35	10.26	6.49	18.28	18.18	18.30	20.03	25.85
english.001.2	100	44.60	0.56	10.27	12.45	11.13	21.64	19.03	18.39	20.12	26.01
proteins.001.1	100	40.36	0.61	1.35	9.97	7.03	21.04	18.09	19.84	25.75	38.24
sources.001.2	100	36.02	0.45	10.00	10.32	10.62	21.02	18.88	18.31	20.05	25.97
Escherichia_Coli	112	31.53	5.18	28.89	58.45	49.10	49.70	49.57	45.33	46.91	53.23
cere	461	120.08	5.07	51.38	78.30	41.34	57.68	57.54	54.59	57.96	59.66
coreutils	205	49.92	3.70	37.64	66.70	35.88	42.80	33.19	30.43	33.00	38.78
einstein.de.txt	92	28.79	0.10	0.57	0.80	0.83	1.14	1.02	1.00	1.08	1.30
einstein.en.txt	467	163.66	0.33	2.55	2.02	2.24	3.52	3.07	3.01	3.19	3.76
influenza	154	10.63	1.59	8.07	19.49	21.50	33.16	32.97	33.32	37.89	50.86
kernel	257	69.39	2.07	30.13	39.95	19.34	21.21	15.69	13.84	14.05	15.20
para	429	116.07	6.09	60.18	82.97	57.41	72.39	72.13	67.84	70.66	80.99
world_leaders	49	8.28	0.51	3.51	3.88	4.52	6.62	5.83	5.72	6.37	7.97

Table 5.3: Run-time in seconds to compress artificial (top), pseudo-real (middle) and real (bottom) text collections from the Repetitive Corpus with baseline general-purpose compressors, RLZ, OLCA, LZ-End, and Block graphs. Block graphs were truncated at text length 4, 8, 16, 32 and 64. RLZ used a dictionary 2% of each collection size and 1 KB uniform sampling.

Collection	**GZIP**	**XZ**	**RLZ**	**OLCA**	**LZ-End**	**Bg4**	**Bg8**	**Bg16**	**Bg32**	**Bg64**
fib41	2	103	62	39	198	77	74	74	74	74
rs.13	1	85	66	32	312	62	60	60	60	60
tm29	2	110	92	98	319	90	86	86	86	86
dblp.xml.00001.1	3	36	13	29	108	40	37	36	35	34
dblp.xml.00001.2	3	36	14	30	121	31	29	27	26	25
dblp.xml.0001.1	3	38	13	29	128	54	50	48	46	43
dblp.xml.0001.2	4	36	15	33	147	52	48	46	42	39
dna.001.1	89	74	12	26	164	116	114	99	84	70
english.001.2	9	51	11	42	166	108	94	80	67	53
proteins.001.1	3	54	9	29	121	102	90	80	70	61
sources.001.2	14	46	8	33	121	102	91	79	66	53
Escherichia_Coli	96	117	33	36	181	197	197	121	80	53
cere	372	436	95	134	1016	605	578	490	437	387
coreutils	12	95	36	88	295	169	138	108	83	64
einstein.de.txt	5	21	11	27	173	25	23	22	22	21
einstein.en.txt	26	110	92	172	552	160	152	150	147	146
influenza	31	91	26	32	218	211	210	188	156	119
kernel	14	111	36	98	319	127	101	79	67	60
para	363	408	105	128	762	535	495	379	307	255
world_leaders	2	22	7	11	48	29	27	25	23	20

the efficient leaf pointer representation outlined in Section 5.2.2 would have an almost negligible improvement in space at the cost of an extra rank call for each leaf node during traversal. However, we do not believe this method would be of value when compressing larger collections such as GOV2 from Section 3. We leave this for future work.

When truncating the graph at low depth, for example, where the final block is length 4 and 8, the leaf blocks dominate the size of the block graph. At higher depths, such as where the final block is of length 16 and 32, the text blocks take up at least half the overall size of the block graph, and, most notably, have no significant impact on compression effectiveness. Furthermore, in some cases a higher truncated depth slightly improves compression, for example, coreutils and Escherichia_Coli. This raises some interesting questions, such as how compressible the concatenated text is and what can be done to reduce the text while still providing random access.

Extract results for all Repetitive Corpus texts are shown across Figures 5.3 to 5.6. Observe that for substring extraction with lengths less than 2^6 all algorithms performed at nearly the same speed, around 9 million characters per second. The mean extraction speed for LZ-End runs never exceeded 9 million characters per second. All block graph implementations achieved faster substring extraction than LZ-End. The larger the truncated block length, the faster the algorithm performed, as it minimizes the number of nodes to access during traversal. This can be seen on all graphs, especially for extraction of larger substrings where we observe an exponential increase in speed as we increase the truncated block length.

RLZ works well in practice across all text collections, repeating our findings from Chapters 3 and 4, that it is a practical and efficient method for compression that supports fast random access. Most importantly, our experiments show that block graphs generally achieve compression comparable to that achieved by LZ-End while supporting significantly faster substring extraction.

5.5 Summary

In this chapter we presented the first practical implementation of the block graph data structure by Gagie et al. [2011]. First, we gave a conceptual overview of the data structure and described its random access capabilities at a high level. Then, we outlined a practical implementation of a block graph using bitvectors and operations on succinct data structures. Finally we empirically evaluated our implementations against a number of general-purpose compressors and LZ-End, a current state-of-the-art compressed index that provides fast random access.

We found that although xz, a general purpose compressor, achieves much better compression, block graphs achieve better compression than gzip except on the Escherichia Coli and influenza files. Most importantly, our experiments show that block graphs generally achieve compression comparable to that achieved by LZ-End while supporting significantly faster substring extraction, demonstrating that block graphs are competitive in both theory and practice.

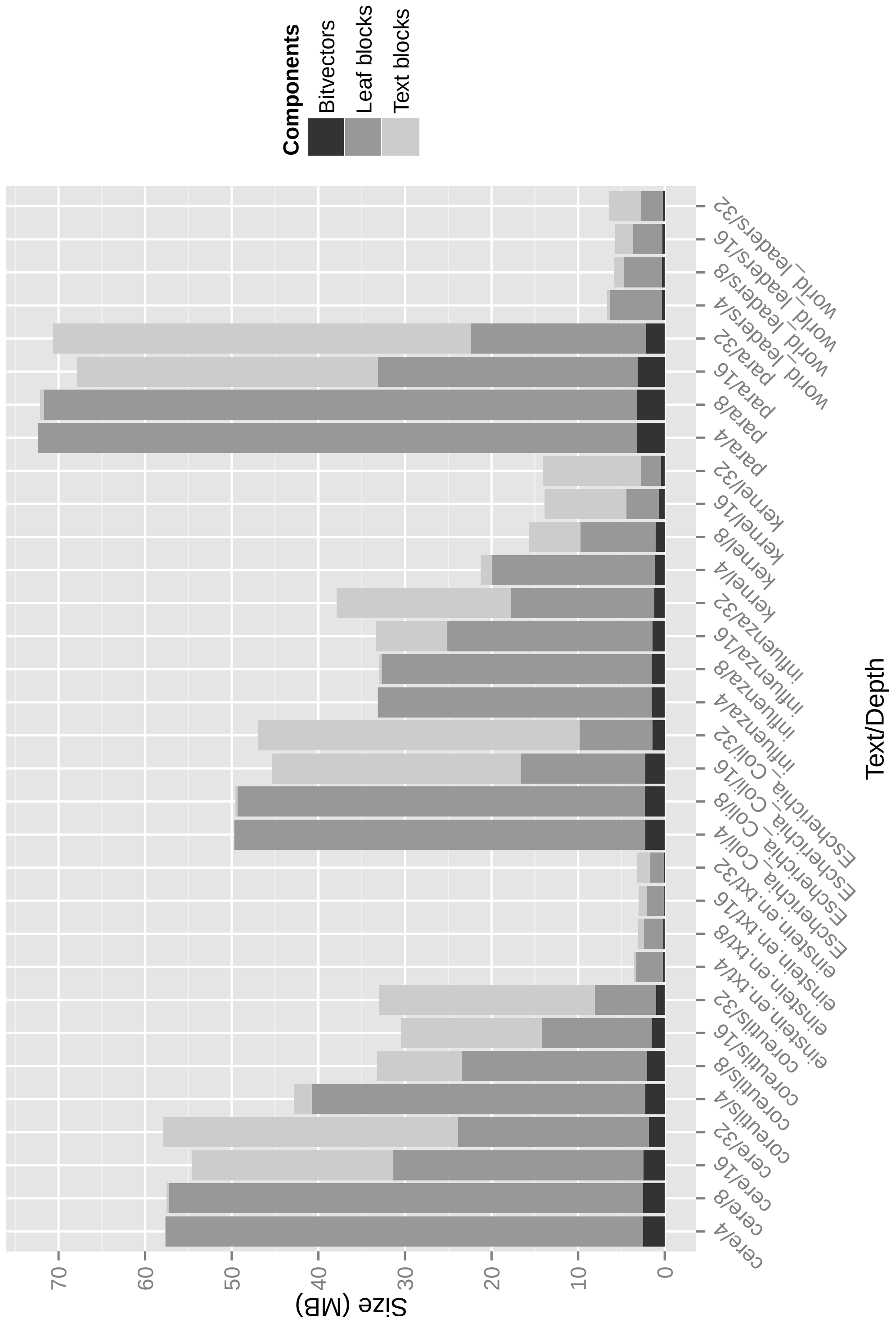


Figure 5.2: Visual representation of the relative space requirements of a block graphs three components, its bitvectors, leaf blocks and text blocks for a variety of input files and truncated depths.

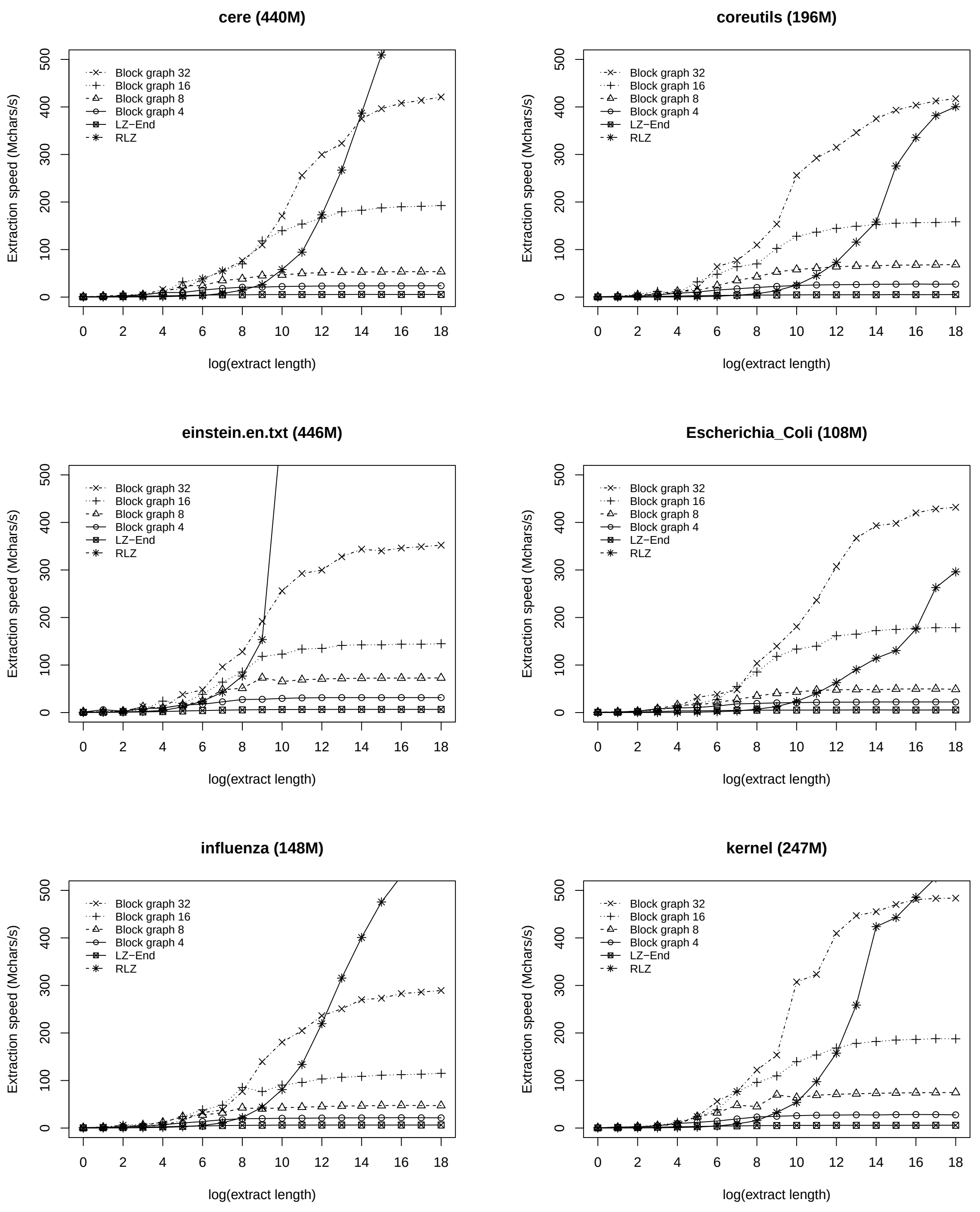


Figure 5.3: Extraction speeds in millions of characters per second versus the binary logarithm of the length of the extracted substring. Each data point is averaged over 10,000 random substring extractions.

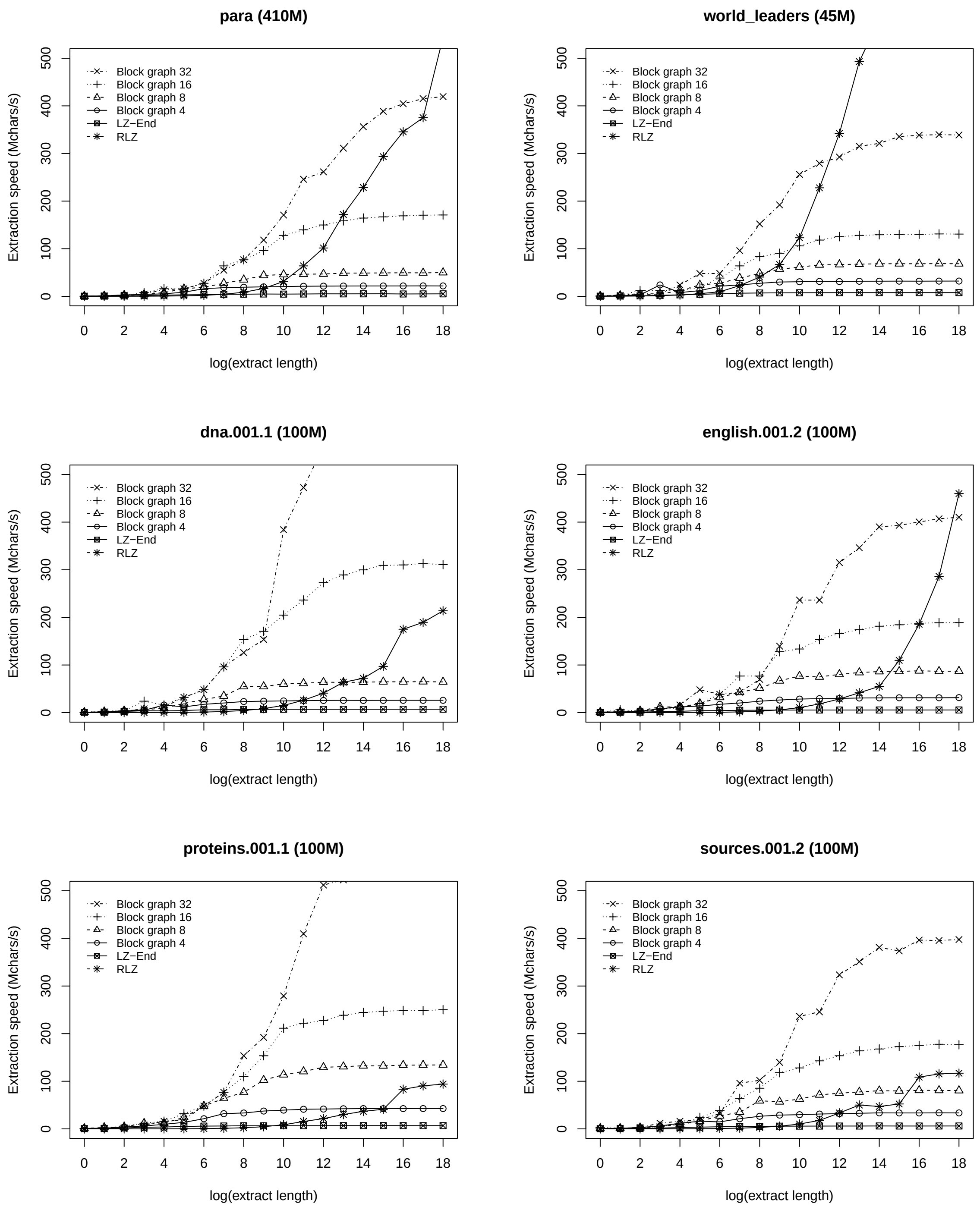


Figure 5.4: Extraction speeds in millions of characters per second versus the binary logarithm of the length of the extracted substring. Each data point is averaged over 10,000 random substring extractions.

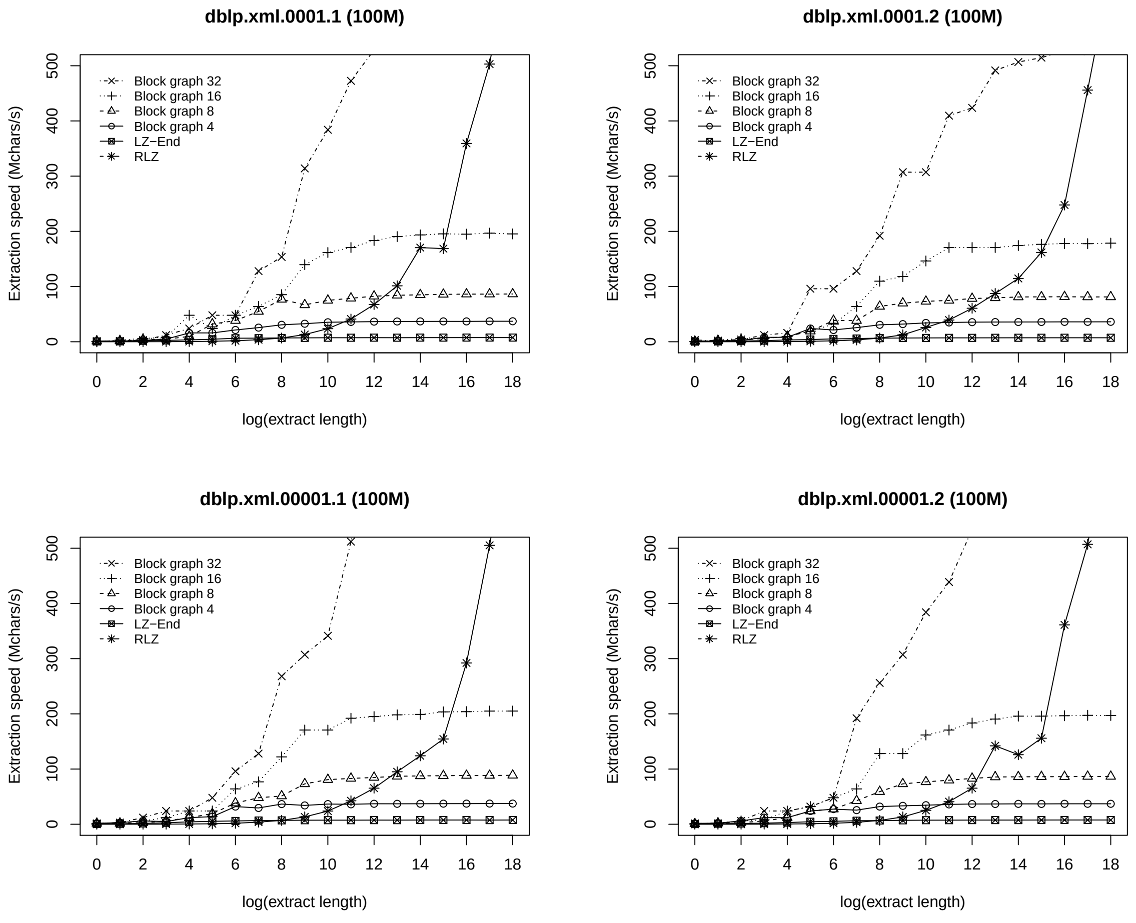
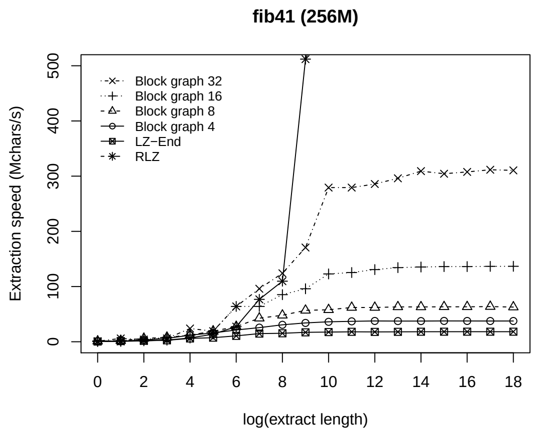
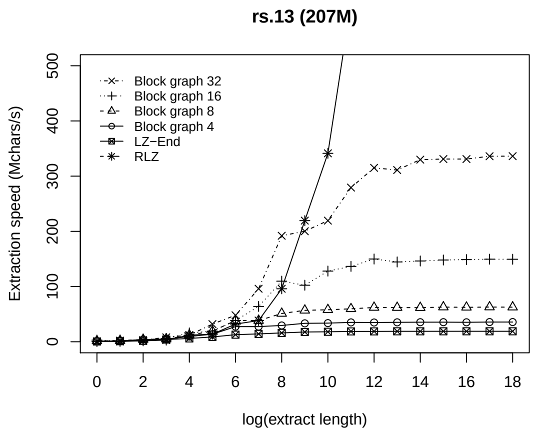


Figure 5.5: Extraction speeds in millions of characters per second versus the binary logarithm of the length of the extracted substring. Each data point is averaged over 10,000 random substring extractions.





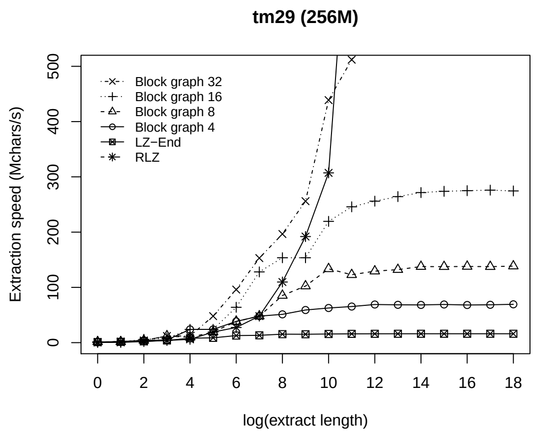


Figure 5.6: Extraction speeds in millions of characters per second versus the binary logarithm of the length of the extracted substring. Each data point is averaged over 10,000 random substring extractions.

CHAPTER 6

Fast and Efficient Compression of High-Throughput Sequencing Reads

Recent advances in high-throughput DNA sequencing technology has dramatically changed bioinfomatics and life sciences research, providing faster and more affordable sequencing for research groups all over the world [Kircher and Kelso 2010]. We are now witness to exponential growth in the generation of genomic data. Figure 6.1 plots the growth of the Sequence Read Archive, a biological database for DNA sequencing data, from its conception in 2009 to the present day. Note the y-axis is logarithmic. This rapid growth is completely unprecedented, and more importantly, is expected to double every ten months for the next decade [Cochrane et al. 2013]. This poses many unique, costly and immediate challenges, from maintenance and storage to the development of algorithms that can scale to such volumes of data. To put this growth in context, Moore's law observes that the number of transistors in integrated circuits doubles every two years. It is no wonder that sequence archives around the world are having trouble keeping up.

The output of a modern DNA sequencing experiment consists of millions of short sequences, typically 30 to 100 characters (or bases) each. Coupled with metadata and a quality score these sequences are often referred to as *reads*, dating back to a time when nucleotides were identified by physically reading the output of a sequencing experiment [Flicek and Birney 2009]. Metadata associated with a sequence contains information such as sequence identifiers and optional machine-specific content. The quality values state how confident the sequencing machine was for each reported base. The output from a sequencer is typically arranged in a standardized format, for example, the FASTQ format [Cock et al. 2010] which is used in the most recent generation of Solexa/Illumina high-throughput sequencers. An example FASTQ sequence is shown in Figure 6.2, the first line contains the metadata, this is followed by the actual nucleotide sequence, then a plus symbol, and finally, the quality scores.

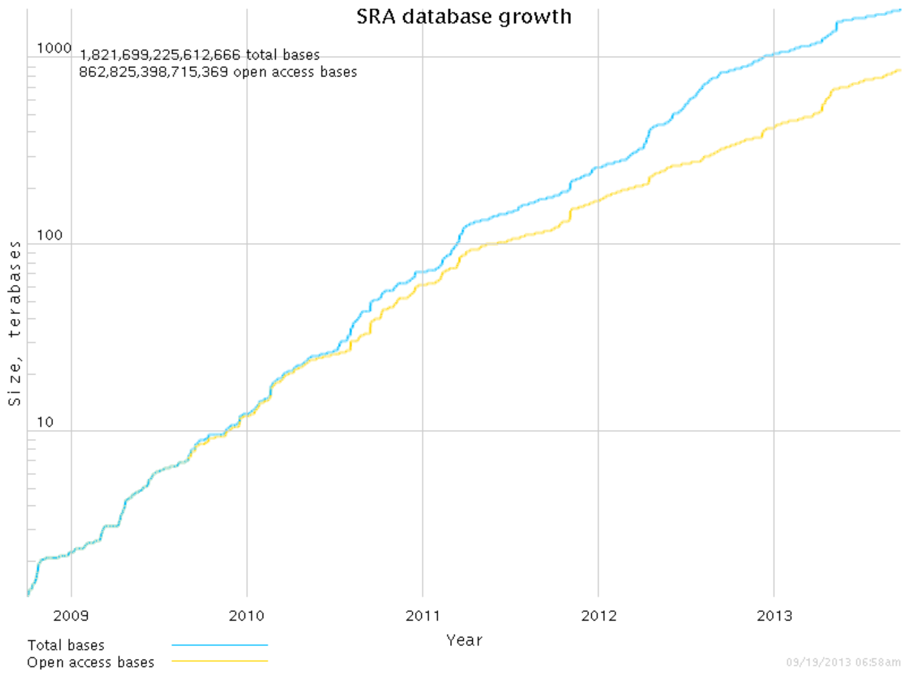


Figure 6.1: The Sequence Read Archive (SRA) database growth from its conception in 2009 to the present day.[1]

At a high level, a sequencer takes samples or reads randomly across a DNA fragment. As the reads are not evenly distributed a much larger sample set is required with respect to the size of the input sequence. This is called *coverage*. A typical sequencing of a human genome aims for about 30 times coverage, otherwise there will not be enough overlap between the reads to facilitate accurate assembly. Sequencing with a high coverage leads to high levels of redundancy, which, in turn gives us a good idea of how compressible a collection is. It is interesting to note that the reads themselves are not very useful. We need to reconstruct the DNA sequence from the reads before we can perform any sort of biological analysis, such as aligning against another sequence and searching for similarities. This process is called *assembly*, where an algorithm determines the most likely position for each read in a sequence and then constructs it from the alignment. This sequencing data is not neglected post assembly. It is still valuable to store, if only to verify a the validity of an assembled sequence in the future.

Sequence archives store their data in compressed form. In most cases they use a general purpose tool such as gzip, which offers little improvement over a trivial static encoding of the reads using two bits per base. There are some practical reasons for using gzip however, most notably that the file format is easily handled on most desktop computers.

Compression of read data is a very active area of research. Compression methods

[1]Image sourced from the Sequence Read Archive Overview http://www.ncbi.nlm.nih.gov/Traces/sra/

```
@SRR002271.2580 FC2012M_R1:1:1:582:726/1
AAACCAGCACATCATGCACATGTACCCCTGAACTTA
+
IIIIIIIIIIIIIIIIGIIII9IA;II59+-CIII
```

Figure 6.2: An example of high-throughput sequencing output using the FASTQ file format.

can be classified as belonging to one of two categories: reference and non-reference based algorithms. A reference-based algorithm aligns each read to a reference genome. Then, each read is encoded in terms of its difference from the part of the genome to which is aligns best. As there are usually few differences between individual genomes in the same species, reference-based compression can achieve a very compact encoding [Fritz et al. 2011, Kozanitis et al. 2011, Yanovsky 2011, Jones et al. 2012]. However, reference-based compression is not always feasible or desirable. First, the reference sequence is separate from the encoding. If the reference becomes corrupt or even misplaced, the reads can no longer be decoded. The reference sequence can be very large, and may not be stored locally, so decoding depends on an internet connection. Perhaps most importantly, some experiments do not have references, for example, in metagenomics, where the sequence sample contains unknown organisms or communities of organisms, such as, bacteria, a human gut sample, or from seawater.

Non-reference based approaches vary, but typically treat compression as a string problem and employ techniques such as Huffman coding [Tembe et al. 2010, Deorowicz and Grabowski 2011a], LZ77 [Chen et al. 2002], or BWT [Mantaci et al. 2005, Cox et al. 2012]. Another approach is to perform reordering of the reads then compress with a general purpose compression tool such as gzip or bzip2 [Hach et al. 2012]. Some schemes focus solely on compression of the read data [Cox et al. 2012]. Others focus on compressing whole sequencer output (reads, quality scores and other meta-data), see Bonfield and Mahoney [2013] for a recent review. There has also been a recent focus on quality score compression [Wan et al. 2012, Janin et al. 2013, Ochoa et al. 2013, Cánovas and Moffat 2013].

In this chapter we focus on read compression and present two novel algorithms for the task. In Section 6.1 we introduce Faust, a scan-based LZ-style compression algorithm. First we detail the algorithm, then discuss practical techniques to implement it efficiently. In Section 6.2 we introduce Afin, an extension of Faust that performs a reordering of the reads in order to gain an improvement in compression and decoding throughput. In Section 6.3 we evaluate both algorithms on a large real-world read database against current state-of-the-art compression algorithms and popular general-purpose baselines. Finally, in Section 6.4 we conclude and provide directions for future work.

6.1 FAUST

We now present Faust, an algorithm capable of scaling to large real-world high-throughput sequencing experiments. At a high level the algorithm resembles a Lempel-Ziv parse with a slight twist. Instead of encoding a string in terms of previously occurring substrings as in traditional LZ77 outlined in Section 2.3.1. Faust encodes full reads against previously occurring reads. To achieve this the collection is divided into fixed-size blocks. We construct an index for each block in turn and perform a scan of all the reads up the start of the current block. The reads in a block are identified as *block reads*. Reads in a scan are identified as *scan reads*. For each scan read we query the index to find a set of block reads that will compress well with respect to the current scan read – recording the best matches as we go. Once the scan is complete we compress the block reads against their most suitable matching scan read. This process is repeated for every block in the collection. In the next section we formally describe the compression algorithm. This is followed by a discussion of practical methods to improve run-time and compression. Finally, we discuss efficient methods to compress a read in terms of another read.

6.1.1 Compression

Consider a collection of n reads each of length l. We split the collection into m fixed-sized blocks of length b, where $m = \lceil n/b \rceil$ and each block is comprised of $\lfloor b/l \rfloor$ reads. Note that the final block could be smaller than b, however, this will have no impact on the algorithm. Algorithm 7 gives the pseudo code for the compression routine. Throughout the description we will refer to the current block as B, and all previously occurring reads as S. For example, block k, B_k, is comprised of reads contained in the range $R[kb, .., (k+1)b-1]$ and S_k, contains the reads in the range $R[0, .., kb-1]$. A visual representation of R and S blocks is shown in Figure 6.3. Note the absence of S in the first block.

We compute the suffix array of B_k, then scan each read in S_k. For each read s we compute its matching statistics [Abouelhoda et al. 2004] with respect to the current block, B_k, that is, $\mathrm{MS}_{s|B_k}$. The matching statistics corresponds to an array of triples, (s, e, l), where $s, .., e$ maps to the range in the suffix array of B_k such that the suffixes beginning at positions $\mathrm{SA}_{B_k}[s_i]$, $\mathrm{SA}_{B_k}[s_i + 1]$, .., $\mathrm{SA}_{B_k}[s_e]$ are prefixed with the text $s[i, l_i]$. This is used to compute a set of candidate reads in B_k that could use s as a reference to encode against.

For each block read we maintain a pointer to the most suitable scan read to compress against. This pointer is initially set to *null*. If the pointer is still *null* at the end of a scan then we encode the read using a static two bit code. We compress each candidate read against s, the current scan read, and update its pointer if s provides better compression than its existing reference. Figure 6.4 expands on this. In the figure we have two reads A and B which are used as references to encode two reads in the current block. At the

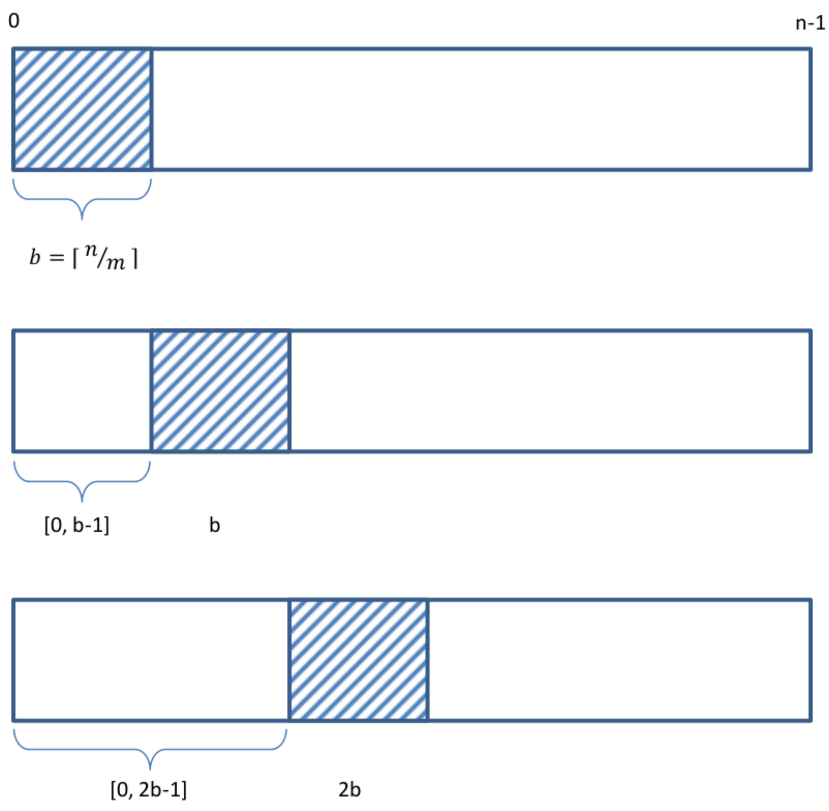


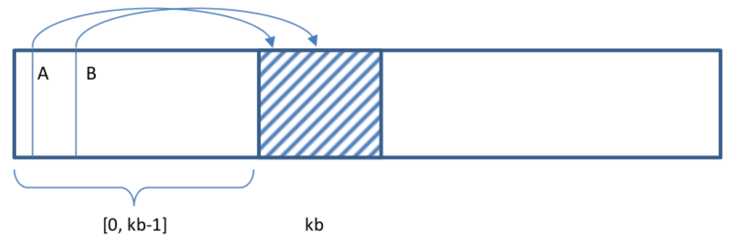
Figure 6.3: A visual representation of the first three blocks and their preceding scan reads.

bottom of the figure we show the values of the reference and block reads aligned by their shared factor. The red bases in each reference read can be safely ignored and the blue bases in each block read need to be retained in order to reconstruct the read during decoding. Clearly not all block reads will be assigned a pointer by the end of each scan. In fact, the first block, $B_0 = R[0, b-1]$ will have no pointers assigned – as there is no prefix of reads to scan before it. In Section 6.1.1 we describe efficient methods to compress a read in terms of another.

Improving Candidate Selection

Candidate selection represents a significant run-time bottleneck during compression. The matching statistics of a scan read with respect to a block generates a large number of candidates, many of which are not useful. Figure 6.5 plots the distribution of shared factor lengths between scan reads and block reads for a Faust run on ERA015743, a collection of approximately 670 million reads of length 100. Note that the vast majority of length values is greater than 80. To avoid redundant comparison of scan reads that share a short factor, we filter candidates with factors less than a specified length.

A further method to improve run-time and compression performance is to use a



```
A reference   ACTCTTCTGTCTGGTTTCTTTGTGACTTTCCTGCTGTGCTGTAAGATAGT
A block          AATCTGTCTGGTTTCTTTGTGACTTTCCTGCTGTGCTGTAAGGGAAAACA

B reference      CTGCATCCTCCTCGGTAAATATGCTTATAAAGTATACCTCCCATAAAAACTCATTA
B block       GAACTGCATCCTCCTCGGTAAATATGCTTATAAAGTATACCTCCCTTGCTACATTA
```

Figure 6.4: Top: A visual representation of two reads, A and B which are currently the selected candidate reference reads for compression of two reads in the block kb. Bottom: The two sets of reference and block reads aligned by their shared common factor. The black base identify the shared factor. The red and blue bases denote their differences.

dynamic threshold that will increase as we process the collection. The idea here is that blocks at the beginning of the collection have few scan reads to select from, so we use a small length threshold in order to expand the number of candidates. Conversely, blocks closer to the end of the collection have a much larger pool of reads to select from so we do not need to be as selective. As you can see from Figure 6.5 a great number of reads share long common factors. In this case we can set a high length threshold, essentially removing all shorter matches from the selection pool.

Reverse Complement Matching

In a sequencing experiment it is not known which DNA strand a given read has been sequenced from. However, sequences from opposite strands are related to each other in a precise way, with a sequence on one side being called the reverse complement of the sequence on the other. These strands are held together by hydrogen bonds where Adenine (A) complements Thymine (T), and Cytosine (C) complements Guanine (G). As a small example, given the sequence AACG, its complement is TTGC, and its reverse complement is CGTT. A larger example is shown in Figure 6.6 displaying two complementary DNA strands. Note that each strand is read in opposite directions. We can check if the reverse complement of each scan read is suitable to compress against block reads, essentially doubling the number of scan reads for each block. Allowing reverse complement matching requires us to store a single bit identifier per read during encoding and has a negative an impact on run-time during compression, by up to 25% on average, see Section 6.3.1.

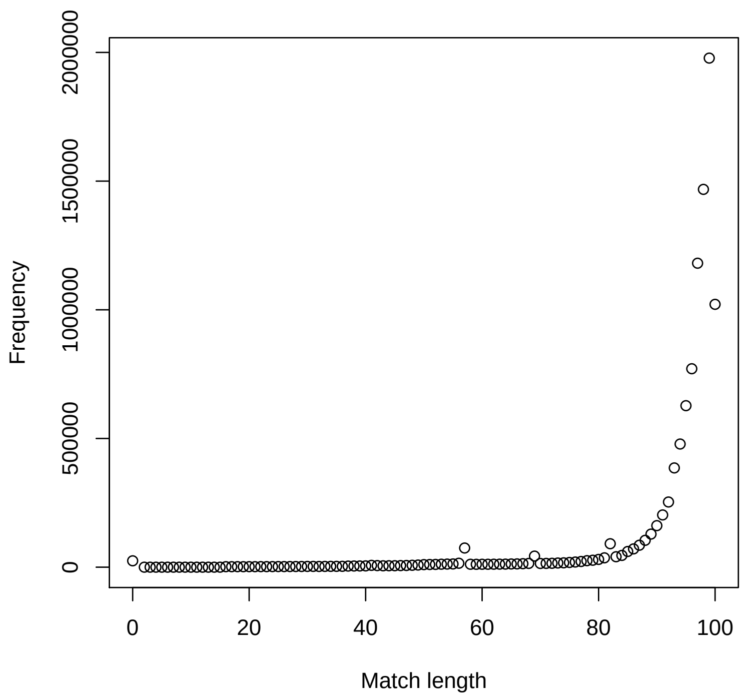


Figure 6.5: Match length distribution between scan and block reads for a full Faust encoding of ERA015743, a collection of approximately 670 million reads of length 100 representing 40 times coverage of a human genome.

```
AAACCAGCACATCATGCACATGTACCCCTGAACTTA
TAAGTTCAGGGGTACATGTGCATGATGTGCTGGTTT

              A = T
              T = A
              C = G
              G = C
```

Figure 6.6: Top: Two complementary strands of DNA. Bottom: Mapping of complementary base pairs. Note that each strand is read in opposite directions.

Read Representation

To represent the encoded reads compactly there are a number of scenarios to consider. Initially we will discuss the most common read representation, where we encode one read in terms of another. Then we will cover two important corner cases: when we find no reference for a block read, and when there is a full match between a block read and a scan read.

Algorithm 7 FaustEncode compresses a collection of reads R, each of length l, using a candidate selection length threshold t.

```
1: function FaustEncode($R, l, t$)
2:     for each block in R do
3:         for $i \leftarrow 0$ to $|block|$ do
4:             $ref[i] \leftarrow \emptyset$
5:             $nbits[i] \leftarrow 2l$        ▷ Reads without a reference are coded in 2 bits per base.
6:         for each read in R[0, block) do
7:             $C \leftarrow \emptyset$
8:             for $i \leftarrow 0$ to $l - t$ do
9:                 $(s_i, e_i, l_i) \leftarrow MS_{read|block}[i]$
10:                if $l_i > t$ then
11:                    $C \leftarrow C \cup$ Candidates$(s_i, e_i)$
12:            for $i \leftarrow 0$ to $|C|$ do
13:                $n \leftarrow$ Encode$(read, C[i])$
14:                if $nbits[C[i]] < n$ then
15:                    $ref[C[i]] \leftarrow read$
16:                    $nbits[C[i]] \leftarrow n$
17:        CompressBlock$(block, ref)$
```

Unlike an LZ parse, where factors are encoded as doubles or triples, we require a 5-tuple. First, we need to store the index of the scan read that we are using to construct the current block read. Then, we need information about the shared factor, specifically, the alignment position in both reads and the length of the factor. A visualization of this is shown in Figure 6.7. At the top of the figure there are two reads $A_{reference}$ and A_{block}, which are both aligned by their shared factor at positions 5 and 2 respectively. The length of the factor is 40 bases. The red bases in $A_{reference}$ can be ignored and the blue bases in A_{base} are required to construct the A_{block} during decoding. The middle of the figure displays the fields we require in the 5-tuple. At the bottom we show the actual 5-tuple used to encode A_{block} in terms of $A_{reference}$. Note that the final field is the concatenation of the blue non-factor bases. We can infer the position of the non-factor bases (that is, if they are positioned to the left or right of the shared factor) from the alignment of the block read position. In the current example, two bases occur on the left of the shared factor, as the block read aligns at index 2, and the remaining bases are to the right.

There are two scenarios where an alternative encoding is required. The first is when a block read has not been assigned a reference read. It is quite possible that we do not find a suitable reference during a scan. In fact, this will always occur for the reads in the first block, as there are no reads preceding it. In this case we encode the read using a two bit static code for each base. A further possibility is that a block read finds a number of candidate reads during a scan, however, the two bit static code was more efficient. The second scenario is where a read and its reference fully match. In this case there is no need to store the 5-tuple, we store the reference value and a flag indicating that the reads fully match.

Encoding the Read

In this section we describe the specific way in which 5-tuples are encoded. We denote the length of the read as l, the length threshold used during compression as t and the position of the aligned shared factor in the reference read and the read to encode as rf and ef respectively.

The reference field can be represented in terms of the number of reads we have currently processed. For example, if we are processing a block k and each block is comprised of n reads we can represent the reference field in

$$\lceil \log(kn+1) \rceil \; bits$$

We reserve the value kn to denote that the current read has no reference and that each base is represented as a two bit static code. It is important to note that we can skip the reference field altogether for the reads in the first block. We know they will never be assigned a reference so we encode each read as with a static code and save $n\lceil \log(n+1) \rceil$ bits. The example in Figure 6.7 does not include reverse complement matching. However, as explained earlier, it can be incorporated with a single binary flag to indicate if we are compressing against the plain reference read or the reverse complement of the reference read. We store both factor alignment positions in

$$\lceil \log(l-t+1) \rceil \; bits$$

Note that we are using the value $(l-t)$ to signify that there is a full match between the scan and block read – the second corner case described above. This also acts as an indicator to the decoder that we do not need to read the final position and length values of the tuple for the current read.

The length field depends entirely on the largest index value of both read positions. Using the example in Figure 6.7 the largest alignment index is 5. Therefore we store the position values in $\lceil \log(40-5+1) \rceil$ bits, or more formally

$$\lceil \log(l - max(rp, ep) + 1) \rceil \; bits$$

6.1.2 Decompression

Decompression is straightforward. Initially we decode the first block of reads that were compressed in two bits per base. Then, we process the rest of the collection a read at a time. For a given compressed read we first decode a reference field. If the reference is greater than the number of processed reads we know that the current read is compressed

```
                 01234567890…
A reference      ACTCTTCTGTCTGGTTTCTTTGTGACTTTCCTGCTGTGCTGTAAGATAGT
A block             AATCTGTCTGGTTTCTTTGTGACTTTCCTGCTGTGCTGTAAGGGAAAACA
                    0123456789…


      (Reference, Reference position, Block position, Length, [Non-factor bases])


                        (A reference, 5, 2, 40, 'AAGGAAAACA')
```

Figure 6.7: Top: The two reads $A_{reference}$ and A_{block} aligned by their shared common factor. Middle: The fields used to represent a a compressed read. Bottom: The actual values used to compress A_{block} in terms of $A_{reference}$.

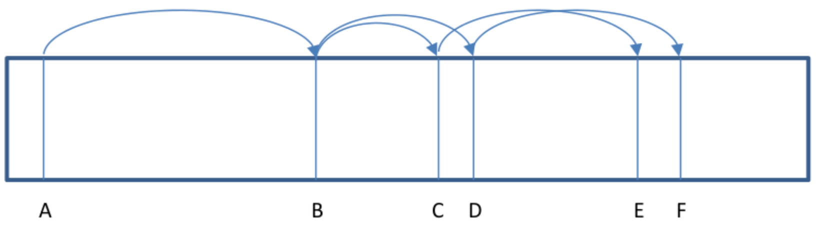
using a static two bit code. Otherwise, we fetch the reference read, either from memory or disk, and continue decoding the first alignment position. There are two remaining scenarios. The read could either be a full or partial match with respect to its reference. If it is a full match we simply write the reference read to the output stream. In the case of a partial match we need to decode the remaining position value, length value and the non-factor bases before we can reconstruct the new read.

The most costly operation during decoding is fetching the reference read. Currently our implementation works entirely in memory by representing uncompressed reads statically in two bits per base. If physical memory is limited or the complete collection can not fit in memory we can delay fetching reference reads until we have reached the limit of available memory, then access each needed reference a batch, with a single scan of the currently decoded file on disk.

6.2 AFIN

We now describe Afin, an algorithm that introduces an extra processing step that performs a reordering of the reads in such a way that we can almost completely eliminate the reference fields from a Faust encoding, obtaining a significant improvement in both compression performance, decoding throughput and memory requirements.

The core idea is to identify relationships between reads and their references. If we can place reads that share the same reference read close to each other, we can reduce the number of bits needed for the reference field, or possibly eliminate it altogether. An added bonus is that during decoding we avoid the costly operation of fetching reference reads from random positions in the already decoded collection. Because of the way Faust processes the collection, a reference for a given read can only be found in the reads of its prefix up to beginning of the block that the read is currently in. As we continue to process blocks these reads could also serve as a reference themselves. This relationship between reads and their references enables us to construct trees of reads. The concept is



```
A          GACTCTTCTGTCTGGTTTCCACATTACCCAGTTATATAAAGACAAATAGT
B          ACAAACTCTGTCTGGTTTCCACATTACCCAGTTATATAAAGACAAGGAAA
C     TCGAGACAAACTCTGTCTGGTTTCCACATTACCCAGTTATATAAAGACAA
E     TCGAGACAAACTCTGTCTGGTTTCCACATTACCCAGTTATATAAAGTTAC
D       CAAACAAACTCTGTCTGGTTTCCACATTACCCAGTTATATAAAGACAGA
F               CTCTGTCTGGTTTCCACATTACCCAGTTATATAAAGACAGAAAACTGTA
```

Figure 6.8: Top: A visual representation of a single Afin graph. In this example read A acts as a reference for read B. Furthermore, read B acts as a reference for two reads, read C and D and so on. Bottom: The aligned read graph rooted at A in depth first order.

Table 6.1: An example of the (read, reference) array R sorted by the reference field, and the corresponding array T generated for the example graph in Figure 6.8.

i	R_{read}	R_{ref}	$T[read]$
0	A	$\emptyset$	A $\rightarrow$ 1
1	B	A	B $\rightarrow$ 2
2	C	B	C $\rightarrow$ 4
3	D	B	D $\rightarrow$ 5
4	E	C	E $\rightarrow$ $\emptyset$
5	F	D	F $\rightarrow$ $\emptyset$

demonstrated in Figure 6.8, where read A acts as a reference for read B. Furthermore, read B acts as a reference for two reads, C and D, and so on. When we reorder the reads and align them by their shared factor it is easy to see that we have successfully clustered a group of reads. In the next section we describe the algorithm to simulate the traversal of each graph. Then, we discuss how decoding works in the absence of the reference field.

6.2.1 Compression

The pseudo code for this algorithm is outlined in Algorithms 8 and 9. From the Faust output we generate an array R of (read, reference) pairs, where $R[i].read$ corresponds to the i^{th} read in the collection and $R[i].reference$ is its reference assigned by Faust. We then sort these pairs by their reference field. This clusters groups of reads that share the same reference. We scan the sorted pairs and in a complimentary array, T, store a mapping of $read$ to the position of its first occurrence as a reference in R. That is, for the read i, the value $T[i]$ gives us the position in R where i first occurs as a reference.

Algorithm 8 AfinEncode accepts an array R of (read, reference) pairs, where $R[i]$.read corresponds to the $i'th$ read in the collection and $R[i]$.reference is the reference read used by Faust during compression.

```
1: function AfinEncode(R)
2:     sort($R$) by reference field
3:     $T[R[0].reference] \leftarrow 0$
4:     for $i \leftarrow 1$ to $|R|$ do
5:         if $R[i-1].reference$ != $R[i].reference$ then
6:             $T[R[i].reference] = i$
7:     for $i \leftarrow 0$ to $|R|$ do
8:         if $R[i].reference == \emptyset$ then
9:             EncodeRead($R[i]$)
10:            Traverse($i, T[R[i].read], R, T$)
```

Algorithm 9 The recursive traversal function used in AfinEncode. Four parameters are required. The current parent and child index into R, the sorted reference array R, and the array T.

```
1: function Traverse($parent, child, R, T$)
2:     if $child == \emptyset$ then return
3:     while $R[child].reference == R[parent].read$ do
4:         EncodeRead($R[child]$)
5:         Traverse($child, T[R[child].read]$)
6:         $child \leftarrow child + 1$
```

As an example, Table 6.1 displays the R and T arrays for the for the tree in Figure 6.8. On the left, R is sorted by the reference field, where A has no reference, A references B, B references C and D, etc.. On the right we have the mapping from a read to its first occurrence in R, where A maps to index 1, B maps to 2, C maps to 4, and so on. Note that E and F are not used as a reference so they do not map to a position in R.

Clearly a tree root is a pair in R that has not been assigned a reference. For each tree, starting at the root, we preform a depth first traversal of its tree. We simulate this traversal using T, which lets us jump into the sorted pairs and process each cluster of reads that share the same reference. For each child node, if $T[R[child].read]$ is assigned a mapping we know that the current read has children, that is, it is used as a reference, so we recurse on *child* and continue the traversal. Otherwise, it is never used as a reference so the read corresponds to a leaf node in the tree and we backtrack in the recursion. We traverse each tree in the same manner.

Read Representation

Now that we have removed the reference field and reordered the collection, we need to store a small amount of information with an encoded read in order to describe the structure of each tree. To achieve this we identify four types of tree nodes.

1. A root node that has not been assigned a reference, $R[i].reference = \emptyset$.
2. An internal node that acts as a reference for a number of child reads and has been assigned a reference, $T[i] \neq \emptyset$ and $R[i].reference \neq \emptyset$.
3. A plain leaf node where no reads use it as a reference, that is, $T[i] = \emptyset$ and $R[i].reference = R[parent].read$.
4. A leaf node that is at the end of a run of nodes sharing the same reference, that is, $T[i] = \emptyset$, $R[i].reference = R[parent].read$, and $R[i{+}1].reference \neq R[parent].read$.

We replace each reference field with a code identifying its node type. As there are only four types we can represent each node type using a two bit header. When translating from the Faust read representation to Afin, care must be taken if a dynamic threshold has been used in the Faust encoding. These runs increase the length threshold once they have processed half of the collection. The problem is that Afin performs a reordering of the reads, removing the middle point. That is, there is no longer an effective way to determine which threshold value was used for each read. This is important as the two alignment fields rely on this value when encoding and decoding. We solve this by encoding each read with respect to the largest threshold value used during the Faust encoding. An alternative solution would be to include a flag bit in a similar manner to the reverse complement bit. This evaluation is left for future work.

6.2.2 Decompression

Decompression is slightly more complicated in Afin than Faust, as we now need a way to traverse each tree and determine each node reference. The pseudo-code for decoding is given in Algorithm 10. We maintain a stack of reads which are used to simulate the depth first traversal of each tree. Initially, only the root node of a tree is on the stack. As a root node has no assigned reference it is coded in two bits per base. If an internal root node is encountered, it is decoded with respect to the current reference, that is, the read at the top of the stack, then we push the internal node onto the stack. In the case of a plain leaf node, decoding is carried out respect to the current reference, and the process continues without pushing. Finally, if a leaf node is encountered at the end of a run of leaf nodes that share the same reference (described as node type 4 earlier) its decoded like a plain leaf node, after which the current reference is popped from the top of the stack – as there are no other reads that use it as a reference.

6.3 Experiments

Both the Faust and Afin compression schemes were implemented for varied fixed and dynamic length selection thresholds as well as encoded with and without reverse complement

Algorithm 10 AfinDecode decodes a compressed Afin stream.

```
function (AfinDecode)
    $top \leftarrow 0$
    while there are more reads to decode do
        $read \leftarrow$ NextRead()
        if read is the root of a graph then
            $S[0] \leftarrow read$                ▷ Clear stack and push the new read to the top.
            $top \leftarrow 1$
            WriteRead($read$)
            continue
        $reference \leftarrow S[top - 1]$
        DecodeRead($read, reference$)
        if the read is an internal reference node then
            $S[top] \leftarrow read$              ▷ Push the current read to the top of stack.
            $top \leftarrow top + 1$
        if the read is the last leaf node in a run then
            $top \leftarrow top - 1$              ▷ Pop the top read from the stack.
```

matching. Each run is identified by A-T[-RC], where A is the algorithm used, T defines the threshold values used, and the optional -RC suffix indicates whether reverse complement matching was used.

We evaluated our compressors against BEETL [Cox et al. 2012], a current state-of-the-art read compression utility using two variants, reverse lexicographical ordering (RLO) and same as previous (SAP) ordering. BEETL is a two-stage compression algorithm. The first stage preforms disk based suffix sorting to compute the BWT of a collection. beetl-rlo sorts the collection such that the reverse of each read is in lexicographical order, then computes the BWT and performs run length-encoding (RLE) of the output. beetl-sap performs an implicit permutation the collection to obtain a more compressible BWT text. In the second stage its output is compressed using PPMd, a statistical compression algorithm. This was achieved via 7zip using the following arguments −m0=PPMd -mo=16 -mmem=2048m. In addition, we compared against a number of state-of-the-art read compression algorithms. SCALCE [Hach et al. 2012], using arguments -B55G (fast) and -B10G (slow) respectively. fastqz [Bonfield and Mahoney 2013] using arguments e (fast) and c (slow). fqzcomp [Bonfield and Mahoney 2013] using arguments -n1 -q2 -s1 (fast) and -n2 -q3 -s8+ -b (slow), SRComp [Selva and Chen 2013] and Quip [Jones et al. 2012].

We compare against two common general purpose compression tools, gzip and 7zip. They are based on LZ77 parsing, however, gzip restricts its dictionary to 32 KB, has a maximum factor length of 255 characters and uses a Huffman code to compress each LZ pair. 7zip can extend its dictionary and look-ahead buffer to maximum of 4 GB and compresses pairs using range coding. gzip was executed with −9 and 7zip, −mo=16 -mmem=2048m. Finally, we compare against an implementation of RLZ from Chapter 3 which was altered to support read data. First, dictionary generation was modified to sample full reads. We used two dictionary sampling methods: uniform and randomly distributed. Pair encoding was also changed. Position values were encoded in $\log(|dictionary|)$ bits and length values were encoded in $\log(|read|)$ bits. Runs can be identified by RLZ-D-T where D corresponds to the size of the dictionary and T is either U or R, indicating, respectively, a uniform or randomly sampled dictionary.

Experiments were run on ERA015743[2], the same read collection used in Cox et al. [2012]. ERA015743 is a 135 GB sequencing of a whole human genome using paired 100 base reads. The collection is comprised of 670 million reads corresponding to an approximate coverage of 40 times. Each read was extracted from its FASTQ markup and concatenated into a single string. Once stripped of metadata and quality scores, a 53 GB file remains.

Experiments were conducted on a 16 core Intel E5-2670 CPU with 64 GB of main memory running Linux (CentOS 6.3) and g++ (GCC 4.7.2) using -march=native -O3. Peak memory usage was recorded using libmemusage, a tool from GNU glibc that hooks core memory allocation functions and records run-time statistics. All memory experiments were preformed separately to timing experiments. Time values were reported using wall clock time. We define compression ratio as a percentage of the encoded output against the original collection size.

6.3.1 Compression Results

Compression results for Faust, Afin and each of the baseline runs are shown in Tables 6.2 and 6.3. All Faust runs reported were constructed using a block size of 2 GB. Combinations of fixed and dynamic candidate selection thresholds were used. In Table 6.2 the top table displays runs where no reverse complement matching was preformed and the bottom displays runs where it was enabled. Overall, the Faust run that achieved the best compression result was faust-16/32-rc, however, it was also the slowest, with an overall time of 32 hours. This slow run-time can be attributed to the increased number of read candidates that result from a very small selection threshold. The Faust run with the most practical of time and space trade-off was faust-32/64-rc, compressing only 0.3 % worse than faust-16/32-rc while running in half the time at 18 hours. The general trend for all combinations of run types is that a lower selection threshold gives better the compression, at the cost of run time. Results using a dynamic threshold satisfied our hypothesis that increasing the selection threshold once we process later blocks in the collection would have little to no effect on compression and demonstrate an improvement in run-time. For example, faust-32-rc compressed ERA015743 to 5.31 GB and ran in 31 hours, while faust-32/64-rc compressed it to 5.35 GB and ran in 18.51 hours.

Figure 6.9 plots per block compression for Faust runs with varied fixed (left) and dynamic (right) selection thresholds. Note each initial block compresses to 25% as the

[2]http://www.ebi.ac.uk/ena/data/view/ERP000460

Table 6.2: Encoding and decoding results for Faust and Afin experiments for varied fixed and dynamic selection threshold values and optional reverse complement checking on ERA015743, a 53 GB collection of reads.

		Encoding		Decoding	
Method	**Enc. (GB)**	**Time (Hrs.)**	**Peak (GB)**	**Time (Min.)**	**Peak (GB)**
faust-32	5.68	21.84	34.95	14.45	13.27
faust-64	6.22	11.11	35.08	14.36	13.27
faust-70	6.42	10.51	34.82	14.35	13.27
faust-80	7.28	10.24	35.21	14.68	13.27
faust-16/32	6.11	25.01	33.51	15.09	13.27
faust-32/64	6.32	13.78	33.46	14.54	13.27
faust-35/70	6.38	12.29	33.46	14.65	13.27
faust-40/80	6.72	13.27	33.46	14.26	13.27
afin-32	3.99	0.17	22.81	11.06	5.28e-3
afin-64	4.53	0.18	22.81	10.43	5.26e-3
afin-70	4.73	0.18	22.81	10.20	5.26e-3
afin-80	5.59	0.19	22.81	10.70	5.26e-3
afin-16/32	4.43	0.17	22.81	12.45	5.26e-3
afin-32/64	4.49	0.18	22.81	12.15	5.27e-3
afin-35/70	4.50	0.17	22.81	12.69	5.26e-3
afin-40/80	4.90	0.18	22.81	12.02	5.26e-3

		Encoding		Decoding	
Method	**Enc. (GB)**	**Time (Hrs.)**	**Peak (GB)**	**Time (Min.)**	**Peak (GB)**
faust-32-rc	5.31	31.60	35.81	18.46	13.27
faust-64-rc	5.62	18.02	35.76	17.96	13.27
faust-70-rc	5.71	13.54	34.40	18.85	13.27
faust-80-rc	6.30	10.24	34.64	17.85	13.27
faust-16/32-rc	5.26	32.63	34.24	18.01	13.27
faust-32/64-rc	5.35	18.51	34.74	17.95	13.27
faust-35/70-rc	5.36	16.65	34.78	17.88	13.27
faust-40/80-rc	5.49	15.49	34.07	17.80	13.27
afin-32-rc	3.62	0.15	22.81	8.16	5.27e-3
afin-64-rc	3.93	0.15	22.81	8.38	5.27e-3
afin-70-rc	4.02	0.15	22.81	8.38	5.26e-3
afin-80-rc	4.61	0.16	22.81	8.85	5.25e-3
afin-16/32-rc	3.53	0.18	22.81	9.83	5.28e-3
afin-32/64-rc	3.60	0.18	22.81	9.06	5.26e-3
afin-35/70-rc	3.53	0.15	22.81	9.56	5.28e-3
afin-40/80-rc	3.73	0.16	22.81	9.45	5.25e-3

Table 6.3: Encoding and decoding results for baseline compressors, gzip and 7zip, RLZ, BEETL, SCALCE, fastqz, fqzcomp, SRComp and Quip on ERA015743, a 53 GB collection of reads.

		Encoding		**Decoding**	
Method	**Enc. (GB)**	**Time (Hrs.)**	**Peak (GB)**	**Time (Min.)**	**Peak (GB)**
2bpb	13.40	0.15	4.10e-6	-	-
gzip	15.01	16.38	4.36e-3	9.53	4.43e-3
7zip	12.86	17.50	0.698	15.57	1.82e-2
rlz-0.1g-u	14.20	26.20	0.50	12.09	0.1
rlz-0.5g-u	12.62	25.17	2.50	12.39	0.5
rlz-1.0g-u	11.64	24.50	5.00	11.45	1.0
rlz-2.0g-u	10.37	26.11	10.00	11.50	2.0
rlz-0.1g-r	14.20	26.89	0.50	12.01	0.1
rlz-0.5g-r	12.61	25.38	2.50	13.51	0.5
rlz-1.0g-r	11.94	25.00	5.00	12.33	1.0
rlz-2.0g-r	10.26	25.45	10.00	11.16	2.0
beetl-rlo	6.49	46.53/2.33	46.3/2.10	121/ -	2.05/ -
beetl-sap	6.95	27.68/2.63	44.1/2.10	64/3954	2.05/40.20
SCALCE (fast)	1.20	2.85	10.00	78.03	1.48
SCALCE (slow)	0.85	3.50	55.00	74.52	1.48
fastqz (fast)	15.12	1.11	0.01	54.00	0.01
fastqz (slow)	11.01	1.20	1.65	712.12	0.23
fqzcomp (fast)	25.03	1.43	0.15	371.01	0.15
fqzcomp (slow)	26.11	1.37	0.15	369.01	0.23
SRComp	0.76	0.80	54.19	8.98	2.06
Quip	30.02	3.17	0.58	206.01	0.61

block is encoded using two bits per base, and subsequent blocks consistently improve as the scan range increases. When using a length threshold of 80, compression actually gets worse after the first block before smoothing out. Note the degrading compression for dynamic threshold runs when the threshold increases halfway through a run. This is quite prominent for larger threshold combinations, for example, 35/70 and 40/80, and has almost no effect with smaller. It is clearly beneficial to include reverse complement matching during encoding, giving a consistent improvement in compression – close to 2 % on average. Table 6.4 displays the percentage of overall reverse complement reads selected during each Faust run. With close to a 40 % selection rate it is an indication that it clearly improves compression. Furthermore, the expected doubling of run-time was not observed, with an actual increase of an acceptable 20-30%.

Peak memory usage during encoding averaged at 35 GB. This is due to our decision not to use any compressed data structures for the block index, primarily to improve run time. Memory usage can be controlled by adjusting the block size giving a nice space-time trade-off. The minor fluctuations in memory across all runs is due to the use of a small dynamic container for storing selection candidates, that is, runs with smaller selection thresholds will have slightly higher peak memory values as the container will allocate more memory. We note that our resource requirements are more efficient than

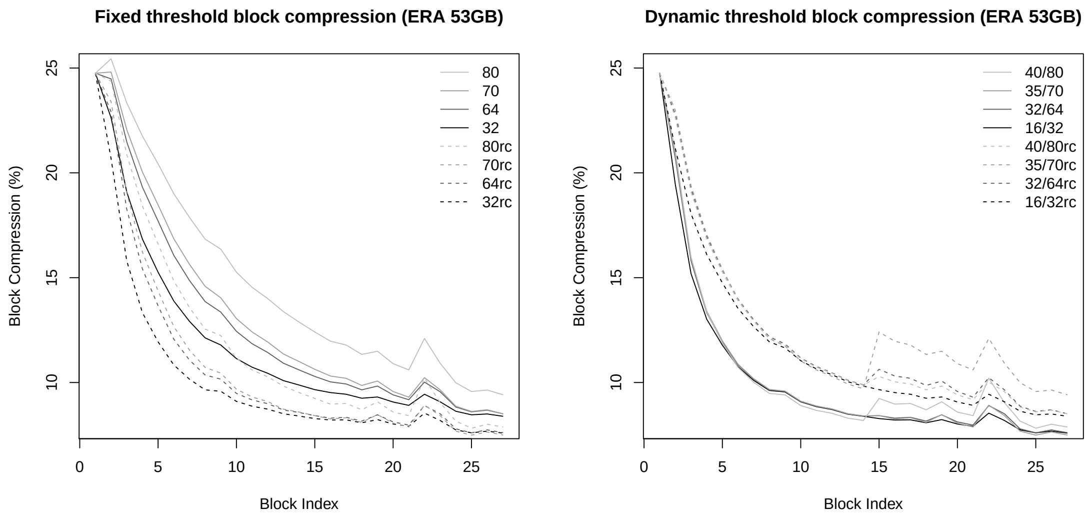


Figure 6.9: Per block compression values for Faust runs for varied fixed (left) and dynamic (right) selection thresholds.

both BEETL runs and run faster than both general purpose algorithms.

Afin runs performed excellently. Each run executed in approximately 10 minutes on top of its corresponding Faust run. As Afin processes Faust output, there was a slight difference in encoding times for runs that did not check reverse complements. This is attributed to the Faust encoding being slightly larger and taking more time to decode. The best overall afin compression result was afin-32/64-rc, reducing faust-32/64-rc from 5.26 GB to an impressive 3.53 GB, almost half the size of the best performing BEETL baseline and competitive with SCALCE and SRComp. Peak memory was fixed for each run at 22.8 GB.

Baseline compression results are shown in Table 6.3. A key observation is that RLZ, the two general purpose compressors gzip and 7zip and baselines fastqz, fqzcomp and Quip struggle to compete against the naive 2 bit static code, with 7zip and the larger dictionary RLZ runs achieving only slightly better compression. gzip uses very minimal memory due to its restricted dictionary. Although 7zip was configured to use its maximum sized dictionary, peak memory usage reached 0.7 GB and only marginally improved on compression compared to gzip. RLZ memory requirements were fixed and correspond to the size of the index built its the input dictionary. There was no noticeable difference in space or time using uniform or randomly sampled reads.

Both BEETL runs achieved competitive compression to an average Faust encoding and compressed 2% worse on average than faust-32/64-rc, the most practical run. Two time results were reported for BEELT runs. For encoding, the first is for computing the BWT and the second for 7zip compression. These times are reversed when reporting decoding. As expected, BWT computation was the dominant cost in terms of time and space

Table 6.4: Percentage reverse complement reads selected during compression.

Method	**% RC**
faust-32-rc	45.17
faust-64-rc	41.39
faust-70-rc	40.15
faust-80-rc	36.88
faust-16/32-rc	46.90
faust-32/64-rc	44.30
faust-35/70-rc	43.84
faust-40/80-rc	42.61

for both runs. beetl-rlo run-time was 19 hours longer than beetl-sap. This is attributed to the cost of the additional run length encoding stage once the BWT was computed. Memory requirements for both runs were close to 10 GB greater than corresponding Faust runs. It is important to note that both BEETL runs perform a reordering of the reads during compression so it is really only comparable to Afin which is far superior in both encoding and decoding. Preserving read order may be desirable during sequence assembly for example, if paired reads are stored together. SCALCE and SRComp runs achieved the best overall compression results, reducing the collection to approximately 1 GB in 2 hours. The two best SCALCE and SRComp runs required 55 GB of memory and were competitive to the Afin runs in terms of space and memory usage.

6.3.2 Decompression Results

Decompression results for Faust and Afin runs are reported in Table 6.2. Faust runs that did not perform reverse complement matching decoded in 15 minutes on average which is faster than 7zip and four minutes slower than gzip. Runs that did include reverse complement matching decoded in an average of 18 minutes. The increased run-time is attributed to the extra computation required for temporally computing the reverse complement of a reference read. Faust decodes an order of magnitude faster than the most competitive baselines. beetl-sap required 66 hours, and beetl-rle was stopped at the 96 hour mark. It is important to note the this difference in run-time is due to the disk-based BWT inversion, which is a costly task. Peak memory for Faust runs was fixed at 13.27 GB, the size of the complete collection encoded in 2 bits per base.

The most notable algorithm was Afin. afin-*-rc runs were superior to all baselines in terms of compressed size and decoding speed. Furthermore, the decoding runs were competitive in memory with gzip, requiring only 5.27 MB on average, which compared to gzip's 4.43 MB is an impressive result. Note Afin's peak memory values fluctuate slightly due the size of the stack required for traversal of each tree. Similar to our observation during encoding, the Afin runs that do not match against reverse complement reads are slightly slower because they are processing a larger compressed file. RLZ runs decoded in 12 minutes on average, faster than Faust runs, slower than Afin and similar to both general purpose algorithms. No decoding results were reported for the 2 bit static code. There is no need to translate it back as it already provides efficient random access.

Although SCALCE achieved significantly better compression, their decoding time was up to 3 times slower than corresponding Faust runs and up to 7 times slower than Afin runs. Peak memory was fixed at 1.48 GB, approximately 10 times less than reported Faust runs, but significantly larger than Afin. Fastqz, fqzcomp and Quip runs achieved the poor decoding times longer than one hour to complete, although, they required little memory. SRComp achieved better decoding results than Faust runs and was competitive in time compared to Afin, decoding in 8.98 minutes and reaching only 2.0 GB peak memory.

6.4 Summary

In this chapter we presented two algorithms for compression of short DNA read collections produced by high-throughput sequencing experiments: Faust, a scan-based algorithm, which encodes reads against suitable previously occurring reads, and Afin, which performs a reordering of the reads in a Faust encoding to exploit the high levels of redundancy throughout a collection. We evaluated both algorithms against current baselines and general purpose compressors on a large real-world sequencing experiment and found that both methods perform well in practice.

In summary, Faust and Afin achieve a 20% improvement in compression compared to general purpose compressors, and encode in similar time. Faust achieves a 2% improvement in compression compared to BEETL, a current state-of-the-art algorithm for read compression. Furthermore, it runs in half the time and in 10 GB less memory. Afin compresses to half the size of each BEETL run and took 10 minutes longer on average than its corresponding Faust run. Note that the Afin run-time includes the initial Faust step, so they are all slightly slower than their corresponding Faust run. Both BEETL runs perform a reordering of the input reads which is not desirable in many contexts, such as collections of paired reads. While Afin does reorder the reads, Faust does not. Faust decoding runs are twice a slow as gzip and very similar to 7zip. Afin decodes quicker than gzip (by an average of 2 minutes) which is a notable achievement. gzip is renowned for sacrificing compression effectiveness for fast decoding speed. Afin not only provides significantly better compression but actually decodes faster. BEETL runs decode in similar time to their encoding results as they need to perform BTW inversion. Both Faust and Afin encode using half the memory required by BEETL. Clearly, both general purpose algorithms compress using significantly less memory, as that is how they were designed, at the sacrifice of compression effectiveness. Faust represents the complete read collection in memory during decoding in a two bit static code which is up to four times more effective than BEETL decoding runs. The best performing baselines were SCALCE and SRComp, compressing to 0.85 GB and 0.76 GB respectively. Encoding time was significantly better than Faust and Afin runs, however, both algorithms required approximately 55 GB of memory where Faust and Afin required 35 GB and 22 GB respectively. In terms of decoding SRComp was competitive to Afin in terms of time, however, used significantly more memory. Finally, Afin decoding requires only slightly more memory than gzip (100 KB on average) which is quite remarkable.

CHAPTER 7

Conclusion

We conclude by giving an overview of the key findings and contributions in this thesis and follow with an outline of directions for future work.

7.1 Contributions

This thesis has presented novel algorithms and data structures for text compression capable of scaling to modern real-world text collections, providing efficient compression, decompression and most importantly, supporting fast random access capabilities. We began in Chapter 2 by identifying a fundamental limitation shared by many existing text compression approaches: the inability to exploit non-local redundancy throughout a text, primarily due to constraints on available memory and the growing disparity between text size and the upper levels of a CPU's memory hierarchy. Algorithms that do provide efficient compression, such as LZMA2, are generally slow at decoding and do not explicitly allow random access, two very undesirable aspects of a compression algorithm.

In Chapter 3 we presented an efficient semi-static compression scheme for large repetitive text collections. We focused our experiments on compression of large web crawls due to the highly redundant nature of such data, however, in Chapter 5 and Chapter 6 we demonstrated that our method is effective across a variety of very different datasets. We described a simple yet effective dictionary generation technique that successfully captures non-local redundancy throughout large texts, providing efficient compression up to half the size of current practical real-world baselines, as well as fast decoding and random access. We proposed a number of coding techniques that offer a variety of trade-offs during compression and decoding such as a byte-oriented codes to achieve fast decoding, and the use of higher-order compressors for example, zlib, to further exploit redundancy throughout a document factorization.

We demonstrated that this approach is suitable for compression in a dynamic environment where a collection grows over time. As long as additional documents maintain similar characteristics as the current documents in a collection we observed that there was an insignificant impact on compression effectiveness. In the scenario where per-document compression degrades below a predetermined threshold we described two technique to enrich the dictionary in order to improve compression. If memory is not constrained we can sample each new document using the technique from 3.1.3 and append them to the current dictionary. An index of the dictionary will need to be recomputed in order to include the new samples during encoding, however, the existing encoding is still valid and avoids the costly process of re-encoding the complete collection. In the case where there are constraints on memory, the dictionary can be regenerated taking the additional documents into consideration. Unfortunately, this approach invalidates the original encoding, and, as a consequence, the collection will need to be compressed once more.

In Chapter 4 we investigated methods to refine the size of a sampled dictionary by removing redundant and unused substrings. This gives explicit control over memory, where we can add useful content to a dictionary in order to improve compression, or to generate a more compact dictionary that achieves similar compression with a smaller memory footprint. Although our sampling method successfully captures non-local redundancy and provides excellent compression, we observed a large proportion of the dictionary was unused or contained redundant content. Furthermore, we identified that there was a strong skew in the samples that were used throughout an encoding, and even among these, we noticed that most of these samples contained repeated material themselves.

We described two techniques to remove redundancy in a sampled dictionary. First, we outlined a pre-processing method where long repetitive substrings are removed from the dictionary before compression is performed. We demonstrated that this method works well in practice, successfully removing large quantities of redundant substrings from a dictionary with no discernible effect on compression effectiveness. Then, we examined dictionary redundancy post-compression. Usage statistics at a sample and character level we recorded during decoding and were used to to make a more informed decision about which components of a dictionary can be removed. We outlined a method where a dictionary is iteratively halved in size by removing least used samples or characters. The collection is then re-encoded relative to the new dictionary. This process is continued until we reach a desired dictionary size of compression effectiveness degrades to a predetermined ratio. We demonstrated that a reduced dictionary gave no discernible impact on compression. Furthermore, a finely tuned dictionary as small as 100 MB can successfully compress both test collections more effectively than all practical baselines.

In Chapter 5 we presented the first practical implementation of the block graph data structure proposed by Gagie et al. [2011]. A block graph is an LZ-style compressed index that is efficient in practice, but also gives strong theoretical bounds in terms of compression and random access. We empirically evaluated our implementation against a number of general-purpose compressors, a variation of RLZ from Chapter 3, as well as LZ-End, a current state-of-the-art compressed index that provides fast random access. Our experiments demonstrated that the index competitive in both theory and practice, achieving compression close to that of LZ-End and gzip while giving superior random access speeds, especially for longer substrings.

Finally, in Chapter 6, motivated by advances in high-throughput sequencing technology which has reduced the time and cost of an individual sequencing experiment and subsequently caused massive growth in genomic databases worldwide. Such databases currently compress read output with general-purpose tools such as gzip. We identified that methods for efficient compression of biological data are quite different to that of natural language texts in order to exploit redundancy throughout a collection, and that general-purpose compressors are not suitable for the task for example, representing the output of an experiment using a naive 2-bit per base code gives equal or better compression effectiveness. We presented two novel algorithms for compression of high-throughput sequencing read data. Faust, a scan-based LZ-style algorithm, which encodes reads against suitable previously occurring reads, and is capable of scaling to large real-world sequencing experiments. Then, we presented Afin, an algorithm that performs a reordering of the reads in a Faust encoding to exploit the high levels of redundancy throughout a collection. We demonstrated that both algorithms improve compression of large real-world read collections and found that each method performs well in practice providing significantly faster decoding compared to general-purpose compressors and BEETL, a current state-of-the-art compression scheme. Faust only achieves a 2% improvement in compression compared to BEETL, but compressed in runs in half the time and in 10 GB less memory. Afin compresses to half the size of each BEETL run and took 10 minutes longer on average than its corresponding Faust run. Most notably, Afin decodes in time quicker than gzip (by an average of 2 minutes). gzip is renowned for sacrificing compression effectiveness for fast decoding speed. Afin not only provides significantly better compression but decodes more efficiently in terms of both time and space.

7.2 Future Work

In this section, we discuss potential areas for future work.

Dictionary generation The problem of generating a representative sample, or dictionary, of a large text collection is very much an open problem. For effective compression a dictionary must sufficiently capture repetitive properties of a text. As collections can potentially be much larger than physical memory this is becoming a non-trivial task.

In Chapter 3 we described a simple yet highly effective sampling technique for relative compression of large text collections. An issue with this approach is that it makes a general assumption that redundancy throughout a collection is uniformly distributed. On collections where this is not the case, compression will may deteriorate severely. There are a number of way to mitigate this issue. Combined with the dictionary refinement techniques discussed in Chapter 4, a simple approach would be to conceptually partition a collection into blocks and record how well each block compresses relative to the current dictionary. Once we have removed redundant content from the dictionary we can add new samples from areas in the collection that were identified to have compress poorly. This could be an iterative approach where we stop after a predefined number of steps, or by reaching some defined equilibrium state between dictionary size and achieved compression.

Generating a dictionary to represent a text has many parallels with grammar compression and the well known *smallest grammar problem* [Charikar et al. 2005]. Constructing a dictionary from a partially constructed grammar, such as Repair, would be an interesting avenue to explore. The block graph data structure from Chapter 5 might also prove fruitful here, as we could use the concatenation of the textual representation of selected internal nodes as a dictionary.

Improving factorization There are a number of improvements to factorization techniques that can be explored in order to increase the efficiency our compression scheme. Kuruppu et al. [2011] find that compression can be significantly improved by using simple non-greedy factorization techniques such as those described by Horspool [1995]. A further improvement at the cost of run-time would be to compute the matching statistics of each document with respect to the dictionary and use it to determine its optimal factorization.

Deorowicz and Grabowski [2011b] identify that suffix scanning described in Section 3.1.4 was a significant run-time bottleneck during compression. They replaced the suffix array with a hash-based system and observed notable improvement in compression time. On the other hand, there are various methods to enhance a suffix array to improve scanning and other operations [Abouelhoda et al. 2004]. A simple method would be to pre-compute the start and end positions for each character range in the initial few characters of the each suffix in the suffix array. This could dramatically improve scanning speed as the initial scans will always incur the most cache misses. Decoding throughput can be improved by providing cache friendly factor selection. In general a factor might occur in many places throughout the dictionary. Our factorization method outlined in Section 3.1.4 selects the first instance of each factor, however, if presented with a number of positions it makes sense to select the index closest to the most recently encoded factor.

Another issue is the many choices for compression of position and length values, all with various space-time trade-offs. We observed the existence of higher-order patterns in the factor values throughout all experimental runs, which could be further exploited to improve both space and throughput such as using alternative integer codes, such as simple9 [Anh and Moffat 2005] or PForDelta [Zukowski et al. 2006] may substantially improve on a vbyte code and give relevant points on the trade-off curve. Finally, we note that our compression scheme can be easily adapted to perform compressed pattern matching [Manber 1997, de Moura et al. 2000]. The main drawback of this proposal is that during compression we require an index, such as a suffix array, to remain in memory in order to compress the pattern relative to a dictionary. However, we note that existing approaches to this problem use word-based semi-static modeling, as discussed in Chapter 1, and are not suitable for compression of larger text collections as their vocabularies size becomes a dominant cost of an encoding.

Block graphs A interesting avenue for research on block graphs would be to explore methods of construction in external memory. To achieve this we need an efficient method to determine leaf nodes during construction, specifically, an algorithm or data structure that is not constrained by physical memory. A disk-based doubling algorithm [Crauser and Ferragina 1999, Arge et al. 2002, Dementiev et al. 2008] would be suitable. Originally used for computing suffix arrays in external memory, it can easily be adapted to output and identify leaf nodes during each iteration.

In Section 5.4 we found that the dominant cost in our block graph implementation was storing leaf nodes. Furthermore, as we truncated the depth of a block graph this was offset by the cost of storing text blocks. Improving the compression of both node types at higher truncated depths could provide a significant improvement in compression.

Read compression The core idea behind Faust and Afin is to encode a read in terms of a previously observed similar read. Searching for candidate reads represents the most computationally expensive task during compression. We proposed a method that identifies candidate reads that share long common substrings and demonstrated that it is efficient in practice. However, there is a serious limitation to our selection method. We only select candidate reads that contain mutations at either end of a read. Consider two identical reads with a single mutation in the center of both reads. Our scheme will fail to identify that both reads are similar, as their longest common substring is only half the size of the read. This drawback essentially excludes a large group of possible candidates.

It is important to note that our initial experiments into read compression used edit distance as a metric for candidate selection, which avoided this issue altogether. However, we found that encoding the differences between reads, that is, representing the insertion, deletion and substitution operations became very costly when the number of mutations increased between reads, and did not result in efficient compression. A future area of research would be to explore different candidate selection techniques or use a number of techniques with separate encodings. For example, we could combine our existing approach and the edit distance heuristic, selecting the most compact representation for each block read. During an encoding a read would incur an additional flag bit to identify the method it was coded with, however, we expect this to be mitigated by the increased candidate selection pool.

Bibliography

M. I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

V. N. Anh and A. Moffat. Inverted index compression using word-aligned binary codes. *Information Retrieval*, 8(1):151–166, 2005.

V. N. Anh and A. Moffat. Index compression using 64-bit words. *Software Practice and Experience*, 40(2):131–147, 2010.

A. Apostolico. The myriad virtues of subword trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12, pages 85–96. Springer Berlin Heidelberg, 1985.

L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On sorting strings in external memory. In *Proceedings of the 29th ACM Symposium on Theory of Computing*, pages 540–548. ACM, 1997.

L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proceedings of the 10th European Symposia on Algorithms (ESA)*, pages 88–100. Springer, 2002.

D. Arroyuelo, G. Navarro, and K. Sadakane. Reducing the space requirement of LZ-index. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 318–329. Springer, 2006.

D. Arroyuelo, G. Navarro, and K. Sadakane. Stronger Lempel-Ziv based compressed text indexing. *Algorithmica*, 62(1–2), 2012.

R. A. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley Longman Publishing Co., Inc., 1999.

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

T. Bell. Better OPM/L text compression. *IEEE Transactions on Communications*, 34 (12):1176–1182, 1986.

T. Bell, I. H. Witten, and J. G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–591, 1989.

T. C. Bell, J. G. Cleary, and I. H. Witten. *Text compression*. Prentice-Hall, Inc., 1990.

D. Benoit, E. D. Demaine, J. I. Munro, R. Raman, V. Raman, and S. S. Rao. Representing trees of higher degree. *Algorithmica*, 43(4):275–292, 2005.

J. Bentley and D. McIlroy. Data compression with long repeated strings. *Journal of Information Sciences*, 135(1-2):1–11, 2001.

F. Berman. Got data?: A guide to data preservation in the information age. *Communications of the ACM*, 51(12):50–56, 2008.

P. Bille, G. M. Landau, R. Raman, K. Sadakane, S. R. Satti, and O. Weimann. Random access to grammar-compressed strings. In *Proceedings of the 22nd Symposium on Discrete Algorithms (SODA)*, pages 373–389, 2011.

J. K. Bonfield and M. V. Mahoney. Compression of FASTQ and SAM format sequencing data. *PLOS ONE*, 8(3):e59190, 2013.

R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of ACM*, 20(10):762–772, 1977.

N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. Lightweight natural language text compression. *Information Retrieval*, 10:1–33, January 2007a.

N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. Improving semistatic compression via pair-based coding. 4378:124–134, 2007b.

N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. New adaptive compressors for natural language text. *Software Practice and Experience*, 38:1429–1450, 2008.

N. R. Brisaboa, S. Ladra, and G. Navarro. k2-trees for compact web graph representation. In *Proceedings of the 19th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 18–30. Springer, 2009.

N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. Dynamic lightweight text compression. *ACM Transactions on Information Systems*, 28:10:1–10:32, July 2010.

M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

S. Büttcher and C. L. A. Clarke. Index compression is good, especially for random access. In *Proceedings of the 16th ACM Conference on Information and Knowledge Management (CIKM)*, pages 761–770. ACM, 2007.

S. Büttcher, C. L. A. Clarke, and G. V. Cormack. *Information Retrieval: Implementing and Evaluating Search Engines.* MIT Press, 2010.

A. Cannane and H. E. Williams. A compression scheme for large databases. In *Proceedings of the 11th Australasian Database Conference (ADC)*, page 6. IEEE, 2000.

R. Cánovas and A. Moffat. Practical compression for multi-alignment genomic files. In *Proceedings of the 36th Australasian Computer Science Conference*, pages 51–60, 2013.

F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems*, 26(2):1–26, 2008.

M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat. The smallest grammar problem. *IEEE Transactions on Information Theory*, 51(7): 2554–2576, 2005.

G. Chen, S. J. Puglisi, and W. F. Smyth. Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science*, 1(4):605–623, 2008.

X. Chen, M. Li, B. Ma, and J. Tromp. DNACompress: fast and effective DNA sequence compression. *Bioinformatics*, 18(12):1696–1698, 2002.

D. R. Clark. *Compact Pat Trees.* PhD thesis, University of Waterloo, Madison, 1996.

F. Claude and G. Navarro. A fast and compact web graph representation. In *Proceedings of the 17th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 118–129. Springer, 2007.

F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proceedings of the 15th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 176–187. Springer, 2009.

J. G. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

G. Cochrane, B. Alako, C. Amid, L. Bower, A. Cerdeño-Tárraga, I. Cleland, R. Gibson, N. Goodgame, M. Jang, S. Kay, et al. Facing growth in the European nucleotide archive. *Nucleic Acids Research*, 41(D1):D30–D35, 2013.

P. J. Cock, C. J. Fields, N. Goto, M. L. Heuer, and P. M. Rice. The sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic Acids Research*, 38(6):1767–1771, 2010.

G. V. Cormack and R. N. Horspool. Algorithms for adaptive huffman codes. *Information Processing Letters*, 18(3):159–165, 1984.

A. J. Cox, M. J. Bauer, T. Jakobi, and G. Rosone. Large-scale compression of genomic sequence databases with the Burrows–Wheeler transform. *Bioinformatics*, 28(11):1415–1419, 2012.

A. Crauser and P. Ferragina. On constructing suffix arrays in external memory. In *Proceedings of the 7th European Symposia on Algorithms (ESA)*, pages 224–235. Springer, 1999.

A. Crauser and P. Ferragina. A theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

B. Croft, D. Metzler, and T. Strohman. *Search Engines: Information Retrieval in Practice.* Addison-Wesley, 2010.

E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions Information Systems*, 18:113–139, 2000.

R. Dementiev, J. Kärkkäinen, J. Mehnert, and P. Sanders. Better external memory suffix array construction. *Journal of Experimental Algorithmics*, 12:1–24, 2008.

S. Deorowicz and S. Grabowski. Compression of DNA sequence reads in FASTQ format. *Bioinformatics*, 27(6):860–862, 2011a.

S. Deorowicz and S. Grabowski. Robust relative compression of genomes with random access. *Bioinformatics*, 27(21):2979–2986, 2011b.

S. Deorowicz, A. Danek, and S. Grabowski. Genome compression: a novel approach for large collections. *Bioinformatics*, 29(20):2572–2578, 2013.

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

M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997.

P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st Symposium on Foundations of Computer Science*, pages 390–398. IEEE, 2000.

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

P. Ferragina and G. Manzini. On compressing the textual web. In *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining (WSDM)*, pages 391–400. ACM, 2010.

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

P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In *Proceedings of the 11th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 150–160. Springer, 2004.

P. Ferragina, R. González, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics*, 13:12, 2009.

P. Ferragina, T. Gagie, and G. Manzini. Lightweight data indexing and compression in external memory. *Algorithmica*, 63(3):707–730, 2012.

J. Fischer. Inducing the LCP-array. In *Algorithms and Data Structures*, pages 374–385. Springer, 2011.

P. Flicek and E. Birney. Sense from sequence reads: methods for alignment and assembly. *Nature Methods*, 6:S6–S12, 2009.

E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, 1960.

M. H.-Y. Fritz, R. Leinonen, G. Cochrane, and E. Birney. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome Research*, 21(5): 734–740, 2011.

T. Gagie, P. Gawrychowski, and S. J. Puglisi. Faster approximate pattern matching in compressed repetitive texts. In *Proceedings of the 22nd International Symposium on Algorithms and Computation (ISAAC)*, pages 653–662, 2011.

T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. A faster grammar-based self-index. In *Proceedings of the 6th International Conference on Language and Automata Theory and Applications (LATA)*, volume 7183 of *LNCS*, pages 240–251, 2012.

T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, and S. J. Puglisi. LZ77-based self-indexing with faster pattern matching. In *Proceedings of the Latin American Symposium on Theoretical Informatics (LATIN)*, volume 8392 of *LNCS*, pages 731–742, 2014.

S. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, 12(3): 399–401, 1966.

G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In *Information retrieval*, pages 66–82. Prentice-Hall, Inc., 1992.

K. Goto and H. Bannai. Simpler and faster Lempel-Ziv factorization. In *Proceedings of the 23rd Data Compression Conference (DCC)*, pages 133–142. IEEE, 2013.

R. Grossi. Random access to high-order entropy compressed text. In A. Brodnik, A. López-Ortiz, V. Raman, and A. Viola, editors, *Space-Efficient Data Structures, Streams, and Algorithms*, pages 199–215. Springer-Verlag, 2013.

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

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

D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press, 1997.

F. Hach, I. Numanagić, C. Alkan, and S. C. Sahinalp. SCALCE: boosting sequence compression algorithms using locally consistent encoding. *Bioinformatics*, 28(23):3051–3057, 2012.

R. N. Horspool. The effect of non-greedy parsing in Ziv-Lempel compression methods. In *Proceedings of the 5th Data Compression Conference (DCC)*, pages 302–311. IEEE, 1995.

R. N. Horspool and G. V. Cormack. Constructing word-based text compression algorithms. In *Proceedings of the 2nd Data Compression Conference (DCC)*, pages 62–71, 1992.

T. J. Hudson, W. Anderson, A. Aretz, A. D. Barker, C. Bel l, R. R. Bernabé, M. Bhan, F. Calvo, I. Eerola, D. S. Gerhard, et al. International network of cancer genome projects. *Nature*, 464(7291):993–998, 2010.

D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, 1952.

G. Jacobson. Space-efficient static trees and graphs. In *Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 549–554. IEEE, 1989.

L. Janin, G. Rosone, and A. J. Cox. Adaptive reference–free compression of sequence quality scores. *Bioinformatics*, 2013. doi: 10.1093/bioinformatics/btt257.

D. C. Jones, W. L. Ruzzo, X. Peng, and M. G. Katze. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic Acids Research*, 40(22):e171–e171, 2012.

J. Kärkkäinen and S. J. Puglisi. Fixed block compression boosting in FM-indexes. In *Proceedings of the 18th Symposium on String Processing and Information Retrieval (SPIRE)*, volume 7024 of *LNCS*, pages 174–184, 2011.

J. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In *Automata, Languages and Programming*, pages 943–955. Springer, 2003.

J. Kärkkäinen, G. Manzini, and S. J. Puglisi. Permuted longest-common-prefix array. In *Combinatorial Pattern Matching*, volume 5577 of *LNCS*, pages 181–192. Springer Berlin Heidelberg, 2009.

J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Lightweight Lempel-Ziv parsing. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA)*, volume 7933 of *LNCS*, pages 139–150. Springer, 2013a.

J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Linear time Lempel-Ziv factorization: Simple, fast, small. In *Proceedings of the 24th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 189–200. Springer, 2013b.

J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Hybrid compression of bitvectors for the FM-index. In *Proceedings of the 24th Data Compression Conference (DCC)*, pages 302–311, 2014a.

J. Kärkkäinen, D. Kempa, and S. J. Puglisi. Lempel-Ziv parsing in external memory. In *Proceedings of the 24th Data Compression Conference (DCC)*, pages 153–162. IEEE, 2014b.

R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 181–192. Springer, 2006.

M. Kircher and J. Kelso. High-throughput DNA sequencing–concepts and limitations. *Bioessays*, 32(6):524–536, 2010.

D. E. Knuth. *The Art of Computer Programming, Pre-Fascile 1A. Draft of Section 7.1.3: Bitwise Tricks and Techniques.* 2007.

D. E. Knuth, J. James H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. *Journal of Discrete Algorithms*, 3(2):143–156, 2005.

C. Kozanitis, C. Saunders, S. Kruglyak, V. Bafna, and G. Varghese. Compressing genomic sequence fragments using SlimGene. *Journal of Computational Biology*, 18(3):401–413, 2011.

S. Kreft and G. Navarro. LZ77–like compression with fast random access. In *Proceedings of the 20th Data Compression Conference (DCC)*, pages 239–248. IEEE, 2010.

S. Kurtz. Reducing the space requirement of suffix trees. *Software Practice and Experience*, 29(13):1149–71, 1999.

S. Kuruppu, S. J. Puglisi, and J. Zobel. Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In *Proceedings of the 17th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 201–206. Springer, 2010.

S. Kuruppu, S. J. Puglisi, and J. Zobel. Optimized relative Lempel-Ziv compression of genomes. In *Proceedings of the 34th Australasian Computer Science Conference (ACSC)*, pages 91–98. Australian Computer Society, Inc., 2011.

B. Langmead, C. Trapnell, M. Pop, S. L. Salzberg, et al. Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biology*, 10(3):R25, 2009.

N. J. Larsson and A. Moffat. Offline dictionary-based compression. In *Proceedings of the 9th Data Compression Conference (DCC)*, pages 296–305. IEEE, 1999.

W. W. Lu and M. P. Gough. A fast-adaptive huffman coding algorithm. *IEEE Transactions on Communications*, 41(4):535–538, 1993.

V. Mäkinen and G. Navarro. Succinct suffix arrays based on run-length encoding. In *Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 45–56. Springer, 2005.

V. Mäkinen and G. Navarro. Implicit compression boosting with applications to self-indexing. In *Proceedings of the 14th International Conference on String Processing and Information Retrieval (SPIRE)*, pages 229–241. Springer, 2007.

V. Mäkinen, G. Navarro, J. Sirén, and N. Välimäki. Storage and retrieval of highly repetitive sequence collections. *Journal of Computational Biology*, 17(3):281–308, 2010.

U. Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Transactions on Information Systems*, 15(2):124–136, 1997.

U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

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

S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino. An extension of the Burrows Wheeler transform and applications to sequence comparison and data compression. In *Combinatorial Pattern Matching*, volume 3537 of *LNCS*, pages 178–189. Springer, 2005.

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

S. Maruyama, H. Sakamoto, and M. Takeda. An online algorithm for lightweight grammar-based compression. *Algorithms*, 5(2):214–235, 2012.

S. Maruyama, Y. Tabei, H. Sakamoto, and K. Sadakane. Fully-online grammar compression. In *Proceedings of the 20th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 218–229, 2013.

E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

A. Moffat. Word-based text compression. *Software Practice and Experience*, 19(2):185–198, 1989.

A. Moffat and A. Turpin. On the implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, 1997.

A. Moffat and A. Turpin. Efficient construction of minimum-redundancy codes for large alphabets. *IEEE Transactions on Information Theory*, 44(4):1650–1657, 1998.

A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer Academic Publishers, 2002.

A. Moffat, J. Zobel, and N. Sharman. Text compression for dynamic document databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(2):302–313, 1997.

A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3):256–294, 1998.

D. R. Morrison. PATRICIA-practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514–534, 1968.

J. I. Munro. Tables. In *Proceedings of the 16th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 37–42. Springer, 1996.

J. I. Munro and V. Raman. Succinct representation of balanced parentheses and static trees. *SIAM Journal on Computing*, 31(3):762–776, 2001.

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

G. Navarro. Wavelet trees for all. *Journal of Discrete Algorithms*, 25:2–20, 2014.

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

G. Navarro and E. Providel. Fast, small, simple rank/select on bitmaps. In *Experimental Algorithms*, pages 295–306. Springer, 2012.

C. G. Nevill-Manning, I. H. Witten, and D. L. Maulsby. Compression by induction of hierarchical grammars. In *Proceedings of the 4th Data Compression Conference (DCC)*, pages 244–253. IEEE, 1994.

G. Nigel and N. Martin. Range encoding: An algorithm for removing redundancy from a digitised message. In *Proceedings of the Institution of Electronic and Radio Engineers International Conference on Video and Data Recording*, pages 24–27, 1979.

G. Nong, S. Zhang, and W. H. Chan. Linear suffix array construction by almost pure induced-sorting. In *Proceedings of the 19th Data Compression Conference (DCC)*, pages 193–202. IEEE, 2009.

G. Nong, S. Zhang, and W. H. Chan. Two efficient algorithms for linear time suffix array construction. *IEEE Transactions on Computers*, 60(10):1471–1484, 2011.

I. Ochoa, H. Asnani, D. Bharadia, M. Chowdhury, T. Weissman, and G. Yona. QualComp: a new lossy compressor for quality scores based on rate distortion theory. *BMC Bioinformatics*, 14(1):187, 2013.

D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007a.

D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experimentation (ALENEX)*, 2007b.

S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):4, 2007.

R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proceedings of the 13th annual ACM Symposium on Discrete Algorithms (SIAM)*, pages 233–242, 2002.

R. Raman, V. Raman, and S. R. Satti. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Transactions on Algorithms*, 3(4), 2007.

R. Rice and J. Plaunt. Adaptive variable-length coding for efficient compression of spacecraft television data. *IEEE Transactions on Communication Technology*, 19(6):889–897, 1971.

J. Rissanen. Generalized kraft inequality and arithmetic coding. *IBM Journal of research and development*, 20(3):198–203, 1976.

J. Rissanen. Arithmetic codings as number representations. *Acta Polytechnica Scandinavica*, 31:44–51, 1979.

J. Rissanen and G. G. Langdon. Universal modeling and coding. *IEEE Transactions on Information Theory*, 27(1):12–23, 1981.

W. Rytter. Application of Lempel-Ziv factorization to the approximation of grammar-based compression. *Theoretical Computer Science*, 302(1–3):211–222, 2003.

K. Sadakane. Improving the speed of LZ77 compression by hashing and suffix sorting. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 83(12):2689–2698, 2000.

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

K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proceedings of the 17th annual ACM Symposium on Discrete Algorithms (SODA)*, pages 1230–1239. ACM, 2006.

D. Salomon. *Data Compression: The Complete Reference.* Springer, 2004.

F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In *Proceedings of the 25th ACM International Conference* *on Research and Development in Information Retrieval (SIGIR)*, pages 222–229. ACM, 2002.

J. J. Selva and X. Chen. Srcomp: Short read sequence compression using burstsort and elias omega coding. *PLoS ONE*, 8(12):e81414, 12 2013.

C. E. Shannon. The mathematical theory of communication. *Bell Systems Technical Journal*, 27(379–423):623–656, 1948.

J. T. Simpson and R. Durbin. Efficient construction of an assembly string graph using the FM-index. *Bioinformatics*, 26(12):i367–i373, 2010.

J. T. Simpson and R. Durbin. Efficient de-novo assembly of large genomes using compressed data structures. *Genome Research*, 22(3):549–556, 2012.

J. Sirén, N. Välimäki, V. Mäkinen, and G. Navarro. Run-length compressed indexes are superior for highly repetitive sequence collections. In *Proceedings of the 15th Symposium on String Processing and Information Retrieval (SPIRE)*, pages 164–175, 2008.

J. A. Storer and T. G. Szymanski. Data compression via textual substitution. *Journal of the ACM*, 29(4):928–951, 1982.

W. Tembe, J. Lowey, and E. Suh. G-SQZ: compact encoding of genomic sequence and quality data. *Bioinformatics*, 26(17):2192–2194, 2010.

A. Tombros and M. Sanderson. Advantages of query biased summaries in information retrieval. In *Proceedings of the 21st ACM Internationl Conference on Research and Development in Information Retrieval (SIGIR)*, pages 2–10. ACM, 1998.

A. Trotman. Compressing inverted files. *Information Retrieval*, 6(1):5–19, 2003.

Y. Tsegay, S. J. Puglisi, A. Turpin, and J. Zobel. Document compaction for efficient query biased snippet generation. In *Proceedings of the 31st European Conference on IR Research on Advances in Information Retrieval (ECIR)*, pages 509–520. Springer-Verlag, 2009.

A. Turpin, Y. Tsegay, D. Hawking, and H. E. Williams. Fast generation of result snippets in web search. In *Proceedings of the 30th ACM International Conference on Research and Development in Information Retrieval (SIGIR)*, pages 127–134. ACM, 2007.

E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

S. Vigna. Broadword implementation of rank/select queries. In *Experimental Algorithms*, pages 154–168. Springer, 2008.

R. Wan, V. N. Anh, and K. Asai. Transformations for the compression of FASTQ quality scores of next-generation sequencing data. *Bioinformatics*, 28(5):628–635, 2012.

P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory (SWAT)*, pages 1–11. IEEE, 1973.

T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

H. E. Williams and J. Zobel. Compressing integers for fast file access. *The Computer Journal*, 42(3):193–201, 1999.

I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes : Compressing and Indexing Documents and Images*. Morgan Kaufmann, 2nd edition, 1999.

V. Yanovsky. ReCoil - an algorithm for compression of extremely large datasets of DNA data. *Algorithms for Molecular Biology*, 6(1):1–9, 2011.

J. Zhang, X. Long, and T. Suel. Performance of compressed inverted list caching in search engines. In *Proceedings of the 17th International conference on World Wide Web (WWW)*, pages 387–396. ACM, 2008.

J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.

J. Ziv and N. Merhav. A measure of relative entropy between individual sequences with application to universal classification. *IEEE Transactions on Information Theory*, 39 (4):1270–1279, 1993.

N. Ziviani, E. S. de Moura, G. Navarro, and R. Baeza-Yates. Compression: A key for next–generation text retrieval systems. *Computer*, 33:37–44, 2000.

J. Zobel and A. Moffat. Adding compression to a full-text retrieval system. *Software Practice and Experience*, 25(8):891–903, 1995a.

J. Zobel and A. Moffat. Adding compression to a full-text retrieval system. *Software Practice and Experience*, 25(8):891–903, 1995b.

J. Zobel and A. Moffat. Inverted files for text search engines. *ACM computing surveys*, 38(2):6, 2006.

M. Zukowski, S. Héman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference of Data Engineering (ICDE)*, page 59. IEEE, 2006.