Efficient Data Representations for Information Retrieval

J. Shane Culpepper

Department of Computer Science and Software Engineering
University of Melbourne
VIC 3010 Australia

shanec@csse.unimelb.edu.au

Submitted in total fulfilment of the requirements of the degree of Doctor of Philosophy

December 2007

Produced on acid-free paper

Abstract

The key role compression plays in efficient information retrieval systems has been recognized for some time. However, applying a traditional compression algorithm to the contents of an information retrieval system is often not the best solution. For example, it is inefficient to perform search operations in maximally compressed data or to find the intersection of maximally compressed sets. In order to perform these operations, the data representation must be fully decompressed.

This thesis explores practical space versus time trade-offs which balance storage space against the competing requirement that operations be performed quickly. In particular, we are interested in variable length coding methods which are both practical and allow codeword boundaries to be found directly in the compressed representation. The latter property allows considerable flexibility in developing algorithms which can manipulate compact sets and sequences and allow selective decompression. Applications of such coding methods are plentiful. For instance, variations of this theme provide practical solutions to the compressed pattern matching problem. They are also a vital element in many of the compact dictionary representations recently proposed. In this work, we propose new data representations which allow key query operations to be performed directly on compressed data.

This thesis draws together previous work and shows the fundamental importance of coding methods in which control information is built directly into the representation. More particularly, this thesis (a) reviews current applications of compression in string searching algorithms; (b) critically evaluates existing coding approaches which allow fast codeword length identification; (c) introduces a new coding method which shares this property; (d) investigates applications of these coding approaches to searching and seeking in compact sequences; and (e) explores new data representations which provide a compromise between good compression and fast querying in compact sets.

Declaration

This is to certify that:

1. the thesis comprises only my original work towards a Ph.D. except where indicated in the Preface,
2. due acknowledgment has been made in the text to all other material used, and
3. the thesis is less than 100,000 words in length, exclusive of tables, maps, bibliographies, and appendices.

J. Shane Culpepper B.S. (University of Alabama, 1992)

For Miette.

Acknowledgements

A lot has changed for me in the three years I have spent at Melbourne University. So many people have helped and encouraged me on this journey that it will be difficult to thank them all. First, I wish to thank my advisor Alistair, for giving me this opportunity, and being patient as I worked out how to juggle personal setbacks and professional obligations. Alistair's keen intellect always found a way to challenge me, and his unwavering standard of excellence made sure I always knew what was expected.

I would also like to thank my thesis committee members Andrew Turpin and Tony Wirth for reading my annual progress reports and giving feedback on how to improve this thesis. Many thanks go to William Webber, Vo Ngoc Anh, and Mike Ciavarella for helping with system administration, coding problems, experimental datasets, proof reading, and everything else imaginable. I would also like to thank my fellow graduate students in Room 4.31a, who provided lots of interesting conversations and endured my constant interruptions: Rob Shelton, Yi Li, and Yuye Zhang.

It would be remiss of me to not also mention Linda Stern, Steven Bird, Paul Gruba, and Antonette Mendoza for repeatedly giving me casual teaching opportunities. I would also like to thank National ICT Australia (NICTA), in particular Peter Stuckey and Chris Leckie of the NIP program, for providing the financial support which allowed me to carry out this research. The University of Melbourne and NICTA provided a quality environment to do academic research with little administrative overhead.

A huge thank you goes out to my extraordinarily supportive family on the other side of the world. Mom, Dad, Scott, Jen, Kathleen, Carter, Jerry, Kim, and all my nieces and nephews: Thanks for everything. Finally, for *mon chéri* Miette, you are my inspiration in life.

J. Shane Culpepper
Melbourne, Australia,
December, 2007.

Preface

Publications arising from this thesis

Chapter 4 was presented in preliminary form at the 13th International Conference for String Processing and Information Retrieval (SPIRE 2005) [Culpepper and Moffat, 2005].

Chapter 5 was presented in preliminary form at the 14th International Conference for String Processing and Information Retrieval (SPIRE 2006) [Culpepper and Moffat, 2006].

Chapter 6 was presented in preliminary form at the 15th International Conference for String Processing and Information Retrieval (SPIRE 2007) [Culpepper and Moffat, 2007]. Additionally, part of Section 6.5 was presented in preliminary form at the 12th Australasian Document Computing Symposium (ADCS 2007) [Moffat and Culpepper, 2007].

Authorship

The `shuff` program is due to Andrew Turpin, University of Melbourne. The arithmetic coder implementation is due to Alistair Moffat and others, University of Melbourne. Data set pre-processing was done using the Zettair information retrieval engine developed at RMIT University, on raw data provided by the United States National Institute of Standards and Technology via the TREC program.

Document preparation

This document was prepared using TeX, LaTeX and BIBTeX. The packages: book, graphics, url, algorithm, natbib, setspace, amsmath, eucal, amsthm, mathtools, times, xspace, mathrsfs, subfigure, multirow, tabularx, enumerate, fancyheadings and fncychap were employed. Style information is a derivative of previous theses written by Mike Liddell and Andrew Turpin.

Contents

1 Introduction **1**

2 Text Compression **7**

2.1 Preliminaries 8

2.2 Information Theory 9

2.3 Compression Systems 13

2.4 Modeling 14

2.4.1 Character-Based Models 15

2.4.2 Dictionary-Based Models 18

2.4.3 Word-Based Models 20

2.4.4 Other Models 22

2.5 Coding 23

2.5.1 Minimum Redundancy Codes 23

2.5.2 Arithmetic Codes 26

2.5.3 Static Codes 27

2.6 Summary 31

3 Searching in Sets and Sequences **33**

3.1 On-line Searching in Text Collections 34

3.1.1 Exact Pattern Matching 34

3.1.2 Other Approaches to On-line Searching 37
3.1.3 Searching in Compressed Text 40
3.2 Off-line Searching in Text Collections 43
3.2.1 Ad Hoc Search Engines 44
3.2.2 Full-Text Indexes 49
3.3 Searching in Sets 54
3.4 Summary 61

4 Practical Coding **63**

4.1 Previous Work 64
4.1.1 Tagged Huffman Codes 64
4.1.2 Variable Byte Codes 65
4.1.3 Dense Byte Codes 66
4.1.4 (S,C)-Dense Codes 67
4.2 Restricted Prefix Byte Codes 69
4.2.1 Minimum Cost Codes 72
4.3 Prelude Considerations 74
4.3.1 Permutation Preludes 76
4.3.2 Bitvector Preludes 77
4.3.3 Difference Gap Preludes 77
4.3.4 Semi-Dense Preludes 78
4.4 Experiments 81
4.4.1 Experimental Setup 81
4.4.2 Efficiency and Effectiveness 83
4.4.3 Prelude and Block-Size Impact 83
4.5 Summary 89

5 Compact Sequences **91**

5.1 Previous Work 92
5.1.1 Byte-Pair Encoding 92

5.1.2 Tagged Huffman Codes 94
5.1.3 Stopper-Continuer Byte Codes 95
5.2 Restricted Prefix Byte Codes 96
5.2.1 Prefix Array Based Codes 99
5.3 Experiments 102
5.3.1 Experimental Setup 103
5.3.2 Seeking in Compressed Text 105
5.3.3 Integer Intermediate Pattern Matching 106
5.3.4 Byte-Aligned Pattern Matching 110
5.4 Summary 113

6 Compact Sets **115**

6.1 Set Operations For Inverted Lists 116
6.2 Set Representations 117
6.2.1 Array of Integers 117
6.2.2 Bitvectors 117
6.2.3 Compressed Representations 119
6.3 Set Intersection 120
6.3.1 Intersecting Multiple Sets 121
6.3.2 Efficient **F-SEARCH** 126
6.3.3 Duality of Searching and Coding 127
6.4 Practical Indexing 128
6.5 Experiments 130
6.5.1 Collection and Queries 131
6.5.2 Array-Based Intersection 133
6.5.3 Compressed Indexing 138
6.5.4 Time and Space Comparisons 140
6.5.5 Engineering a Hybrid Representation 141
6.6 Summary 147

7 Conclusions **149**

Bibliography **153**

List of Algorithms

1	Encoding a static byte code	29
2	Decoding a static byte code	29
3	Calculating the partitioning value in `scbc`	68
4	Decoding an `rpbc` block	71
5	Brute-force code calculation in `rpbc`.	73
6	Calculating a Semi-Dense Prelude	80
7	Seeking in Restricted Prefix Byte Codes.	98
8	Brute-force searching in Restricted Prefix Byte Codes	98
9	BMH searching in `rpbc`	100
10	Binary Set Intersection	120
11	Set versus Set Intersection (svs)	121
12	Baeza-Yates Intersection Algorithm (bya)	122
13	Adaptive Intersection Algorithm (adp)	123
14	Sequential Intersection Algorithm (seq)	124
15	Max Successor Intersection Algorithm (max)	125
16	Golomb Search Algorithm	127
17	Hybrid Intersection Algorithm 1 (hyb+m1)	143
18	Hybrid Intersection Algorithm 2 (hyb+m2)	144

List of Figures

1.1 The Google search engine 3

2.1 Unique decodability 11

2.2 The universal compression model 13

2.3 An example of the Burrows-Wheeler Transformation 17

2.4 Dictionary-based LZ77 compression 18

2.5 Dictionary-based LZ78 compression 19

2.6 Spaceless word modeling 20

2.7 Example of a Huffman tree 24

2.8 Arithmetic coding 26

2.9 Integer based coding 31

3.1 Modern search engine schematic 45

3.2 An inverted list 46

4.1 Decoding a restricted prefix byte code 69

4.2 Block-based semi-static coding 75

4.3 Semi-dense prelude construction 78

4.4 Semi-dense prelude costs as sub-alphabet size increases 84

4.5 Efficiency of restricted prefix byte codes for varying blocksize 87

4.6 Effectiveness versus efficiency for byte coding variants 88

5.1 Byte Pair Encoding 93
5.2 False match filtering in stopper-continuer codes 95
5.3 Brute-force searching in rpbc 97
5.4 Restricted Prefix Array Codes 101
5.5 Intermediate-based compressed pattern matching 102
5.6 Compressed pattern generation from an integer sequence 105
5.7 Seeking in Compressed Text 106
5.8 Integer based pattern matching with large alphabets 107
5.9 Integer-based pattern matching for large alphabets 108
5.10 Compressed searching in a spaceless-words parsed file 109
5.11 Compressed searching in a RE-PAIR based file 111
5.12 Byte encoded pattern matching for large alphabets 112

6.1 Two level auxiliary indexing 129
6.2 Index terms versus pointers 132
6.3 Effects of **F-SEARCH** on intersection 134
6.4 Effects of **F-SEARCH** on intersection 135
6.5 Match depth comparisons in adaptive intersection 136
6.6 Intersection in compact set representations 139
6.7 Tradeoffs in index cost and query throughput 140
6.8 Total data used versus partitioning for hyb. 142
6.9 Comparison of compressed index for theGOV2 dataset using hyb 145
6.10 Efficiency of the hyb+m1 algorithm for the GOV2 dataset. 146
6.11 Efficiency of the hyb+m2 algorithm for the GOV2 dataset. 146
6.12 Space versus CPU for hybrid bitvector representations 147

List of Tables

2.1 Universal integer codes 30

4.1 Byte-aligned codeword comparison 72

4.2 Statistical properties of the benchmark collection 81

4.3 Compression effectiveness in byte code variants 82

4.4 Prelude costs for byte code representations 84

4.5 Prelude impact on decoding efficiency 85

4.6 Effectiveness impact of differing block sizes 86

5.1 Statistical properties of search test files 103

6.1 Query Statistics 131

6.2 Average **F-SEARCH** calls based on query length 137

6.3 Space cost comparisons for compact set representations 138

6.4 Total space costs for bitvector and byte code hybrids 141

Chapter 1

Introduction

Computers and mobile devices are indispensable in our daily lives. They allow us to organize our busy schedules, communicate instantaneously with people on the other side of the world, and find information. The level of access and control over information we have today was unimaginable two decades ago. But managing all of this information, and making it readily available, is neither simple nor transparent.

Thankfully, it is possible to make use of internet services like Google, Yahoo, or MSN without understanding the mechanisms which make them work. The services are intuitive and serve a utilitarian purpose. Their ease of use and ubiquity is one of the most remarkable success stories of modern times. However, “ease of use” should not be confused with simplicity. The massive quantity of information which must be manipulated every day by these services provides fascinating challenges for the scientists and engineers who design, build, and maintain these complex systems.

For many, the underpinnings of a service like the Google search engine are shrouded in mystery. But complex software systems such as these can conceptually be decomposed into two separate but cooperating components. The first is a *data collection*, which stores information in a pre-defined format. The other component is a suite of *algorithms*, which process the data. For example, searching for a phrase using Google is a combination of a “search” algorithm and a data source (webpages from the World Wide Web), working in tandem.

In this thesis, we explore the inherent relationship between the representation of data, and the algorithms which manipulate it. In particular, we are interested in how to choose compact representations of data which still allow efficient manipulation. In other words, we wish to minimize space usage and maximize algorithm performance. By carefully considering the operations which must be performed on a data source in a given context, it is possible to propose *practical* solutions. These practical solutions consider both the information content and the underlying computer hardware architecture.

In general, finding the perfect balance between time and space efficiency is difficult. The extraordinary evolution of computer hardware is one of the greatest contributing factors. Over the past two decades, processing speed and disk storage capacity has doubled every 24 months, for a total gain of more than a factor of 5,000. But the cost of accessing data from secondary storage devices has only grown linearly [Hennessy and Patterson, 2006]. The yearly increase in processing speed compared to load and store costs in the memory hierarchy means that many of the underlying principles of algorithm design are now being reconsidered. The evolution of practical algorithm design is expected to become even more dramatic as the "manycore" model of hardware production makes massive parallelism the de facto standard.

The performance gap between memory and processor continues to widen, and the trend is likely to continue for some time. Modern processors take around 200 clock cycles to access Random Access Memory (RAM) while "expensive" operations such as floating point multiplication take about 4 clock cycles [Asanovic et al., 2006]. Loading memory segments into RAM from secondary storage now has costs which are on the order of tens of thousands of clock cycles. Add the massive growth of data available on the internet to the equation and the challenges become even more apparent. These practical considerations serve as motivation for a vibrant research community where developing algorithms which minimize space complexity is as important as minimizing time complexity.

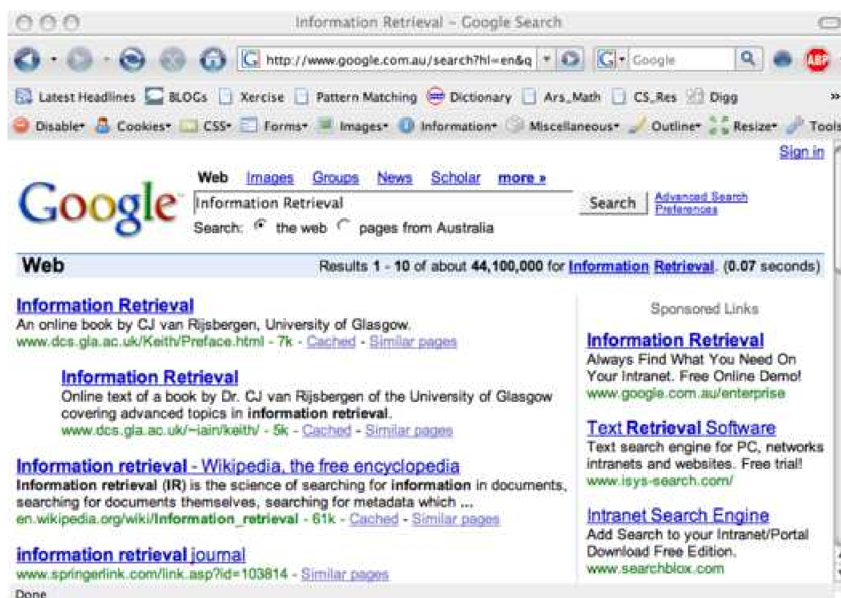


Figure 1.1: A search for "`information retrieval`" on the Google search engine returns 44.1 million ranked results in about 70 msec. Retrieved May 11, 2007.

Information Retrieval

Information Retrieval (IR) is the multidisciplinary science of data search. Information Retrieval is now a mature field in computer science, and its paradigms are actively applied in diverse areas such as search engines, file systems, digital libraries, and video surveillance. The data repositories can store anything from text to digital video, and the collection size is often immense. These massive collections pose significant challenges for the algorithms which manipulate the data.

A prime example of a massive dataset is the World Wide Web. There is considerable debate over exactly how much information is available on the World Wide Web. The dynamic nature of the web makes it difficult to determine the exact size. However, we can make a rough estimate. A recent study asserts that there were at least 11.5 billion documents accessible to major internet search engine services, as of January 2005 [Gulli and Signorini, 2005]. If each page is around 20 kB, then there is approximately 215 TB of text available. This does not include video and VoIP data, which is arguably

responsible for the majority of internet bandwidth used today.

Search engines such as Google, Yahoo, or MSN are capable of processing thousands of queries per second, and each individual query can involve hundreds of megabytes of data [Barroso et al., 2003]. In Figure 1.1 we see an example of remarkable efficiency despite massive data volume. A search for "`information retrieval`" returns around 44 million results in 0.07 seconds. Finding algorithms and data structures which provide disciplined compression, and still support a subset of key operations is tantamount to success in such systems. So, the tension between space and time efficiency can readily be seen in the web services we use everyday, making them a suitable framework for this research.

Data Compression

Since Shannon [1948] established the fundamental relationship between *information content* and *redundancy*, researchers have looked for techniques to store information in a reduced format. The idea of removing redundancy, while maintaining information content, is at the core of all data compression techniques. The reduction does not come without a cost. Traditionally, the cost is CPU time; but as CPU speeds continue to increase much faster than access times on primary and secondary storage devices, compression becomes a more and more important enabling technology.

Two standard metrics often employed in data compression are *efficiency* and *effectiveness*. Efficiency is the measure of a resource requirement. Generally, it is the speed or throughput of an algorithm. It can be measured in CPU time (sec), symbols per second (sym/sec), or another similar hybrid measure. Effectiveness is the amount of redundancy removed. It is commonly expressed as a compression ratio (%), or in bits per symbol (bps). It should come as no surprise that efficiency and effectiveness are generally in tension, as it is a classic case of time versus space complexity.

This leads to our primary research question: What trade-offs are possible between efficiency and effectiveness? Is it possible to trade a small amount of effectiveness for a large boost in efficiency? These are complex questions, and often depend on the

application domain, as much as the relative weighting of the metrics. As was mentioned earlier, this balancing act is dynamic, as the underlying hardware constraints continue to evolve. The balancing act also suggests there is no single "best" universal compression algorithm. Instead, it reinforces the importance of targeting the modeling and coding to the task and computing environment at hand.

Compact Data Structures

New approaches to effectively store information must not only support efficient query and modification, but also use the minimum amount of space possible. The key premise of *compact* or *succinct* data structures is based on one key observation. If we are willing to allow a small amount of control information back into our compressed representation, it is possible to (sometimes dramatically) enhance functionality. A deeper understanding of data compression enables us to minimize the redundancy in the data, and perform complex operations directly on the compact representation. This thesis evaluates the renewed interest in compression and and its role in compact data representations which support efficient operations in both sets and sequences of data. We also identify and utilize interesting properties of novel coding methods.

Overview

This thesis draws together previous work and shows the fundamental importance of coding methods in which control information is built directly into the representation. Chapter 2 reviews the fundamentals of text compression. Chapter 3 reviews the role of searching in sequences and sets. In Chapter 3, we also consider the key role of compression in designing efficient search algorithms for various set and sequence representations. Chapter 4 investigates the fundamental importance of byte-aligned coding, and explores new prelude representations for compression algorithms which must manipulate large alphabets. In Chapter 5 we investigate the value of codeword boundary discovery for byte-aligned, variable length codes, as well as applications of searching and seeking directly in compressed text. Next, we look at compact set representations for fast query evaluation in Chapter 6. In particular, we focus the

problem of compact set intersection which is pivotal in information retrieval systems today. Finally, Chapter 7 reviews the lessons learned and discuss possible extensions of the current work.

Contributions of the Thesis

This thesis provides an extensive review of state-of-the-art techniques for efficient representations of sets and sequences, and comprehensive empirical studies for many of the key approaches discussed in the literature. In addition, this thesis makes several new contributions which are efficient and effective in practice. The contributions of this thesis include: a new approach to integer based byte coding (Section 4.2), two new approaches to efficient prelude representations (Section 4.3.3 and Section 4.3.4), new approaches to searching directly in integer intermediate representations (Section 5.3.3) and byte encoded representations (Section 5.2), a new approach to set intersection for ordered lists of integers (Section 6.3.1), a new succinct data structure for efficient set intersection (Section 6.4), and new hybrid approaches for faster set intersection using minimal space (Section 6.5.5). Together, these contributions highlight the extraordinary flexibility of byte-aligned coding for a variety of important problems in the text retrieval domain.

Chapter 2

Text Compression

Data compression is a key enabling technology in computer science, and is typically used in two fundamental ways. Firstly, compression greatly enhances our ability to store and transmit information efficiently. Compression for storage and transmission is a mature area of research at this point, and there are many good books on the topic (See for instance Storer [1988], Bell et al. [1990], Nelson and Gailly [1995], Witten et al. [1999], Moffat and Turpin [2002], Hankerson et al. [2003], Sayood [2006], Salomon [2007]). Secondly, compression has recently played an increasingly important role in another way. The explosion of research on *succinct* data structures is closely tied to the success of compression in storage technologies. These data representations allow more information to be kept in primary memory, facilitating faster processing. The role of compression in algorithmic research is discussed in more detail in Chapter 3.

Data compression has a critical role in the efficient transmission of information. Consumer devices such as cell phones and portable computing devices are now commonplace, allowing unconstrained connectivity. But, internet bandwidth usage is typically limited and tolled. Compression reduces the amount of information transferred, saving costs and time. There are a multitude of data types where compression techniques are applicable in such a scenario, and various trade-offs are possible. Audio, video, graphical, and textual information can all benefit from data compression.

If some information loss is permissible, *lossy* compression methods may be used. Lossy compression methods do not allow a perfect reconstruction of the original data, but often lead to large reductions in storage costs. These methods are particularly important in streaming media and internet telephony applications. In fact, many consumer devices such as digital cameras and portable music players would not be possible without lossy compression standards such as JPEG and MP3. However, lossy methods are not attractive in applications such as text compression where the original data must be reconstructed, with no information loss. Compression algorithms which are fully reversible are referred to as *lossless* methods.

This thesis focuses lossless compression in massive text collections. In Section 2.1 and Section 2.2 we review the fundamentals of information theory, and provide basic definitions which will be used throughout the thesis. In Section 2.3, Section 2.4, and Section 2.5 we will review compression systems, and explore the relationship between modeling and coding in modern text compression algorithms.

2.1 Preliminaries

For clarity, we now introduce the basic terminology which is used throughout this thesis. A *set* is a collection of distinguishable objects, often called *members* or *elements*. Sets can be finite or infinite, and do not impose any ordering on the members. Strictly speaking, each element may only appear in the set once. If elements may appear multiple times, the resulting collection is referred to as a *bag* or *multiset*.

In contrast, a *sequence* $\mathcal{X} = x_0 x_1 \ldots x_{n-1}$ is a collection of elements in which the order of the elements must be maintained, and elements can occur any number of times. Alternately, a sequence is simply a multiset in which the order is important. The sequence *length* or *size*, often denoted by n, is the total number of elements in the sequence; with x_i denoting the i th element. A sequence can be thought of as a list of elements drawn from a previously defined set S. If the member set is non-empty and finite, we call the set the *universe* $\mathcal{U}$ of size u, or an *alphabet* Σ of size σ, depending on the context. Any finite sequence drawn from an alphabet is called a *string* or *word*. To

avoid ambiguity, a finite sequence will always be referred to as a *string* or *text* in this thesis and a *word* will refer to a member of a *lexicon*, such as English. Furthermore, we denote the set of all strings over an alphabet Σ as Σ^*.

Let $\mathcal{T} = t_0 t_1 \ldots t_{n-1}$ represent a *text*, of length n, drawn from an alphabet Σ. A text $\mathcal{T}$ of length $n = 0$ is called the *empty text*, and denoted ε. A string $\mathcal{F}$ is called a *factor* of a text $\mathcal{T}$ if there exist strings x, y, such that $\mathcal{T} = x\mathcal{F}y$. If $x = \varepsilon$, then $\mathcal{F}$ is a *prefix* of $\mathcal{T}$. If $y = \varepsilon$, then $\mathcal{F}$ is a *suffix* of $\mathcal{T}$. The string $\mathcal{F}$ is a *subsequence* of $\mathcal{T}$ if there exists the strings $f_0 f_1 \ldots f_{m-1}$ and $t_0 t_1 \ldots t_{n-1}$, such that $\mathcal{F} = f_0 f_1 f_{m-1}$ and the text $\mathcal{T} = t_0 f_0 t_1 f_1 \ldots f_{m-1} t_{n-1}$. Thus, $\mathcal{F}$ is derived from $\mathcal{T}$ by erasing factors in $\mathcal{T}$. A factor or subsequence $\mathcal{F}$ of a string is *proper* if $\mathcal{F} \neq \mathcal{T}$.

2.2 Information Theory

The primary goal of information theory is to quantify information and the fundamental laws which govern it. This is typically accomplished by understanding the nature of the information represented by a message, and devising a representation in which all redundancies are removed. The unit of measure for information can, in principle, use any radix. The most common unit of measure for information are bits. Unless otherwise noted, we limit our discussion to binary for clarity.

The process of converting information (either discrete or analog) into a digital sequence is called *coding*. The reverse process is commonly referred to as *decoding*. Coding takes a source alphabet, $\mathcal{S} = \{s_1, \ldots, s_u\}$, and a channel alphabet, Σ, and generates a list of *productions*:

$$\begin{array}{ccc} s_1 & \mapsto & c_1 \\ & \vdots & \\ s_u & \mapsto & c_u, \end{array}$$

where $c_1 \ldots c_u \in \Sigma^*$. The simplest approach is to create a mapping $\phi : \mathcal{S} \mapsto \Sigma^*$ from the given channel alphabet to a string s_i, where each string in Σ^* is a fixed length ℓ

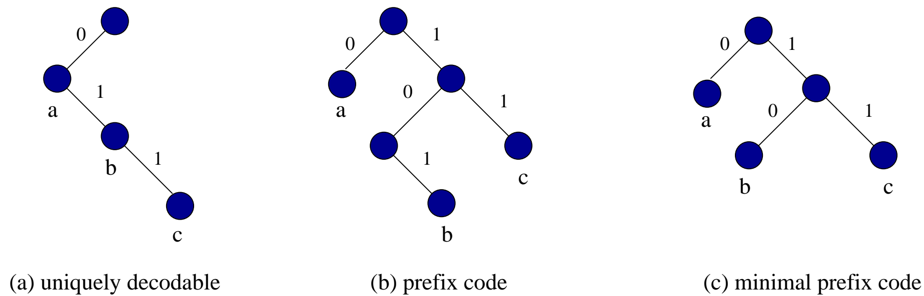
of binary digits (the channel alphabet in this case), called a *codeword*. Here, ϕ is a concatenation of the codewords. For example, given $\mathcal{S} = \{a, c, g, t\}$, $\Sigma = \{0, 1\}$, and the encoding scheme,

$$
\begin{aligned}
a &\mapsto 00, \\
c &\mapsto 01, \\
g &\mapsto 10, \\
t &\mapsto 11,
\end{aligned}
$$

then $\phi(gattaca) = 10001111000100$. In this instance, all of the codes s_i appearing in the encoding scheme are *fixed-length*, with a common length $\ell = 2$ bits, but *variable length* codes are also possible and even desirable in certain circumstances.

A *variable length code* also maps each member of a source alphabet $\mathcal{S}$ to a binary string s_i. But, the *length* $\ell(s_i)$ is no longer constant. The motivation for using a variable length code is based on the notion that the frequency of appearance of each member of the source alphabet may not be uniformly distributed. For instance, consider the letters ‘e’ and ‘z’ in natural language text. The letter ‘e’ appears more often than ‘z’ in English, so it can often be beneficial to use a short codeword to represent ‘e’ and a long codeword to represent ‘z’. In fact, this intuition is the basis of Morse code where more common letters are represented using shorter strings and less common letters receive longer strings [Bell et al., 1990].

A desirable property of variable length codes is *unique decodability*. In order for the mapping to be reversible, each codeword must be distinguishable from every other codeword. So, the coding function ϕ should be injective. Figure 2.1 depicts various encodings which ensure distinct codewords are assigned for each symbol. Figure 2.1 (a) shows codewords which are distinct, but not *instantaneously decodable*. A valid parsing of an encoded string using these codewords cannot be reliably decoded from left-to-right without buffering. For example, consider the encoding $\phi(ccc) = 011011011$. If we begin decoding from left-to-right, we cannot disambiguate the code



(a) uniquely decodable (b) prefix code (c) minimal prefix code

Figure 2.1: Principal coding methods used which allow variable length codes to be uniquely decoded.

until the length of the decoding window is equal to the length of the longest assigned codeword. However, if we enforce a prefix-free property, any encoded sequence is instantaneously decodable. A code is considered prefix-free if no codeword is a prefix of any other codeword. Figure 2.1 (b) and (c) show examples of prefix-free codes. Notice that Figure 2.1 (a) is uniquely decodable but is not prefix-free. Both (b) and (c) are prefix-free and (c) is also *minimal* or *full*. A minimal prefix code assures that the shortest code possible is used by each symbol by ensuring that each node in the tree is either a codeword, or a proper prefix of a codeword. Alternative approaches to devising uniquely decodable mappings are investigated by Gilbert and Moore [1959] and Fraenkel and Klein [1993].

There are several possible approaches to generate variable length codes. The lower bound on codeword lengths for uniquely decipherable codes was investigated in detail by Kraft [1949]. In this work, Kraft demonstrated that every uniquely decipherable code in a given source alphabet $\mathcal{S} = s_1, \ldots, s_\sigma$, with corresponding codeword lengths $\ell(s_i)$, where $1 \leq i \leq \sigma$, satisfies

$$\sum_{i=1}^{\sigma} 2^{-\ell(s_i)} \leq 1. \tag{2.1}$$

McMillan [1956] later extended these results and showed that any code satisfying Equation 2.1 can be transformed into an equivalent prefix-free code.

The ability to generate sequences of variable length codes which are decodable is only part of the issue at hand. We must also consider which members of the source alphabet are the most probable, as these elements are the ones which should receive the shortest codes. In seminal work, Shannon [1948] devised an elegant solution which addresses this shortcoming. Shannon pointed out that the average codeword length ℓ_{avg} is bounded below by the zeroth order self-information, $\mathcal{H}$, of a discrete source, where

$$\mathcal{H} = -\sum_{i=1}^{n} p_i \log_2 p_i. \tag{2.2}$$

The term, $-\log(p_i)$, is often referred to as the *information content* of symbol s_i. The *redundancy* of the code is $\mathcal{R} = \ell_{avg} - \mathcal{H}$, and cannot be negative.

The bounds established by Shannon allow us to easily estimate how compactly a sequence can be represented if the source is generated as a sequence of Bernoulli trials over a fixed probability distribution. It is also of interest to know how compactly a subset of items can be represented, for problems which require efficient set-based operations, such as dictionary data structures. The problem can be reduced to the combinatorial cost in bits to unambiguously store an n-subset from a universe of u possibilities. Given that there are $\binom{u}{n}$ possible ways of extracting n items from a universe of u possibilities, and the cost of storing each item is $\log(n)$ bits, the following bound can be determined:

$$\left\lceil \log \binom{u}{n} \right\rceil = \log \frac{u!}{(u-n)!n!} \approx n \left(\log \frac{u}{n} + 1.44 \right), \tag{2.3}$$

when $n \ll u$ applies, and using Stirling's Approximation, $n! \approx \sqrt{2\pi n} \cdot n^n / e^n$, for the derivation. Brodnik and Munro [1999] were the first to derive this result, and it serves as an information-theoretic lower bound on the number of bits necessary to store a set [Sadakane and Grossi, 2006]. Equation 2.3 represents the optimal space in the worst case, over all possible sets of size n over a universe of size u, and also provides a bound for efficient set representations, which are explored in Chapter 6.

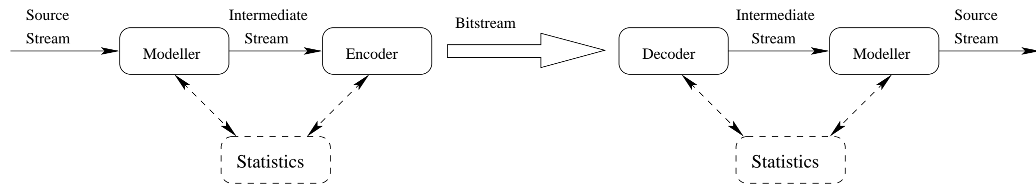


Figure 2.2: The fundamental operations of a compression system.

2.3 Compression Systems

A great deal of effort has gone into designing compression systems for every type of input data imaginable, and they contain many complex details. A compression system is a program which uses one or more algorithms to reduce the cost of storing a message, in a pre-defined format. Since there are often several cooperating algorithms in the compression system, it is often valuable to decompose the system into logical sub-components. Figure 2.2 depicts the fundamental operations of a universal compression system. Rissanen and Langdon [1981] propose a two component decomposition, consisting of modeling and coding, while Moffat et al. [1998] propose an optional third statistical module. The *model* is responsible for learning and applying domain knowledge to a given data stream. The *statistics module* gathers information about the probability distribution of the data stream, and *coding* assigns identifiers for symbols in the data stream, using statistical information collected. In the three component model, the statistics module is responsible for estimating symbol probabilities, and in the implementation can be coupled with the modeler, or the coder, or with both.

Many systems do not differentiate between modeling and coding, often making empirical comparisons difficult. With care, the two can be separated cleanly which allows more detailed comparisons to be made. For systems that clearly delineate modeling and coding, the intermediate output can usually be thought of as a stream of integers. This offers a great deal of flexibility in choosing an appropriate coding algorithm. We can either use a *static code*, such as unary, or take a more disciplined approach and generate a *minimum redundancy code*.

If the statistics module is coupled with the encoder, it is used to generate estimates

of the source alphabet and assigns appropriate *channel codes*. These estimations can be generated in a number of different ways: *static*, *semi-static*, or *adaptive*. Semi-static modeling is an *off-line* technique because two passes over the source are required. The first pass gathers statistics of the source, and then a second pass is made to encode the source. Semi-static modeling has the disadvantage that the source alphabet and its probability ordering must be transmitted as well as the message, which can be costly in some circumstances. This is generally accomplished by encoding the necessary source alphabet information in a *prelude* or *header*. It is possible to mitigate the costs of making two passes in semi-static codes by using some form of block buffering. Modern network and hardware protocols often allow efficient buffering for moderately sized blocks. Block based semi-static coding is explored in detail in Chapter 4.

Static and adaptive models are *on-line* since only one pass is required over the message source. A static model uses a fixed probability distribution which is predetermined and known by both the encoder and decoder, while adaptive models are computed on the fly. Static models are used when computational resources are low, throughput is of paramount importance, or on short messages. Fully adaptive models have been studied extensively and are generally used in network based transactions, where decompression costs are not favored over compression costs [Sayood, 2006, Salomon, 2007]. The application domain often dictates the type of statistical model used.

2.4 Modeling

Modeling is a robust and complex field of study in its own right. The appropriate model depends on the type of data which must be compressed. For instance, the best model for eliminating redundancy in video streams almost certainly isn't the best model for files containing English text. In this section, we discuss the most popular models for processing English text. Compression modeling is a heavily researched area and several good methods have been developed over the years. For a more detailed treatment see, for example, Sayood [2006], or Salomon [2007].

2.4.1 Character-Based Models

The simplest text model is a *0-order* model, where all symbols are considered as mutually independent. In this model, the characters are simply ranked by frequency. The *0-order* empirical entropy, $\mathcal{H}_0$, for a text $\mathcal{T}$ containing n symbols, is $\mathcal{H}_0 = (1/n)\sum_i^\sigma f_i \log(f_i/n)$, where f_i is the number of occurrences of the i th element in the source alphabet, σ is the size of the alphabet, and n is the length of $\mathcal{T}$.

Higher order models are also possible. There are several possible approaches to defining higher order models. The simplest approach is to use a Markov model, where the probability of a symbol is dependent on the previous k symbols. Other approaches include finite state models, linear models, or even hidden Markov models. Here we focus only on a standard Markov modeling approach. Regardless of the approach used, compression can typically be improved if symbols are considered to be conditionally dependent on previously seen symbols. A *k-order* model depends on the frequency of the symbol considering all the occurrences of symbols preceded by those k characters, where k is the *context* of the symbol. If we assume a Markov source, the *kth-order empirical entropy*, $\mathcal{H}_k$ of a text $\mathcal{T}$, of length n, over an alphabet Σ of size σ, represents the expected uncertainty of a randomly chosen symbol for a particular context k [Ferragina and Manzini, 2000]. For any length-k word $w \in \Sigma^k$, let $w_\mathcal{T}$ denote the string of characters following w in $\mathcal{T}$. So, the expected self information is

$$\mathcal{H}_k = \begin{cases} \dfrac{1}{n}\displaystyle\sum_{i=1}^{\sigma} f_i \log \frac{n}{f_i} & \text{if } k = 0, \\ \dfrac{1}{|\mathcal{T}|}\displaystyle\sum_{w\in\Sigma^k} |w_\mathcal{T}| \cdot \mathcal{H}_0(w_\mathcal{T}) & \text{if } k \geq 1. \end{cases} \tag{2.4}$$

In the $k = 0$ case, f_i is simply the frequency of the i th symbol in the alphabet. When $k \geq 1$, $w_\mathcal{T}$ represents the concatenation of the symbols immediately following occurrences of a string w in $\mathcal{T}$, and $|w_\mathcal{T}|$ is the frequency of w in $\mathcal{T}$. It should also be noted that $\mathcal{H}_k \leq \log\sigma$ holds for all k. Higher order entropy models have received a great deal of attention in recent years because of its role in bounding various succinct

data structures (see for example [Ferragina and Manzini, 2000, 2005, Sadakane and Grossi, 2006, Navarro and Mäkinen, 2007]) and because of the fact that PPM and BWT compression schemes often exhibit compression effectiveness based on these models.

One of the most successful approaches to higher order models was initially presented by Cleary and Witten [1984] and was referred to as Prediction by Partial Matching (PPM). Finite context models such as PPM offer state-of-the-art compression effectiveness at the cost of decoding efficiency. Several improvements to the basic model have been proposed primarily to resolve key issues related to assigning probabilities to previously unseen symbols [Moffat, 1990, Howard and Vitter, 1992, Bunton, 1997]. The *zero-frequency problem* crops up in may different guises in natural language processing. A detailed analysis of probability estimation was performed by Åberg et al. [1997]. Several experimental studies of various PPM methods have also been carried out [Bunton, 1996, Tehan, 1998, Åberg, 1999]. Maintaining the higher order contexts in PPM has traditionally been expensive in terms of both memory and processing power. However, recent work has dramatically reduced the resource requirements, but are still less efficient than LZ and BWT-based methods [Effros, 2000, Shkarin, 2002].

Finally, we mention the latest popular addition to compression modeling, the Burrows-Wheeler Transformation (BWT). The key idea proposed by Burrows and Wheeler [1994] is to use a reversible sorted-context transformation, which is extremely effective and flexible in practice. Then, a move-to-front (MTF) transformation of Bentley et al. [1986] is applied to exploit the locality are produced by the transformation, followed by a zero-order coder.

Figure 2.3 provides an example of the BWT algorithm for the word "`ALABAMA`". In the first step, a total of n permutations of the message of length n characters is produced by cyclically rotating the characters. The permutations are then sorted lexicographically by the characters preceding the last character of each string. The string in (c) can then be transmitted, after MTF and entropy coding, along with an identifier uniquely marking the first character in the message. Decoding is accomplished by reconstructing the

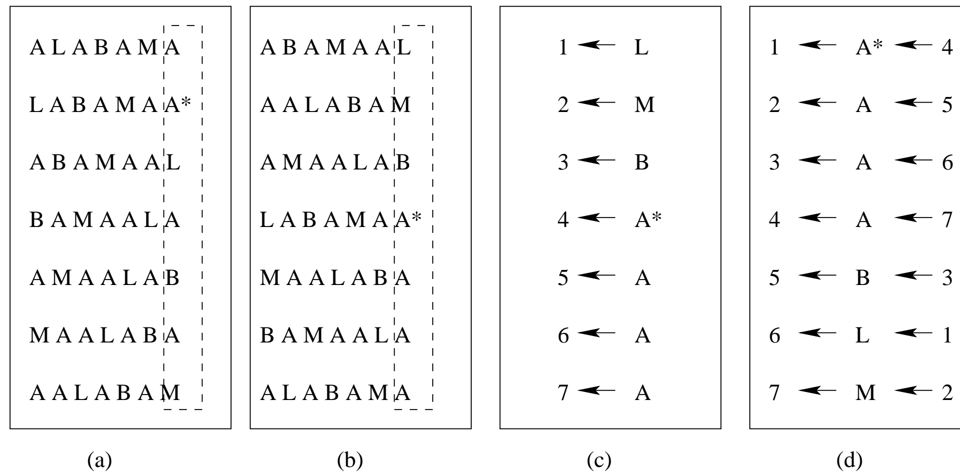


Figure 2.3: Example of the BWT algorithm for the word "ALABAMA". (a) The n permutations of a string of n characters is generated by cyclically rotating the characters. (b) The resulting strings are then sorted lexicographically by the all characters preceding the last character of each string. (c) The resulting string can then be transmitted. (d) Recovering the original string is done by assigning a rank identifier of the transmitted string, and an additional rank identifier produced from a stable sorting of the transmitted string.

transmitted string and assigning rank identifiers for each symbol. Then, the string is sorted (using a stable sort), and a second set of rank identifiers are assigned based on the new ordering. The original message is then produced by following the corresponding rank identifier, and position offsets. The first character "A" has a rank identifier of "1" (on the left). The corresponding "1" is then located in the right identifier array, to produce the character "L". The "6" corresponds to an "A" in the third position, and so on. Once all of the characters are cycled through, the original message is retrieved.

The BWT is a practical solution to text compression, and several implementations, such as BZIP2 and SZIP are readily available [Schindler, 1997, Seward, 2000, 2001]. A great deal of research has been initiated on improving various aspects of this novel transformation [Chapin, 2000, Wirth, 2000, Deorowicz, 2000, Wirth and Moffat, 2001, Deorowicz, 2002]. The BWT method is also a fundamental component of an emerging class of fascinating class of compressed, full-text indexing methods, which are discussed in Section 3.2.2. There is also a surprising duality between the BWT and

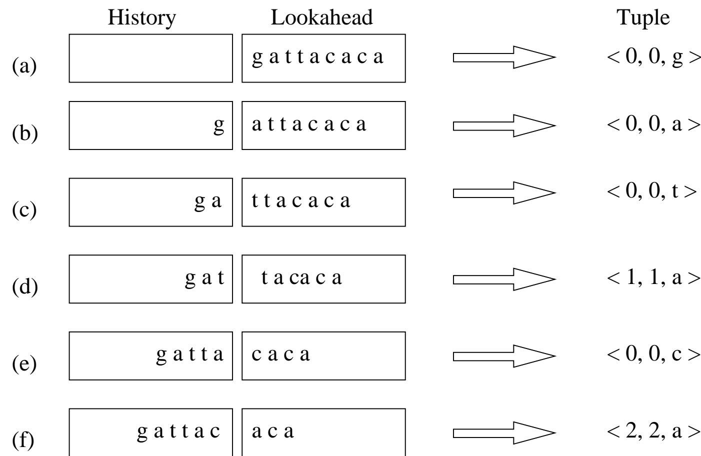


Figure 2.4: Dictionary-based compression of a simple DNA string, using an LZ77 model. Each row illustrates a string being encoded, with the corresponding 3-tuple. Two fixed-length buffers are maintained, a history buffer, and a lookahead buffer. Once the history buffer is filled, the fixed *window* will begin “sliding” across the text.

PPM modeling approaches, which have been recently explored by Cleary and Teahan [1997] and Moffat and Turpin [2002]. A complete discussion of BWT is out of scope in this thesis, but fascinating nonetheless.

2.4.2 Dictionary-Based Models

The family of models used most often for English prose are variations of an index-based approach which dynamically generates a dictionary of common substrings [Ziv and Lempel, 1977, 1978]. These methods, referred to as LZ77 and LZ78 respectively, are still the dominant compression model in desktop PC applications. The first method, initially described by Ziv and Lempel [1977], generates a dictionary by maintaining a sliding window of recently seen symbols. Common substrings of the sequence appearing in a lookahead buffer are replaced by references to previously seen phrases in the history buffer, and encoded as a 3-tuple, of the form $\langle p, l, s \rangle$. In the tuple, the previously successful match p, and the match length l, are transmitted along with the symbol s which produced the mismatch. Figure 2.4 illustrates a basic LZ77 approach. Here, the DNA text “`gattacaca`” is the message text. For simplicity, the history and

	Message	Dictionary	Tuple
(a)	g a t t a c a c a t		
(b)	g a t t a c a c a t	1 → g	< 0 , g >
(c)	g a t t a c a c a t	2 → a	< 0 , a >
(d)	g a t t a c a c a t	3 → t	< 0 , t >
(e)	g a t t a c a c a t	4 → ta	< 3 , a >
(f)	g a t t a c a c a t	5 → c	< 0 , c >
(g)	g a t t a c a c a t	6 → ac	< 2 , c >
(h)	g a t t a c a c a t	7 → at	< 2 , t >

Figure 2.5: Dictionary-based compression of a simple DNA string, using an LZ78 model. Each row illustrates a string being encoded, with the corresponding 2-tuple, and the iterative symbol assignments made in the dictionary. The shaded region shows the region of text already processed, and the unshaded region is the unprocessed text.

lookahead buffer are the same length, but in typical implementation the history buffer is much larger than the lookahead buffer. To begin, initial symbol appearances are assigned a default value of $\langle 0, 0, s \rangle$, where s is the current symbol. Once there is sufficient information in the history buffer, longer matches start to occur, as in step (d).

Shortly thereafter, Ziv and Lempel propose an alternative approach, whereby the dictionary is grown one character at a time, removing the need to perform an exhaustive search on a sliding window. This method, referred to as LZ78, is simpler to implement, and still offers reasonable compression effectiveness. Figure 2.5 illustrates the LZ78 approach on the DNA text "`gattacacat`". Each row shows a section of text being encoded as a 2-tuple, $\langle p, s \rangle$, where p is the identifier assigned to the longest substring of the current phrase. For example, step (e) shows how the phrase "ta" is added to the dictionary, as the tuple $\langle 3, a \rangle$, where 3 identifies prefix t, previously assigned in the dictionary. Many variations of these basic approaches have been proposed through the years, primarily because of their modest resource requirements (see, for instance, Welch [1984] or Klein [1995]).

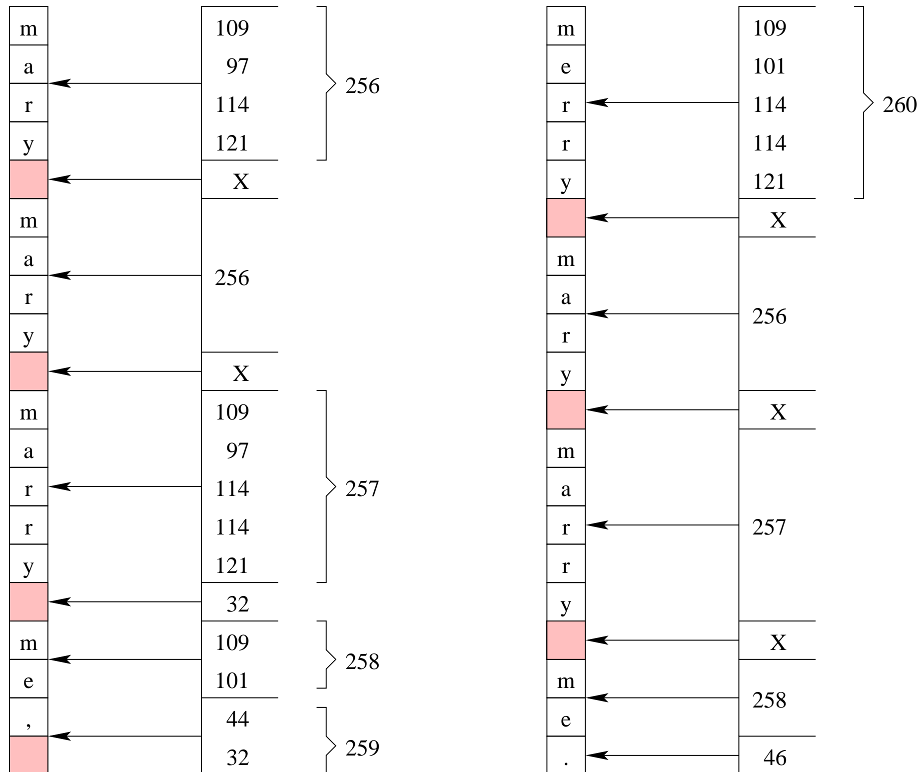


Figure 2.6: A simple spaceless word model which maps words in a text stream to a sequence of positive integers. Adapted from Moffat and Isal [2005].

2.4.3 Word-Based Models

Another possible approach to modeling text is to use a word-based model. If we consider a text $\mathcal{T}$ to be an alternating stream of words and non-words, we can parse $\mathcal{T}$ into two distinct streams [Bentley et al., 1986, Horspool and Cormack, 1987, Ryabko, 1987, Moffat, 1989]. Here a *word* is any contiguous sequence of alphanumeric characters and a non-word or *separator* is any contiguous sequence of non-alphanumeric characters. While this model only applies for languages with separable words, it works well with English text.

Linguistics has greatly contributed to this understanding by providing empirical laws which are applicable to natural language texts. The first empirical law we examine describes the relationship between a power law probability distribution, and

the frequency of words in English text. Zipf's Law suggests that the frequency of a word is approximately inversely proportional to it's *rank* [Zipf, 1949]. More precisely, $w_f = 1/r^{\alpha}$, where r is the rank of the word and α is an empirical constant close to 1. Later work by Mandelbrot [1953] generalized the original results of Zipf to more closely model a power law distribution. The second empirical law we mention is Heap's law, which allows us to accurately predict the growth rate of a vocabulary, relative to the length of the text $\mathcal{T}$ [Heaps, 1978]. According to Heap's law, the size of the vocabulary σ for a text of n words, is approximately $\sigma = kn^{\beta}$, where $10 \leq k \leq 100$ and $0.4 \leq \beta \leq 0.6$ are empirical parameters.

Early approaches modeled words and non-words as two independent contexts, resulting in increased compression effectiveness [Bentley et al., 1986, Moffat, 1989]. A subtle modification to the basic word modeling was proposed by de Moura et al. [2000]. Instead of treating the word and separators as two independent contexts, they can be treated as a single context. Furthermore, all separators which are a single space can be dropped from the output stream and re-interpolated when the original message is reconstructed by the decoder. This method offers a moderate boost in overall compression effectiveness as single spaces are typically the most common symbol in a 0-order character based context. For all practical purposes, the *spaceless word model* has become the de facto standard in word-based modeling in English prose.

One potential drawback of word-based models is the potentially large alphabet size, requiring care when transmitting the prelude to the decoder. There are two basic approaches to transmitting the alphabet when using a word-based model. The first approach is simply to transmit the entire dictionary as a stream of characters separated by EOW symbols, along with an appropriate symbol mapping (perhaps ordered by rank) directly to the coder, which can be costly. Another approach is to build the dictionary adaptively in a manner similar to LZ77 and transmit it implicitly in the output stream, spreading the additional cost over the entire stream. Neither of these approaches is particularly effective. Additional methods for effective prelude transmission are presented in Chapter 4.

In Figure 2.6 we see an example of the word to integer mapping system proposed by Moffat and Isal [2005]. The input text is parsed into an alternating sequence of words and separators. The first appearance of any word or separator is encoded using its ASCII representation. The ASCII sequence is then assigned the next unused codeword greater than 255. For instance, in Figure 2.6 the first appearance of the word "mary" is encoded using its ASCII representation "109-097-114-121" and assigned the codeword "256". All subsequent appearances of "mary" are coded as "256". Note that a strictly alternating sequence of word followed by separator is assumed. If no separator is encoded then two words are assumed to have a single space as a separator.

Semi-static zero-order word models have been used extensively in information retrieval systems [Witten et al., 1999]. For standard file compression, character-based PPM models tend to do better than word models. Little progress has been made on traditional k-order word-based models primarily because of the overhead imposed by the explosive growth in cost to maintain higher order contexts for large alphabets [Moffat, 1989]. Recent research to generate word-based PPM models was considered recently with limited success [Adiego and de la Fuente, 2006]. A more promising approach based on word-based variations of BWT are currently being investigated [Moffat and Isal, 2005].

2.4.4 Other Models

It is also possible to build higher-order word-based models which make prediction based on language grammar [Tehan, 1998, Nevill-Manning and Witten, 1997, 2000, Wan, 2003]. These models construct a set of *production rules* which map *terminal* and *non-terminal* symbols in an ordered fashion. The complexity of generating formal grammars can be difficult for English text, so many of the practical approaches use an adaptation of a simple character-based approach where recursive substitutions of the most common bigrams a produced [Larsson and Moffat, 2000]. These methods are typically off-line, and can be computationally expensive, but are surprisingly effective in many cases [Wan, 2003].

Other variable context models are possible. Two approaches which have favorable effectiveness but are generally less efficient in practice are dynamic Markov compression proposed by Cormack and Horspool [1987] and context-tree weighting [Willems et al., 1995, 1996]. A variable context model which is based on the notion of *context mixing* currently offers state-of-the-art compression [Mahoney, 2005]. The modeling approach used by PAQ is similar to PPM but symbol prediction uses a weighted combination of probability estimates from several pre-conditioned models drawn from different contexts. While PAQ is computationally expensive even on modern hardware, it offers astonishing performance on a variety of data types. For experimental comparisons of PAQ variations, see `http://cs.fit.edu/~mmahoney/compression`.

2.5 Coding

The intuition behind variable length coding and basic properties of codes was covered previously in Section 2.2. Here we focus on the three predominant approaches to generating codes. The choice of coding method is often dictated by the application domain. If semi-static modeling is acceptable, variants of minimum redundancy codes offer flexible solutions with tightly bounded effectiveness. If the model produces intermediates with a heavily skewed distribution and optimal compression is desired, arithmetic codes are a good choice. If performance is paramount and guarantees about optimality are not required, static codes are often a good compromise. We will review minimum redundancy codes in Section 2.5.1, arithmetic codes in Section 2.5.2, and static codes in Section 2.5.3.

2.5.1 Minimum Redundancy Codes

The problem of assigning codewords which minimize the average codeword length, relative to a probability distribution, has been a topic of active research since the very early days of information theory. The first heuristic methods were proposed

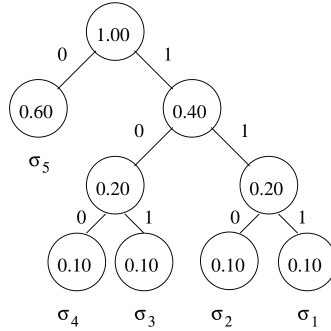


Figure 2.7: A Huffman tree representation for the alphabet $\Sigma = \{(\sigma_1, 0.6), (\sigma_2, 0.1), (\sigma_3, 0.1), (\sigma_4, 0.1), (\sigma_5, 0.1)\}$ where each element tuple is a (name, p_i) pair. Each internal node shows the cumulative probability for its child nodes.

by Shannon and Fano [Fano, 1949]. Shannon-Fano codes take a top-down tree-based approach, where the probability space is recursively partitioned into two equally weighted segments until each segment consists of a single codeword. While there are no guarantees that such codes are minimum redundancy, they worked well in practice.

The challenge of generating *minimum-redundancy codes* which offer some guarantee of optimality was first tackled by one of Fano's graduate students, David Huffman. Huffman codes solve the problem by building a tree bottom-up instead of top-down [Huffman, 1952]. In Figure 2.7 we see a typical Huffman tree construction. As is demonstrated shortly, building an actual tree is not strictly necessary, but does help when trying to visualize how the codes are constructed. In this example, we start with the alphabet $\Sigma = \{(\sigma_1, 0.60), (\sigma_2, 0.10), (\sigma_3, 0.10), (\sigma_4, 0.10), (\sigma_5, 0.10)\}$ ordered by decreasing probability. The algorithm first combines the two members with the lowest probability into a single package, resulting in the intermediate list $\{(\sigma_1, 0.60), (\sigma_4 \mapsto 0 : \sigma_5 \mapsto 1, 0.20), (\sigma_2, 0.10), (\sigma_3, 0.10)\}$. The next step then combines the two smallest elements into a single package $\{(\sigma_1, 0.60), (\sigma_4 \mapsto 0 : \sigma_5 \mapsto 1, 0.20), (\sigma_2 \mapsto 0 : \sigma_3 \mapsto 1, 0.20)\}$. The procedure is continued until there are no more packages to combine. The final code mapping for Σ is $\{\sigma_1 \mapsto 0, \sigma_2 \mapsto 100, \sigma_3 \mapsto 101, \sigma_4 \mapsto 110, \sigma_5 \mapsto 111\}$.

Approaches to tree based codeword assignment have been investigated by several authors mostly in regards to bounding fully dynamic approaches to minimum redundancy approaches [Huffman, 1952, Knuth, 1985, Vitter, 1987, Milidiú et al., 1999]. Minimum redundancy codes need not rely on tree based approaches. In fact, tree based implementations do not provide efficient decoding, since an explicit tree must be walked a single bit at a time to recover the encoded symbol. The only real requirement is that each symbol is assigned a proper length, which corresponds to a valid minimum redundancy arrangement. Codewords can then be assigned using one of many algorithms with table based approaches being favored because of their decoding efficiency. Approaches to *canonical codes* have received significant attention through the years [Schwartz and Kallick, 1964, Connell, 1973, Hirschberg and Lelewer, 1990, Moffat and Turpin, 1997, Liddell and Moffat, 2007]. Destructive array based approaches allow code construction to be performed in linear space [van Leeuwen, 1976, Katajainen et al., 1995, Moffat and Turpin, 1998]. Canonical codes are assigned sequentially based on code length and allow an ordering to be imposed on the alphabet members which can greatly reduce the costs of prelude transmission. Furthermore, implicit knowledge of the length of each symbol allows decoding to proceed as chunks of bits, greatly enhancing throughput [Moffat and Turpin, 1997, Liddell and Moffat, 2006].

Other approaches to constructing minimum redundancy codes have been investigated. Skeleton trees are a recent addition to generating minimum redundancy codes which also allow efficient decoding compared to the use of explicit trees [Klein, 2000]. The basic idea proposed by Klein is to identify flat sub-trees and remove them. The remaining leaves then define either a codeword or the number of bits remaining in the codeword. Finally, a handful of state machine based approaches have also been proposed which have nice theoretical properties but are generally not particularly efficient in practice [Choueka et al., 1985, Siemiński, 1988]. Readers interested in details of these approaches are referred to Moffat and Turpin [2002].

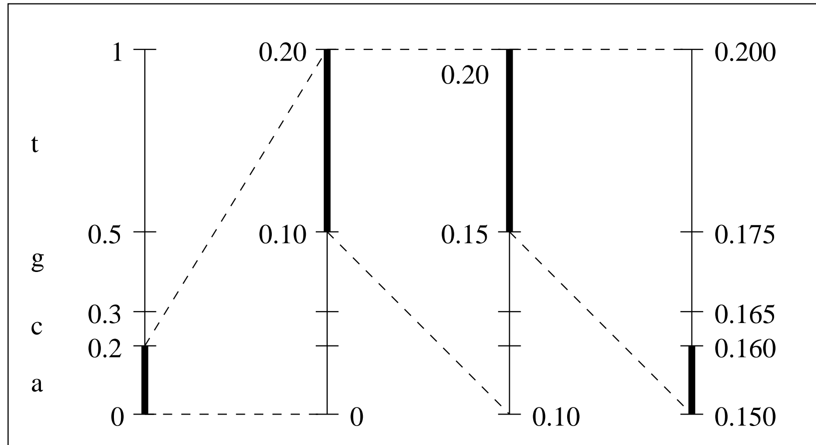


Figure 2.8: Arithmetic coding of the string "`atta`", for the four symbol alphabet $\Sigma = \{a, c, g, t\}$ with $p_a = 0.2$, $p_c = 0.1$, $p_g = 0.2$, and $p_t = 0.5$. Each successive character narrows the range.

2.5.2 Arithmetic Codes

The basic premise of arithmetic codes is to aggregate codewords so that each symbol can be represented using a fractional number of bits. While this approach may seem impossible at first glance, it can be accomplished very efficiently with care. Indeed, the possibility of using fractional bit lengths was mentioned as early as 1948 by Shannon, but valid approaches to implementing such a system seemed difficult until Rissanen and Langdon [1979] and Pascoe [1979] proposed using methods based on finite precision arithmetic. In Figure 2.8, we see a simple example of arithmetic coding in action. In this example, a unique real number is generated in the range $[0, 1)$. Using the alphabet, and corresponding probabilities shown in Figure 2.8, the string "`atta`" is encoded as follows. The first character "a" narrows the interval to $[0, 0.2)$. As each successive character is encountered, the range is further narrowed. Once the last symbol has been read, the resulting range is $[0.150, 0.160)$, which can be represented in binary, minus the decimal as $[\texttt{0010011001}, \texttt{001010001})$. The message can be encoded by any value in this range, with 0.15625 (`00101`) being a reasonable choice. The prelude costs to transmit the alphabet and corresponding probability distribution is extra.

Arithmetic codes are most practical when the model produces highly skewed distributions in an adaptive fashion. Nevertheless, arithmetic codes allow us to encode a text $\mathcal{T}$ to a codeword length of at most $\mathcal{H}_0(\mathcal{T}) + 2$. Minimum redundancy codes, on the other hand, may lose up to one bit per symbol compared to the optimal code length, while arithmetic codes lose at most 2 bits per message. Several practical improvements have been proposed in an attempt to mitigate the costs of generating arithmetic codes [Witten et al., 1986, Moffat et al., 1998]. In addition, substantial improvements have been developed to improve cumulative probability calculations for arithmetic codes [Fenwick, 1994, Moffat et al., 1998, Moffat, 1999]. Despite their attractive bounds, many systems avoid using arithmetic codes because of concerns about litigation related to patent issues, and because they are slower than static and semi-static minimum redundancy codes. Arithmetic codes only come into their own for complex, multi-context, adaptive compression models which generate highly skewed probability distributions.

2.5.3 Static Codes

Static codes are an appropriate choice when the the size of the source alphabet cannot be bound, or resource requirements are limited. Static codes map positive integers onto binary codewords according to a particular probability distribution, meaning that there is an expectation that the model used will produce a stream of integers as an intermediate. An underlying assumption made by all members of this coding family is that the source alphabet exhibits a natural, non-increasing probability distribution. If this is not true for a particular integer sequence, it is advantageous to generate a ranked symbol mapping which permutes the alphabet into one that does have the necessary probabilities.

Many of static coding schemes mix simple unary and minimal binary coding methods to produce a prefix-free variable-length code. A *unary code* represents a natural number, n, as a string of $n - 1$ one-bits followed by a single zero-bit. Despite their simplicity, they are optimally efficient when encoding a sequence following an

exponential probability distribution, with a base of 0.5 or less. In contrast, a *minimal binary code* is optimal for distributions where all members are equiprobable. Minimal binary codes use all possible prefixes for a given set of integers. For an alphabet of σ symbols, the first $2^{\lceil \log \sigma \rceil} - \sigma$ codewords are encoded using $\lfloor \log \sigma \rfloor$ bits and the remaining codewords are encoded using $\lceil \log \sigma \rceil$ bits. For example, the integers 1 – 6 are mapped to the codes $\{\texttt{00}, \texttt{01}, \texttt{100}, \texttt{101}, \texttt{110}, \texttt{111}\}$. One drawback of minimal binary codes are that the require σ_{max} to be known in advance, and they are not infinite.

Extensions to the basic coding techniques were proposed by Elias [1975] and independently by Levenshtein [1968]. These methods are an elegant compromise between binary and unary coding methods. Elias codes can be conceptually partitioned into two variable length blocks. Here, we refer to the first segment as the *selector*, and the second segment as the *locator*. The selector indicates a range which collectively forms buckets of size $1, 2, 4, \ldots, 2^k$, and the locator specifies the exact position of the value in the bucket. Elias γ-codes encode the selector in unary taking $1 + \lfloor \log x \rfloor$ bits and encodes the locator in $\lfloor \log x \rfloor$ bits using a binary encoding, for a total of $1 + 2\lfloor \log x \rfloor$ bits. Elias δ-codes encode the selector using a γ-code instead of unary, resulting in a total of $1 + 2\lfloor \log \log 2x \rfloor + \lfloor \log x \rfloor$ bits to encode an arbitrary integer x. The final code proposed by Elias is referred to as an ω-code. These codes attempt to mitigate the costs for large values by recursively encoding the length of the locator to a set length of 2 bits. Each recursive selector segment is encoded as in minimal binary form. In addition, a 0-bit is appended to the end of the code so that codeword boundaries can be identified. A code similar to an Elias ω-code was proposed by independently by Even and Rodeh [1978].

Instead of using the Elias approach of growing bucket sizes exponentially, Golomb codes use a fixed bucket size b [Golomb, 1966]. Given an arbitrary integer x, and a predefined bucket size b, compute the integer quotient q, and the residue r, using

$$q = 1 + \frac{x - 1}{b},$$

$$r = x - qb.$$

Algorithm 1 Encoding a static byte code

INPUT: a radix R (typically 128), and the original message $\mathcal{T}$.
OUTPUT: The coded message, $\mathcal{Z}$.

```
1: set v ← read_symbol (T)
2: while v ≥ R do
3:    write_byte (v mod R)
4:    set v ← v div R − 1
5: end while
6: write_byte (R + v)
```

Algorithm 2 Decoding a static byte code

INPUT: a radix R (typically 128), and the coded message $\mathcal{Z}$.
OUTPUT: The original message, $\mathcal{T}$.

```
1: set v ← 0 and x ← read_byte (Z) and p ← 1;
2: while x < R do
3:    set v ← v + (x + 1) × p;
4:    set p ← p × R;
5:    set x ← read_byte (Z);
6: end while
7: set v ← v + (x + 1 − R) × p;
8: write_symbol (v − 1);
```

In a manner similar to Elias γ-codes, the quotient q is now encoded using unary, and the residue r is encoded using a minimal binary code. The two parts are concatenated together to produce the final prefix code. If b happens to be a power of 2, then the practical efficiency can be improved through the use of bit shifts to generate the final codes. These special-case Golomb codes are often referred to as Rice codes [Rice, 1979]. Gallager and van Voorhis [1975] show that Golomb codes are in fact worst-case optimal when coding a geometric probability distribution of the form $P = [(1 - p)^{x-1}p \mid 1 \leq x]$, where

$$b = \left\lceil \frac{\log_e(2 - p)}{-\log_e(1 - p)} \right\rceil .$$

The final class of static codes we discuss are the *static byte codes*. Variants of byte codes have been used in commercial relational databases such as Oracle for more than 15 years [Antoshenkov, 1994, Wu et al., 2001]. They are also the focus of several comprehensive empirical studies on compression in information retrieval

Value	Unary	Elias		Golomb	Stopper-Continuer
		γ	δ	$b = 3$	Radix-16
1	0	0	0	0 0	0000
2	10	10 0	100 0	0 10	0001
4	1110	110 00	101 00	10 0	0011
8	11111110	1110 000	11000 000	110 10	0111
10	1111111110	1110 010	11000 010	1110 0	1000 0001

Table 2.1: Codeword comparisons for various integer coding schemes.

systems [Williams and Zobel, 1999, Scholer et al., 2002, Trotman, 2003]. The codes are conceptually simple, and are versatile and efficient in practice. Algorithm 1 highlights the simplicity of generating the codes. In essence, 7 bits in each byte is used to store information, and the remaining bit is used to mark the "end" of a codeword. When using a radix $R = 128$, any symbol s in the range $0 < s < 2^7$ can be coded using a single byte, any s in the range of $2^7 < s < 2^{14}$ is coded using exactly two bytes, and so on. Algorithm 2 demonstrates how sequential byte reads, in conjunction with a few arithmetic operations, result in simple decoding.

Initial work with static byte codes assumed a radix $R = 128$, but any radix can be used in practice. Other radix values do not guarantee that the codewords are byte aligned, potentially resulting in less flexibility in applications such as compressed pattern matching. In essence, the *stopper-continuer model* of prefix coding is, in fact, a generalization of static byte codes. In the stopper-continuer model, a block of 2^R values is separated into two partitions. Then, one partition is used to represent *continuers*, which signal that the next block is also part of the current codeword, and *stoppers* which mark the block at the end of a codeword. This ensures that all codewords instantaneously decodable. This generalization is explored further by Rautio et al. [2002]. A more comprehensive discussion of various byte coding methods can be found in Chapter 4.

Table 2.1 gives examples for various static integer codes, for a subset of values between 1 and 10. Elias, Golomb, and unary codes were discussed previously. The final column shows an example of the more general class of variable block codes. Stopper-

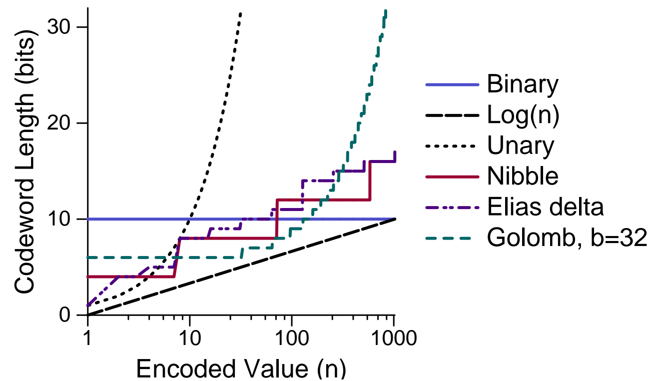


Figure 2.9: Comparison of different integer based coding methods.

continuer codes are typically byte aligned, but, as mentioned previously, any radix is possible. Stopper-continuer codes that use 4 bits per block are referred to as nibble codes.

Figure 2.9 shows how the length of each code grows relative to the encoded value. Static codes following a fixed distribution, such as Elias δ-codes and nibble codes, perform well for a variety of possible distributions. Parameter-based Golomb codes can often perform better than parameterless static codes, but, like binary codes, require knowledge of the sequence to be compressed in order to calculate a parameter value.

There are many other static integer codes discussed in the literature. Many have attractive theoretical properties, if properties of the underlying probability distribution to be encoded are known in advance. Methods such as Fibonacci coding are a standard example of such codes [Apostolico and Fraenkel, 1987, Fraenkel and Klein, 1996]. For further information, the reader is referred to the more comprehensive surveys of integer coding by Fenwick [2003] or Moffat and Turpin [2002].

2.6 Summary

Compression algorithms allow us to store and transmit data efficiently. The versatility of the coding-modeling paradigm results in the discovery of new applications for these classical techniques. When combined with other techniques such as *bit-parallelism*,

external memory algorithms, and locality of access, data compression allows practical manipulation of massive datasets. The role of data compression continues to grow in the design of space-efficient algorithms and in mitigating the transmission costs of data transfers. In the next chapter, we examine the duality of compression and search algorithms and discover how the two seemingly disparate techniques compliment one another.

Chapter 3

Searching in Sets and Sequences

Searching is a pervasive problem in computer science. It is a mature research area and a great variety of practical and theoretical algorithms are known. However, efficient searching in massive data sets remains a problem with tangible applications. There are several approaches for improving the efficiency of various searching operations in large data sets. For instance, algorithms can target localized access patterns which exploit the memory hierarchy; approximation algorithms can be used; or compression can be applied to reduce amount of information processed. Algorithms which are cache oblivious or attempt to minimize block transfers in the memory hierarchy are abundant [Frigo et al., 1999, Vitter, 2001, Kumar, 2003]. Synopsis data structures are another novel approach to certain types of on-line searching problems where bounded approximate solutions are acceptable [Gibbons and Matias, 1999]. Applying compression is also a viable solution to building space efficient search algorithms.

The growing gap between processing and access speeds in the memory hierarchy allow greater flexibility in boosting algorithmic techniques such as searching through the use of compression techniques. Also, we are beginning to better understand the fundamental duality between compression and searching, and how to exploit it to build better algorithms. By incorporating compression into algorithm design, more information is kept close to the processor, and efficiency is improved. With care, certain classes of algorithms can outperform their uncompressed counterparts.

This chapter reviews three scenarios where compression is playing an increasingly important role in algorithm design. In Section 3.1, we examine classical on-line search algorithms and see how compressed pattern matching can increase overall efficiency of these algorithms. Then in Section 3.2, we investigate important indexing methods which are also benefiting from compression. Finally, in Section 3.3, we discuss the key role compression techniques have in succinct data structures.

3.1 On-line Searching in Text Collections

Simple pattern matching algorithms are a core component in many applications today. For applications such as network-based intrusion detection, which make pre-processing the search text prohibitive, algorithms which pre-process the search pattern instead work well in practice. In this section, we discuss the best on-line searching algorithms, and see how incorporating compression can further enhance the process.

3.1.1 Exact Pattern Matching

The *pattern matching problem* is defined as finding all occurrences of a pattern $\mathcal{P} = p_1p_2 \ldots p_m$ in a text $\mathcal{T} = t_1t_2 \ldots t_n$ where both $\mathcal{T}$ and $\mathcal{P}$ are sequences over a finite alphabet Σ of size σ. This is a well studied problem and many efficient solutions have been proposed over the last thirty years. The best approaches use one of three searching techniques and a *search window* to position the pattern relative to the text.

The simplest sliding window implementation is the naive or *brute-force* algorithm. In this case, the window to be considered is simply the length of the pattern, m. Matching is left to right and any mismatch in the window produces one symbol shift, and matching restarts at the beginning of the window. In the worst case, the algorithm executes in $\mathcal{O}(nm)$ time. Depending on the size of the alphabet, and the underlying probability distribution, the algorithm can perform considerably better in the average case.

Other general approaches of interest which can decrease the worst case complexity to $\mathcal{O}(n+m)$, include prefix-based, suffix-based, and factor-based searching. We only discuss a few key results here. Many excellent books are dedicated to this topic, and the reader is referred to these for a more general introduction to pattern matching [Crochemore and Rytter, 2002, Navarro and Raffinot, 2002, Crochemore et al., 2007].

Prefix-Based Approaches

The basic idea behind prefix-based approaches is to pre-compute offsets in the search pattern, and find the longest prefixes of $\mathcal{P}$ that are also suffixes of $\mathcal{T}$. The pattern is then matched against the text from left to right. The first $\mathcal{O}(n+m)$ time algorithm to solve this problem was proposed by Knuth et al. [1977], and is almost always referred to as the KMP approach. The KMP algorithm also has another valuable property, the worst-case performance is independent of the alphabet size σ, making it an attractive alternative in situations where σ is large. More recent algorithms of this class have focused on bit-parallelism for constant factor improvements [Hyyrö et al., 2006]. The shift-or and shift-and algorithms proposed by Baeza-Yates and Gonnet [1992] are capable of finding all pattern occurrences in $\mathcal{O}(n\lceil m/w\rceil)$ time, where w is the size of a computer word in bits. These algorithms work well in practice, and are simple to implement. For small σ or small P, bit-parallel approaches are often the most efficient Navarro and Raffinot [2000]. However, the performance of prefix-based approaches degrades quickly as the length of the pattern m increases, relative to more commonly used suffix-based methods.

Suffix-Based Approaches

Suffix-based approaches are among the most successful algorithms in use today. The key intuition of this approach is based on the observation that longer shifts can be achieved on average if the pattern is matched from right to left in the search window. The first such algorithm was proposed by Boyer and Moore [1977], and has performance which is $\mathcal{O}(mn)$ in the worst case. However, the average case complexity

of the Booyer-Moore algorithm is $\mathcal{O}(n)$, and has been the subject of many practical and theoretical studies [Gusfield, 1997, Navarro and Raffinot, 2002]. Two practical simplifications to the original BM algorithm were proposed by Horspool [1980] and Sunday [1990]. The former is often referred to as BMH and is the most efficient exact pattern matching algorithm in many character-based applications [Navarro and Raffinot, 2002].

The Boyer-Moore family of algorithms use three novel heuristics to maximize the shifts obtained: the right to left scan; the bad character rule; and the good suffix rule. The bad character heuristic uses pre-generated information to locate the rightmost location of some character x in a pattern $\mathcal{P}$, where x is a mismatched character in the text $\mathcal{T}$. The good suffix heuristic uses pre-processing information to identify factors of $\mathcal{P}$ which match a multi-character suffix of $\mathcal{T}$, which, in turn, has been matched in the current search window. If a factor y of $\mathcal{P}$ exists which is equal to the suffix y of $\mathcal{T}$ previously matched, then the pattern can be shifted to align y in $\mathcal{P}$ and $\mathcal{T}$, otherwise the longest prefix of $\mathcal{P}$ which matches a suffix of $\mathcal{T}$ must be computed.

The BMH variant is a simplification of BM where an assumption is made that the bad character heuristic generally gives longer shifts in the search window, for a sufficiently large alphabet Σ. The modification proposed by Horspool is to always use the last character of the search window as the shift character instead of the most recently unmatched character. By making a small modification to the bad character heuristic, the Horspool variant is currently one of the simplest and most efficient algorithms available today. The variant proposed by Sunday uses a similar simplification. Sunday proposed using the next character after the search window to compute the shift instead of the last character of the window. However, the Horspool algorithm is more amenable to programming optimizations such as loop unrolling which makes it faster in practice than the Sunday variant [Navarro and Raffinot, 2002].

Factor-Based Approaches

Research in combinatorics of words as well as word-based automata representations has led to a recent class of search algorithms which exploit factors of strings. Factor-based approaches lead to optimal average case algorithms under certain conditions. The difficulty of finding efficient algorithms in this class is mainly due to the difficulty of identifying the set of all possible factors in a pattern. Advances in efficiently building *suffix automata* have recently made the factoring problem more tractable. The first algorithm for factor-based searching is commonly referred to as Backward DAWG Matching (BDM) [Crochemore et al., 1994]. A bit-parallel version of BDM was recently proposed by Navarro and Raffinot [2000] and is called Backward Nondeterministic DAWG Matching (BNDM). Another efficient modification called Backward Oracle Matching (BOM) was also recently introduced [Allauzen et al., 2001]. Backward Oracle Matching is particularly effective when the pattern $\mathcal{P}$ is longer than w, a machine word. The interesting twist proposed in BOM is that it is sufficient to know which factors of $\mathcal{T}$ are *not* factors of $\mathcal{P}$ in order to maximize shifts in the search window.

3.1.2 Other Approaches to On-line Searching

The prevalence of string matching across all sub-disciplines of computer science has has led to many new problems being derived from the basic pattern matching problem. Efficient algorithms have been devised for many of the variations. We do not cover all of the possible extensions here but do mention two which are widely applicable in algorithmic research: approximate pattern matching, and multi-pattern matching, both of which are described shortly. For a more comprehensive discussion of other common extensions to the basic string matching problem see, for instance, Navarro and Raffinot [2002] and the references therein.

Approximate String Matching

The necessity of pattern matching algorithms which permit errors drives the development of new algorithms for *approximate string matching*. These algorithms are widely

applicable in diverse areas such as computational biology and signal processing. For instance, the role of approximate pattern matching in biological applications is well documented [Gusfield, 1997]. The goal of approximate pattern matching is to find all occurrences of a pattern $\mathcal{P}$ in a text $\mathcal{T}$, allowing a bounded number of *differences* (k) between $\mathcal{P}$ and a substring in $\mathcal{T}$. The most common metric to quantify the difference is called the Levenshtein edit distance [Levenshtein, 1966]. The Levenshtein distance is defined as the minimum number of single-character insertions, deletions, and substitutions necessary to convert $\mathcal{P}$ into a substring of t of $\mathcal{T}$. So, the edit distance $\xi(\mathcal{P}, t)$ is the minimum number of edits to convert $\mathcal{P}$ to t. For example, ξ ($\texttt{meat}$,$\texttt{feet}$) $= 2$ (two substitutions in this case). The original Levenshtein assumes that inserts, deletes, and substitutions have equal weight, but other weightings can be beneficial depending on the application domain. More intricate methods to determine edit distance are also possible.

A large number of approximate pattern matching algorithms are documented in the literature [Navarro, 2001]. The earliest algorithm developed to solve the approximate pattern matching problem was proposed by Sellers [1974]. Sellers's algorithm uses standard dynamic programming techniques to identify possible matches in $\mathcal{O}(nm)$ time. However, early adaptations of Sellers's dynamic programming approaches are memory intensive when m is large.[1] Instead of the traditional dynamic programming tables, it is possible to model the search using using a deterministic finite state automaton [Ukkonen, 1985]. In this approach, each DFA state is a possible column in a dynamic programming matrix. These approaches are amenable to bit-parallel speed-ups which has led to a dramatic improvement in performance for approximate searching algorithms. Bit-parallel approaches to approximate pattern matching now dominate for almost all permissible values of k, m, and σ [Navarro, 2001].

Several efficient bit-parallel variations have been proposed which attempt to balance low order complexity terms based on different pattern and text configurations. The

[1] The original approach of Sellers [1974] implies the use of $\mathcal{O}(nm)$ space to store the dynamic programming array, but the recent textbook by Navarro and Raffinot [2002] show that only $\mathcal{O}(m)$ space is strictly necessary.

earliest approach, suggested by Wu and Manber [1992], uses a dynamic programming array parallelized by rows, and can find all occurrences in $\mathcal{O}(\lceil m/w \rceil kn)$ time. Baeza-Yates and Navarro [1999] propose a variant which is parallelized by diagonals instead and finds occurrences in $\mathcal{O}(\lceil km/w \rceil n)$ time. Building on this work, Myers [1999] suggests computations should be parallelized using differences between consecutive rows or columns, resulting in a worst-case complexity of $\mathcal{O}(\lceil km/w \rceil n)$. Several recent results have simplified the original approach of Myers, which remains the dominant approach to approximate pattern matching today [Hyyrö, 2001, Hyyrö et al., 2006].

In some instances, it can be beneficial to *filter* the text before applying pattern matching. Filtering allows an algorithm to rule out positions in $\mathcal{T}$ which cannot possibly match the pattern $\mathcal{P}$. Filtering still requires one of the algorithms previously discussed to be used in order to evaluate the remaining positions, and remove false matches. An early approach, called PEX, partitions the pattern $\mathcal{P}$ into $k + 1$ pieces [Baeza-Yates and Navarro, 1999]. Using a simple counting argument, it can be shown that at least one of the pieces must appear unchanged in the text. The cost of PEX filtering is $\mathcal{O}(nk(\log_\sigma m)/m)$ and can greatly enhance performance when σ is large. A more recent filtering algorithm proposed by Fredriksson and Navarro [2004], called OPT, reduces the cost to $\mathcal{O}(n(k + \log_\sigma m)/m)$, but appears to be less resilient to variations in the size of σ.

Multiple String Matching

Both exact and approximate pattern matching algorithms can be modified to support multi-pattern matching, in which a set of strings $P = \{p_1, p_2, \ldots, p_r\}$ are searched for over $\mathcal{T}$ simultaneously. Simple solutions repeatedly search over the text $\mathcal{T}$ using a pattern matching algorithm such as BMH r times, for a cost of $\mathcal{O}(r(n+m))$. The earliest multi-pattern matching algorithm to take a more disciplined approach to the problem was proposed by Aho and Corasick [1975]. The approach of Aho and Corasick can be conceptualized as an extension of the KMP algorithm. The algorithm uses an augmented trie constructed from the pattern set, to yield a worst-case complexity of $\mathcal{O}(n + occ)$.

Attempts to modify BM algorithms were less successful, since searching for multiple patterns simultaneously leads to a higher probability of matching individual characters, reducing the shift lengths. Wu and Manber [1992] overcome the problem by using blocks of characters, reducing the possibility that each block appears in one of the patterns, essentially simulating a larger alphabet size. Alphabet lookups are enhanced using a size-limited auxiliary hash table instead of a traditional character index table used by BM-based algorithms. Bit-parallel algorithms are also amenable to multi-pattern searching, but can handle only a few simultaneous patterns, limiting their usefulness in practice. Allauzen and Raffinot [1999] investigate strategies to extend BNDM and BOM. These algorithms work best for small alphabets and large patterns, just as their simple pattern matching counterparts.

3.1.3 Searching in Compressed Text

An intriguing extension to the basic pattern matching problem is to search directly in compressed data. The *compressed matching problem* was introduced by Amir and Benson [1992] and is defined as: Given a *pattern* $P = p_1 \dots p_m$, a *text* $T = t_1 \dots t_u$ and a corresponding *compressed text* $Z = z_1 \dots z_n$ find all occurrences of P in T such that $\{|x|, T = xPy\}$ is satisfied using only P and Z. The naive approach is to decompress the text before performing the pattern matching step. However, recent research has shown that there are attractive alternatives that do not require decompression. The problem continues to receive attention as CPU speed is increasing much faster than the ability to perform I/O operations on secondary storage devices.

Approaches to the compressed searching problem have followed many lines of study. Much of the difficulty lies in identifying the appropriate entry point in the compression system for adding the necessary search method. It is possible to perform the search at the coding level using modified Huffman codes, or in the modeling layer using modified Liv-Zempel (LZ)-based methods. The problem is made more complex when trying to vary the type of search (exact matching, multiple matching, approximate matching and others are all possible). To overcome some of these conceptual hurdles, a

search oriented abstraction was proposed by Kida et al. [1999] called *collage systems*, and serves as a basis for several theoretical studies on compressed pattern matching. However, a large gap between theoretical and practical solutions to efficient compressed pattern matching remains despite the broad applications of such techniques.

Ziv-Lempel Approaches

The earliest investigations into compressed pattern matching were based on LZ and tried to quantify worst-case complexity of the algorithms [Amir et al., 1996, Kida et al., 1998]. The majority of current research in compressed pattern matching involves either LZ78 or LZW. Simple searching in LZ78 was investigated by Farach and Thorup [1998]. Attempts at integrating BM-based searching and LZ78 and LZW were proposed by Navarro and Tarhio [2000] and Shibata et al. [2000]. Integration of bit-parallel algorithms such as Shift-Or and BNDM with LZ78 and LZW have produced reasonable performance as well [Kärkkäinen et al., 2003, Matsumoto et al., 2000, Kida et al., 1999, Fredriksson and Grabowski, 2006]. Practical approaches using LZ77 have been shown to be more difficult to manage [Navarro and Raffinot, 2004]. The first software system to search directly in LZ77 files, called `LZGrep`, was proposed recently by Navarro and Tarhio [2005]. Surveys of compressed pattern matching in LZ compressed text cover results in more detail [Rytter, 1999, 2004].

Huffman-based Approaches

Another viable approach to compressed pattern matching is to target operations at the coding level. Both time and space efficiency bounds on Huffman codes are well understood and serve as a tractable entry point into developing new algorithms [Klein and Shapira, 2001]. Other researchers have proposed tighter bounds using KMP-based searching in conjunction with basic Huffman codes [Shapira and Daptardar, 2006]. More efficient approaches using large alphabets (such as word-based parsing) have also appeared in recent years [Turpin and Moffat, 1997, Fredriksson and Tarhio, 2003, 2004]. However, solutions based on Huffman codes which are easily applied

in practice continue to be elusive, as quick navigation can be cumbersome without simple methods to quickly identify codeword boundaries. This limitation has motivated detailed investigations into coding methods which allow codeword boundaries to be discovered quickly in an on-line fashion.

Byte-aligned Approaches

The use of bytes as the smallest sub-unit in compressed pattern matching was first considered by Manber [1997]. The basic idea, called byte-pair encoding (bpe) was to fix the alphabet size at 2^8, and partition the text into blocks. In the first step of the algorithm, the unassigned character values are determined. Then, the most common character bigrams are identified and given a new codeword from the available symbols, until a maximum of 256 values are assigned to either single characters, or common character pairs. The net effect is a modest reduction in the amount of text left to process. More importantly, any of the standard string matching algorithms can be used directly on the compressed text. The idea is simple and effective. However, the bpe algorithm does not give competitive compression effectiveness when compared to more disciplined approaches such as LZ77 or word-based Huffman coding. The idea of using byte-aligned codes instead of the traditional bit-aligned approaches was an important first step in creating algorithms that are both effective and efficient.

Many of subsequent algorithms proposed for compressed pattern matching are variants of the byte-aligned pattern matching. The first breakthrough came when de Moura et al. [2000] proposed a simple method which, when coupled with a word-based model, combined effective compression with efficient byte-aligned searching. The *tagged Huffman code* devised in their work is among the first algorithms proposed where compressed pattern matching is significantly faster than the generic “decompress then search” baseline. These results led to a flurry of publications on new variants of byte codes [Brisaboa et al., 2003b,a, 2004, 2005]. The new methods improved the compression effectiveness while maintaining the raw efficiency of the tagged Huffman method. Much of the research presented in Chapter 5 is a logical extension of these

recent results, and further details of the family of byte-aligned codes are given in Section 4.1.

3.2 Off-line Searching in Text Collections

For simple patterns and small text, sequential pattern matching offers efficient results in an on-line fashion. If the search text is large and rarely changes, it can be advantageous to use auxiliary data structures to improve search performance. The key difference is that on-line searching is best handled by preprocessing of the pattern $\mathcal{P}$, while off-line searching supports preprocessing the entire text $\mathcal{T}$. When many searches must be performed on the same text, off-line search methods are generally preferred.

There are two common approaches to off-line searching. The appropriate choice is dictated by the composition of the search text. If the text is a natural language text such as English, inverted indexes combined with information retrieval models can be used to greatly enhance search functionality and leverage word-based statistical information. Other text collections which do not have clear-cut word boundaries, or when querying will not be on whole words, must rely on *full-text* indexes which enhance searching by allowing quick substring resolution. Note that the term *full-text* is overloaded in the information retrieval literature. The traditional definition of full-text referred to keyword-based searches on indexes across the complete text, instead of indexes on document titles and abstracts. Henceforth, our definition of full-text indexes follows the new meaning as popularized in the recent seminal work of Ferragina and Manzini [2000]. Examples of collections which benefit from this approach include, but are not limited to, genomic sequence data; and languages such as Chinese, Korean, and Japanese.

Index-based approaches are flexible and efficient in practice. It should be noted that efficiency in all of the index-based data structures has been greatly enhanced by directly incorporating compression. In Section 3.2.1, we review basic inverted indexing along with information retrieval modeling, and see how compression has become an integral part of the modern systems. In Section 3.2.2, we discuss various approaches to full-text

indexing and review the increasingly important role compression has in efficient index design.

3.2.1 Ad Hoc Search Engines

In Chapter 1 we saw an example the extraordinary efficiency of algorithms in modern search engines. These algorithms can process information about billions of documents in a few milliseconds. The key to such performance can be attributed to advances in distributed computing, and a very simple data structure called an *inverted index*. Inverted indexes are carefully constructed to enable efficient ad hoc queries in large natural language text collections and are closely linked to the broader problem of information retrieval. By generalizing the pattern matching problem, information retrieval systems facilitate models which allow ranking of query occurrences by *relevance*. The primary goal of information retrieval systems is to satisfy a user's *information need*. A user provides a query and the information retrieval system *evaluates* the request and returns a list of locations in the collection, ranked by similarity. More formally, an *ad hoc search* takes a query $\mathcal{Q}$ and a text $\mathcal{T}$, partitioned into N documents $\{\mathcal{D}_1, \mathcal{D}_2, \ldots, \mathcal{D}_N\}$, and returns an subset $\mathcal{A}$ of documents ordered by similarity $\hat{S}$ to $\mathcal{Q}$.

The complexity of modern search engines has increased dramatically over the last decade, primarily in response to the massive growth of information available on the word wide web. Figure 3.1 gives a high level overview of a typical web search engine. In order to facilitate efficient querying, the systems perform significant off-line processing to build an inverted file , and support for fast snippet extraction. A user submits a query, typically a list of words and operators, which is expanded into a list of search terms and operation constraints. This list is processed by a ranking module which in turn returns a ranked list of potential solutions to the user, often formatted with a document summary.

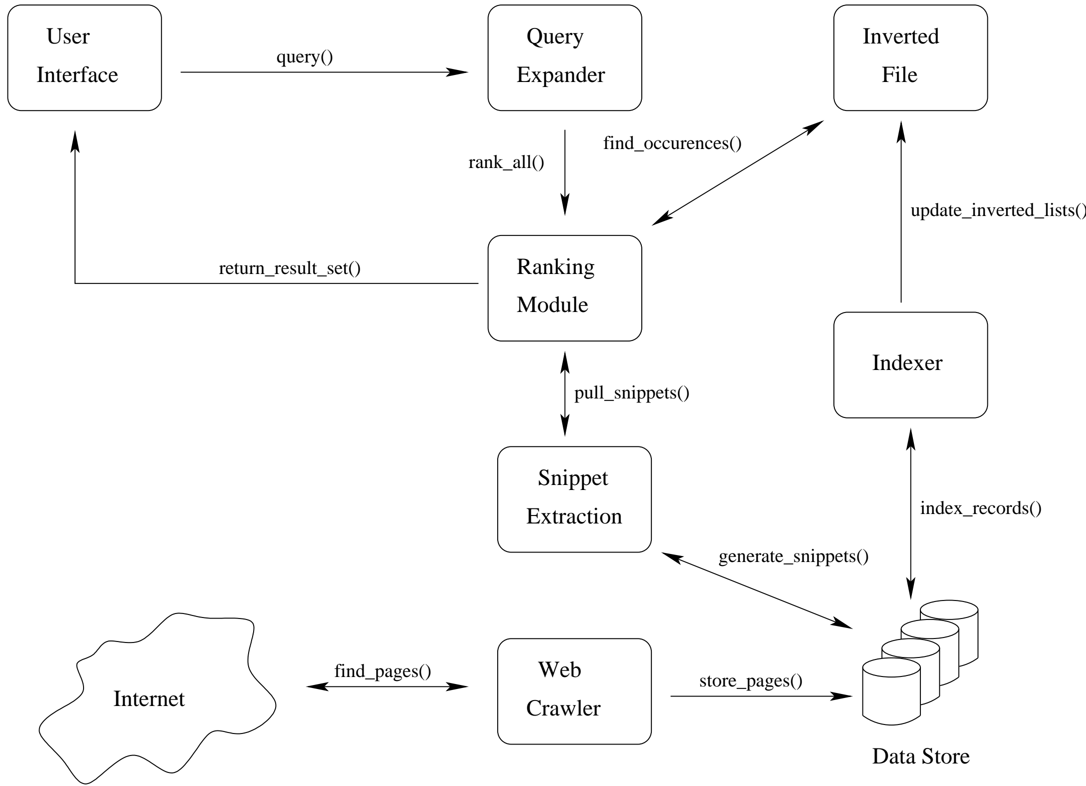


Figure 3.1: A high level overview of the cooperating sub-systems present in a typical web search engine. Meeting a user's information need requires careful coordination between potentially disparate sub-systems.

Ranking and Similarity

Ranking of relevant documents in a large collection is normally accomplished by deriving various statistical coefficients from the text, to create similarity metrics. Developing a robust similarity metric is a difficult task in its own right. A few of the more successful similarity metrics devised include the vector space model [Salton and Lesk, 1968, Salton et al., 1975], probabilistic models [Maron and Kuhns, 1960, Robertson and Jones, 1976], language models [Ponte and Croft, 1998, Zhai and Lafferty, 2001], and latent semantic indexing [Deerwester et al., 1990]. Many of the statistical formulations for similarity are based on a weighting $w_{d,t}$ derived from the frequency of a query term t in a document d. The term's weighting $w_{d,t}$ is generally

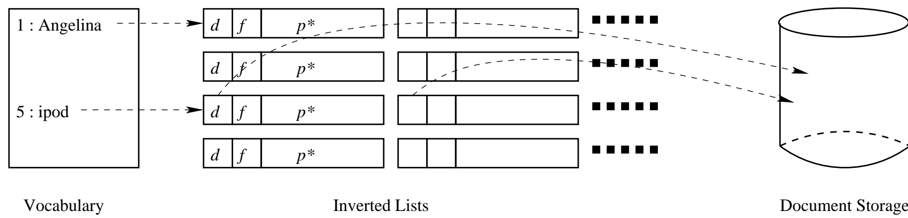


Figure 3.2: An inverted index is composed of three components. Each term in the vocabulary is mapped to an inverted list of $\langle d, f \rangle$ pairs. The document identifier d can in turn be used to locate the original full text document.

derived using a variation of $tf \times idf$ where *tf* is the term frequency and *idf* is the inverse document frequency. Using a $tf \times idf$ metric ensures that the term frequency in an individual document positively affects the similarity contribution, while the term's frequency across all documents negatively affects its contribution. This ensures that terms that are common in all documents do not dominate the more discriminating terms in a query. A comprehensive review of similarity metrics is beyond the scope of this thesis. See Witten et al. [1999], Baeza-Yates and Ribeiro-Neto [1999], Grossman and Frieder [2004], Singhal [1997], or Zobel and Moffat [1998] for overviews of similarity computations.

Inverted indexes

Inverted indexes have been used to enhance search capabilities in document retrieval systems for more than thirty years [Bleier, 1967, Stone, 1987, Witten et al., 1999]. An inverted index is a mapping of a *vocabulary*, and the documents which contain them. The vocabulary is often a subset of words or terms appearing in the document collection, but can be any defined set of attributes. For each term in the vocabulary, an ordered list of tuples called a *postings list* or a *inverted list*, containing a set of pointers, each of which stores a *document identifier* and the term frequency.

In Figure 3.2 we see a common representation for an inverted index. Each term of the vocabulary maps to a list of document identifier and term frequency pairs. Document identifiers are typically represented as unique integers, and can be used

to access the original document text via a document address mapping, or *docmap*. The overall term frequency and the term frequency per document is also stored in the vocabulary to facilitate efficient similarity scoring. For certain advanced query expansion schemes, it may also be advantageous to store a list of position offsets ($p*$) in order to support multiword queries. However, position offsets are not strictly necessary for *bag of word* queries, and can greatly increase the space usage of an index, meaning that they are sometimes stored separately.

Compressed inverted indexes

Efficient representations of inverted indexes typically rely on integer compression techniques. Applying compression to inverted indexes reduces the overall storage and processing overhead associated with large text collections. Compact representations can also dramatically increase performance of the search engine, as less data is transferred between primary and secondary memory.

Early approaches to reduce the overhead of manipulating inverted lists focused primarily on bitvector representations of document identifiers [Bookstein and Klein, 1990]. Conjunctive Boolean queries of q terms can be reduced to the intersection of q ordered sets, and bitvector intersection can be performed efficiently on modern processors. However, bitvectors are prohibitively expensive to store for words which appear infrequently. So, many of the early studies attempted to offset the sparse set costs by breaking the bitvectors into k-bit blocks and using Huffman codes for compression [Jakobsson, 1978, Fraenkel and Klein, 1985]. Extensions to this approach included attempts to remove "zero" blocks from the representations to enhance effectiveness [Choueka et al., 1986, Bookstein and Klein, 1992]. Many other variations on efficient bitvector representations resulted in modest compression gains [Bookstein and Klein, 1991b,a,c, Schuegraf, 1976, Choueka et al., 1987, 1988, Faloutsos and Jagadish, 1992].

Other approaches to efficient representations focused on compact representations of ordered integer sequences. Instead of attempting to represent a sparse universe U efficiently, it is possible to only store the n items in a set in compressed format. As discussed in Section 3.2.1, an inverted list representation is an ordered list of tuples containing document identifier and term frequency pairs. But, the values in the tuple have quite different statistical properties, and separating the two at implementation time allows effective modeling to occur. For example, the document identifiers in each postings list are unique and ordered, in the form:

$$\langle d_1, d_2, d_3, \ldots, d_{f_t} \rangle,$$

where f_t is the total number of documents that contain the term. This list can be transformed into a list of *first-order differences* or d-*gaps*, and represented as

$$\langle d_1, d_2 - d_1, d_3 - d_2, \ldots, d_{f_t} - d_{f_t - 1} \rangle.$$

If position offsets are stored in the inverted index in order to facilitate multiword query support, they too can benefit from this representation.

Sequences of d-gaps are more amenable to integer compression, as term clustering often occurs in large text collections, producing runs of small gaps, which are highly compressible. If the sets are stored as d-gaps, the average size of a typical document identifier list can be dramatically reduced using one of the integer coding methods discussed in Section 2.5.3. The family of static integer coded proposed by Elias [1975] are simple and effective methods to encode the d-gap document identifier lists, and the corresponding term frequencies. Parameter-based Golomb codes, Rice codes, and their derivatives also proved to be a quite versatile method for index compression [Golomb, 1966, Gallager and van Voorhis, 1975, Teuhola, 1978, Rice, 1979]. In the early 1990s, this sequence of ideas was further explored by [Witten et al., 1991, Moffat and Zobel, 1992a,b, Zobel et al., 1992, Bell et al., 1993].

However, the enhanced compression effectiveness does not come without a cost. The lists must now be accessed sequentially, which can be expensive for longer lists. It is possible to use auxiliary information to improve navigation in compressed lists of d-gaps [Moffat and Zobel, 1996, Anh and Moffat, 1998]. New approaches to balance efficiency and effectiveness in compressed sets are considered in Chapter 6. Others approaches to enhance compression effectiveness in inverted lists attempt to exploit clustering inherent in lists of document identifiers [Moffat and Stuiver, 2000, Blandford and Blelloch, 2002, Silvestri et al., 2004]. A small amount of compression effectiveness can also be traded for processing efficiency using variable length byte codes [Williams and Zobel, 1999, Scholer et al., 2002, Trotman, 2003], or word-aligned codes [Anh and Moffat, 2005, 2006]. All of these themes are returned to in subsequent chapters.

3.2.2 Full-Text Indexes

Inverted indexes are a robust solution for search problems dealing with readily-parsable natural language text. However, there are other applications of searching where the underlying text $\mathcal{T}$ does not include explicit word boundaries. For example, searching in large DNA databases, or searching in languages such as Chinese or Japanese, require a different approach to building efficient indexes. Such indexes are generally referred to as *full-text indexes* and are free of assumptions about word boundaries. This makes them a more flexible approach in certain cases, but full-text indexes typically require more space than inverted indexes, making them attractive targets for compression techniques. One of three related data structures is often the core of such systems: suffix trees [Weiner, 1973], suffix arrays [Manber and Myers, 1993], or directed acyclic word graphs [Blumer et al., 1985]. We will look at each of these data structures in turn and discuss recent advances which effectively apply compression directly to full-text indexes.

Indexing data structures

Full-text indexes are typically extensions of data structures which facilitate fast pattern matching such as suffix trees, suffix arrays, and directed acyclic word graphs. All of these structures are derived from early work on suffix trees which are in turn derived from pioneering work on digital search tries [Fredkin, 1960]. Suffix trees are one of the most important data structures in stringology. Examples of their diversity include: biological applications such as genome alignments, tandem repeats, motif discovery, and promoter identification [Gusfield, 1997, Ko and Aluru, 2006, Rajasekaran, 2006]; frequent pattern mining [Fischer et al., 2005]; and data compression techniques such as LZ [Ristov and Laporte, 1999, Chen et al., 2007] and BWT [Ferragina and Manzini, 2004]. Grossi and Italiano [1993] survey the historical importance of suffix trees and highlight the many times the basic structure has been rediscovered.

Weiner [1973] describes a suffix-based tree which can in turn be converted into a more succinct representation. Each suffix is represented by a unique leaf storing its starting position. Traversing the tree from the root allows efficient access to any substring in the text. Weiner goes on to describe an algorithm to construct suffix trees directly from the original text in linear time. Shortly thereafter, McCreight [1976] described a simpler approach to suffix tree construction, which also reduced the space cost by about 25%. The on-line construction of suffix trees in linear time remained an elusive problem for some time. Ukkonen [1995] describe the first on-line construction algorithm for suffix trees, which begins with the first character of the string and adds each successive character. Previous approaches parsed the text back to front.

Suffix trees can consume substantial memory as a result of explicitly storing the navigational pointers in the representation. As a result, Manber and Myers [1993] proposed a more compact data structure called a *suffix array* to store the same information. Suffix arrays can be constructed directly from suffix trees using $\mathcal{O}(n \log n)$ bits of space during construction, where n is the number of characters in the original text. The suffix array is constructed by visiting each leaf of a suffix tree left to right and storing the positions directly. Despite the ability to construct

suffix arrays directly from suffix trees, which in turn can be constructed in $\mathcal{O}(n)$ time, direct construction of suffix arrays in linear time remained elusive for a decade after the data structure was first described by Manber and Myers [1993]. In 2003, three different approaches to suffix array construction in linear time suddenly appeared [Kärkkäinen and Sanders, 2003, Kim et al., 2003, Ko and Aluru, 2003, 2005]. Puglisi et al. [2007] review the multitude of algorithms for suffix array construction now known and provides a detailed investigation of various time and space tradeoffs in the algorithms. The biggest concern for using suffix arrays with massive data sets remains the prohibitive memory cost at construction time, typically on the order of $\mathcal{O}(5n)$. However, suffix arrays are considerably more space efficient than traditional suffix trees after the index is constructed. This has led to wide-scale adoption of suffix arrays as the basic building block for many of the more advanced index methods in use today. Finding all occurrences of a pattern in a pre-processed text can be performed in $\mathcal{O}(m \log \sigma + occ)$ in a suffix tree and $\mathcal{O}(m + \log n + occ)$ in a suffix array. Both require $\mathcal{O}(n \log n)$ bits of space, but suffix arrays are generally smaller by a constant factor [Navarro and Mäkinen, 2007].

Another viable approach to full-text indexes which has received less attention than suffix-based trees and arrays is the directed acyclic word graph (DAWG) [Blumer et al., 1985]. The DAWG is also referred to as a suffix DAWG or *suffix automaton* by several authors [Crochemore et al., 2007]. These structures have found applications in several string processing tasks. Finding all repetitions of a pattern in a text using a DAWG and a detailed discussion of the structure of the automaton was considered in detail by Crochemore [1986].

Compressed full-text indexes

Suffix trees, suffix array, and DAWGs all offer an efficient solution to finding all pattern occurrences in pre-processed text. However, none of the basic solutions are particularly space efficient or easily amenable to use on external memory devices [Clark and Munro, 1996, Kurtz, 1999, Hunt et al., 2001]. So, in order to apply these data structures in

applications which must process massive data sets, less voluminous representations are necessary.

Compact directed acyclic word graphs (CDAWGs) were introduced by Blumer et al. [1987]. A CDAWG is produced by compacting the edges of a DAWG and augmenting them with additional information. Crocehmore and Vérin [1997] highlight the similarities between suffix trees and CDAWGs. Using compaction and minimization (in the sense of automata theory), they show a direct relationship between suffix tries, suffix trees, DAWGs and CDAWGs. A linear on-line algorithm for construction of CDAWGs inspired by Ukkonen [1995] was proposed by Inenaga et al. [2005]. For small alphabets, CDAWGs offer competitive space economy to suffix-based trees and arrays, but no empirical studies of their relative efficiency has been published to date.

Early approaches to dramatically reduce the size of suffix trees and suffix arrays met with limited success. A hybrid version of a suffix tree and suffix array called a *suffix cactus* was proposed by Kärkkäinen [1995]. Space usage for the suffix cactus is somewhere between a suffix tree and and suffix array and pattern matching requires $\mathcal{O}(m\sigma + occ)$ time. Shortly thereafter, Kärkkäinen and Ukkonen [1996] investigated another variation called a *sparse suffix tree*, which represents a subset of suffixes in the text. However, if every k th suffix is taken, searching can take up to $\mathcal{O}(kn)$ time, which may be unacceptable when n is large.

Alternative methods to minimizing space usage in suffix trees were investigated by Munro et al. [2001a], and require $n \log n + \mathcal{O}(n)$ bits of space and support queries in $\mathcal{O}(m + occ)$ time. The same time bound for suffix arrays was obtained without making a standard-unit RAM model assumption by using two auxiliary lookup tables [Abouelhoda et al., 2002]. Perhaps the most promising early method was introduced by Kärkkäinen and Sutinen [1998], and called the LZ-*index*. The LZ-index can process queries in $\mathcal{O}(m + occ)$ time and requires $\mathcal{O}(n)$ bits of space for patterns shorter than $\log n$, otherwise the index occupies $\Theta(n \log n)$ bits. A practical solution to reducing the size of suffix arrays called a *compact suffix array* was proposed by Mäkinen [2000] whereby some of repetitions in a suffix array are replaced by links to other positions.

Mäkinen's solution concisely highlights the similarity of DAWGs, CDAWGs, suffix arrays and suffix trees.

Research in the late 1990s produced consistent improvements to space-economical representations in full-text indexes. However, access to the original text was still necessary in order to process the occurrences, making these approaches unwieldy in practice. This dramatically changed when Ferragina and Manzini [2000, 2005] introduced the first full-text self-index, which can reconstruct the original text directly from the index, without explicitly storing the text. Ferragina and Manzini referred to the new data structure as an FM-*index* and elegantly combined elements of suffix arrays, BWT modeling, and MTF coding to support backwards searching directly in the compressed representation. The FM-index uses at most $5n\mathcal{H}_k + o(n)$ bits of space and can find all occurrences of a pattern in $\mathcal{O}(m + occ \log^{1+\epsilon} n)$ time where $\mathcal{H}_k$ is the kth order empirical entropy and ϵ is a constant $1 < \epsilon \leq 1$. However, the original FM-index worked only for constant σ, resulting in several iterative improvements.

Several different self-index variations were introduced in short order. In seminal work, Grossi and Vitter [2000, 2005] proposed the first index which guaranteed a space usage of $\mathcal{O}(n)$ bits while supporting queries in $\mathcal{O}(m/\log_\sigma n + \log_\sigma^\epsilon n)$ time. However, the *compressed suffix array* (CSA) still required the original text to list pattern occurrences. Sadakane [2000, 2003] quickly extended the CSA to a self-index representation which used $\mathcal{O}(n(\mathcal{H}_0 + \log\log n))$ bits, where $\mathcal{H}_0$ represents the zeroth order entropy, and query support in $\mathcal{O}(m \log n + occ \log^\epsilon n)$ time. Additional time-space trade-offs in CSAs were explored by Rao [2002].

Alternative approaches to self-indexing include the LZ-index, and the compact suffix array. The *compressed compact suffix array* (CCSA) proposed by Mäkinen and Navarro [2004] augmented the compact suffix array of Mäkinen [2000] with a self-index. The CCSA supports queries in $\mathcal{O}(m + occ \log n)$ time and uses $\mathcal{O}(n\mathcal{H}_k \log n)$ space. Ferragina and Manzini [2005] consider a self-index variation similar to the LZ-index proposed by Kärkkäinen and Ukkonen [1996] which can perform queries in $\mathcal{O}(m + occ)$ time and $\mathcal{O}(n\mathcal{H}_k \log^\epsilon n) + o(n)$ bits space where $0 < \epsilon < 1$.

The most efficient full-text index in space and almost always in time is currently the Alphabet Friendly FM-index [Ferragina et al., 2004]. The Alphabet Friendly FM-index can perform queries in $\mathcal{O}(n + occ \log^{1+\epsilon} n)$ time and uses $n\mathcal{H}_k$ space. Time-space improvements as well as applications for FM-indexes, LZ-indexes, and CSAs are currently being produced at an astonishing rate, and are beyond the scope of this thesis. The reader is referred to a recent comprehensive survey on compressed full text indexes by Navarro and Mäkinen [2007] for recent advances, and further references therein.

3.3 Searching in Sets

Manipulation of sets in computer science is a standard problem, and algorithms to support common operations were among the first developed. When choosing an appropriate representation for a set, it is prudent to first identify the subset of key operations which must be supported. Targeted operations can improve the efficiency and reduce space usage, at the cost of functionality.

There are two broad categories of operations on sets. Operations which return information about the set are called *query operations*. Operations which change the contents of a set are called *modifying operations*. At the most basic level, set search and manipulation can be reduced to the classical *dictionary problem* by requiring only three key operations:

INSERT(S, x),	Return $S \cup x$.
DELETE(S, x), and	Return $S - x$.
MEMBER(S, x).	Return **TRUE**, and a pointer to x if $x \in S$; otherwise return **FALSE**.

The operations **INSERT** and **DELETE** are modifying operations, and **MEMBER** is the only query operation. A data structure which supports all three operations is called a *dynamic dictionary*. Alternately, a *static dictionary* does not support the basic modifying operations after the initial dictionary is constructed. Static dictionaries, such as a sorted array, can lead to simpler set representations, and efficiently support targeted queries.

Dictionaries are often used to build more intricate data structures which support more advanced set operations. These "high" level operations typically manipulate multiple elements in a single operation. A subset of advanced operations of interest which affect multiple set members include:

INTERSECT(S,T)	Return $S \cap T$.
UNION(S,T)	Return $S \cup T$.
DIFFERENCE(S,T)	Return $S - T$.
EQUIVALENCE(S,T)	Return **TRUE** if $S = T$, otherwise return **FALSE**.
SPLIT(S,x)	return two sets $S' : \{y \in S \mid y < x\}$ and $S'' : \{y \in S \mid y \geq x\}$.
RANGE(S,x,y)	Return all elements in $S \cap [x \dots y]$.

Note that **SPLIT** and **RANGE** assume that the set is ordered. Supporting efficient operations which evaluate multiple set members has fostered the development of new primitive operations. These primitive operations serve as components in more complex query and modifying operations, and universally assume that the set is ordered over an integer domain. Common primitive operations include:

PREDECESSOR(S)	Return a pointer to the element in S that immediately precedes the current one.
SUCCESSOR(S)	Return a pointer to the element in S that immediately follows the current one.
F-SEARCH(S,x)	Return a pointer to the least element $y \in S$ for which $y \geq x$, where x is greater than the value of the current element.
RANK(S,x)	Return $\lvert\{y \mid y \in S \text{ and } y \leq x\}\rvert$.
SELECT(S,r)	Return a pointer to the r th largest element in S.

Membership Queries

If efficient **MEMBER** queries are needed, hash tables reliably provide constant time query performance on average. Early approaches included hashing with separate

chaining [Dumey, 1956] and hashing with open addressing [Peterson, 1957, Ershov, 1958]. Both methods are efficient in practice but do not offer constant time worst-case guarantees for all data. For a comprehensive history of hashing, see, for instance, Knuth [1998], Vitter and Flajolet [1990], and Czech et al. [1997]. Here, we focus our discussion on techniques which offer constant time **MEMBER** queries, and bound space usage.

The first hashing technique to offer $\mathcal{O}(1)$ time membership was described by Carter and Wigman [1979]. In their scheme, often referred to as *universal hashing*, all basic dictionary operations can be performed in $\mathcal{O}(1)$ time and require $\mathcal{O}(n^2)$ space as well as a collection of suitable hash functions. At construction time, a hash function is selected at random from a uniform collection of "good" hash functions. Dietzfelbinger et al. [1992] subsequently showed random selection to be strictly necessary, ruling out the possibility of devising deterministic selection algorithms. For many simple dictionary applications, these results are the only ones necessary. However, the $\mathcal{O}(n^2)$ space requirement is unacceptable for large dictionaries.

In seminal work, Fredman et al. [1984] lowered the space bound to $\mathcal{O}(n)$ while maintaining a construction time of $\mathcal{O}(n)$ using two level perfect hashing. The FKS hashing scheme uses a two-level data structure where each element of a top level array of size $\mathcal{O}(n)$ contains a pointer to a unique second-level array $A_0 \dots A_{n-1}$. The second-level arrays $A_0 \dots A_{n-1}$ add a total of $\mathcal{O}(n)$ additional space. Each element access therefore requires two table lookups using two distinct hashing functions, one for each level of the data structure. Devising "good" hash functions for the FKS data structure can be problematic and simplified hashing functions have subsequently been explored by Dietzfelbinger et al. [1994, 1997]. The biggest drawback of the FKS hashing scheme is that it does not support **INSERT** and **DELETE**. Dynamizing the FKS structure is possible by relaxing the tight space bounds [Dietzfelbinger et al., 1994].

A recent two-table approach proposed by Pagh and Rodler [2004] has dramatically simplified dynamic dictionary design. The novel approach referred to as *cuckoo hashing* uses two tables A_1 and A_2, each of size $m = n/\alpha$. Any element x is

guaranteed to be stored in either $A_1[x_{h_1}]$ or $A_2[x_{h_2}]$, each of which is uniquely access by hash function h_1 and h_2 respectively. The key insight which allows simple constant time **MEMBER** queries relies on careful **INSERT** operations. Given a new element x, the **INSERT** operation will first try to store x at location $A_1[x_{h_1}]$. If the position is occupied by another element y, replace y with x and attempt to store y at location $A_2[y_{h_2}]$. If $A_2[y_{h_2}]$ is occupied by some element z, replace z with y and attempt to store z at $A_1[z_{h_1}]$. The process ends when an empty table position is located or a constant number of failures has occurred. If the failure state occurs, two new hash functions can be selected from a collection and the table is rebuilt from scratch. The probability of reconstruction has been shown to be $\mathcal{O}(1/n)$ [Devroye and Morin, 2002]. Along with an expected insertion cost of $\mathcal{O}(1)$, this makes cuckoo hashing a simple and effective data structure in practice.

Maintaining constant time **MEMBER** queries in minimal space has received considerable attention. Space-efficient representations aspire to using space proportional to equation 2.3. The simplest approach within a constant factor of optimal space usage which still supports constant time membership queries is an unadorned bitvector [Buhrman et al., 2002]. However, sparse sets in which $n \ll u$ are not efficiently handled by a bitvector. A more detailed discussion of bitvectors can be found in Chapter 6. The FKS data structure uses $\mathcal{O}(n \log u)$ bits of space. Brodnik and Munro [1999] describe a succinct data structure which supports constant time membership using an auxiliary data structure which within a constant factor of the information theoretic optimal $\mathcal{O}(\lceil \log \binom{u}{n} \rceil)$ bits. Pagh [2001] reduces the total cost to $o(n) + \mathcal{O}(\log \log u)$ bits by using a folklore technique called *quotienting* (originally described by Knuth [1998]). Both of these approaches have high constant factors associated with them in practice, but are significant advances in their own right. However, the random nature of the set ordering limits their usefulness when constructing data structures to support more complex operations.

Predecessor and Successor Queries

Other primitive operations, such as **PREDECESSOR** and **SUCCESSOR**, require an ordering to be imposed on the set representation. Both operations exhibit equivalent algorithmic difficulty and, as a result, many of the key results focus only on **PREDECESSOR**. Classical data structures for supporting **PREDECESSOR** are balanced binary search trees (for example AVL trees, Red-Black trees, or 2-3 trees), which can perform **MEMBER**, **INSERT**, **DELETE**, **PREDECESSOR**, and **SUCCESSOR** in $\mathcal{O}(\log n)$ time [Cormen et al., 2001]. This is optimal for comparison based data structures.

Faster algorithms are possible when the data structure supports indirect addressing. For example, a set S drawn from a universe of size u can be stored in an ordered array of u words of memory to support **MEMBER**, **SUCCESSOR**, and **PREDECESSOR** queries in constant time. Each element is simply stored in a unique location. If we consider sets where n is the number of elements in S and $n \ll u$, more principled approaches are necessary. If the size of the universe is limited such that a single machine word can hold an arbitrary element, more interesting solutions are possible. van Emde Boas [1977] proposed the first sub-logarithmic tree-based solution which supports **MEMBER**, **PREDECESSOR**, and **SUCCESSOR** in $\mathcal{O}(\log\log u)$ time and $\Omega(u)$ space. The introduction of y-fast tries by Willard [1983] lowered the space bound to $\mathcal{O}(n)$ but requires randomized rehashing at levels of the trie to support **INSERT** and **DELETE** in $\mathcal{O}(\log\log u)$ time.

Query time can be further improved by using trees of higher degree [Fredman and Willard, 1993]. A *fusion tree* stores additional information in each node so that the correct search path can be determined in constant time per node. Fusion trees are a variation of the packed B-trees of Bayer and McCreight [1973], with a branching factor $b = (\log n)$. This reduces the cost of dictionary operations, **PREDECESSOR** and **SUCCESSOR** to $\mathcal{O}(\log n/\log\log n)$. This approach can be further generalized to trees of any degree b to support queries in $\mathcal{O}(\log_b n)$ with $\mathcal{O}(n)$ space usage, and $\mathcal{O}(n)$ construction time [Hagerup, 1998, Willard, 2000]. A further improvement

to generalized B-trees was proposed by Andersson [1995] to support **MEMBER** and **PREDECESSOR** in $\mathcal{O}(\sqrt{\log n})$ time, expected $\mathcal{O}(\sqrt{\log n})$ time for **INSERT** and **DELETE** while using only $\mathcal{O}(n)$ space. Building on the work of Andersson and Thorup [2000], a tight lower time bounds for **PREDECESSOR** and consequently **SUCCESSOR** of $\Theta(\sqrt{\log n/\log\log n})$ was derived by Beame and Fich [2002].

Finger Search Queries

Finger search queries are often used to facilitate more complex operations which require incremental updates or searches in the vicinity of a previous operation. A simple representation for supporting **F-SEARCH** on static lists is an ordered array of integers. An **F-SEARCH** is possible on an array of integers using binary search in $\mathcal{O}(\log n)$ time. Other search algorithms such as interpolative, exponential, or sequential search are possible. A detailed exploration of time and space-efficient techniques for **F-SEARCH** implementations in a static setting can be found in Chapter 6.

Dynamic data structures which support **F-SEARCH** typically require additional auxiliary information. Balanced tree-based approaches include B-trees [Guibas et al., 1987], $(2,4)$-trees Huddleston and Mehlhorn [1982], and splay trees [Sleator and Tarjan, 1985, Cole et al., 2000, Cole, 2000]. The B-trees of Guibas et al. and the $(2,4)$-trees of Huddleston and Mehlhorn support **INSERT** and **DELETE** in amortized constant time and **F-SEARCH** in $\mathcal{O}(\log d)$ time, where d represents the rank difference between the previous element in the ordered set and the target element. Splay trees support **INSERT**, **DELETE**, and **F-SEARCH** in $\mathcal{O}(\log d)$ time with an amortized initialization cost of $\mathcal{O}(n)$. Other randomized solutions exist, including skip lists [Pugh, 1990] and treaps [Seidel and Aragon, 1996]. Many of these approaches trade space via additional routing pointers for time efficiency.

A space efficient alternative to $(2,4)$-trees was proposed by Blelloch et al. [2003]. The data structure of Blelloch et al. requires no level links or parent pointers. Instead, it uses an auxiliary data structure referred to as a *hand* which adds an additional $\mathcal{O}(\log n)$ space in order to keep track of the current finger pointer. Space efficient treaps were

investigated by Blandford and Blelloch [2004] and deterministic, space-efficient skip-lists were briefly investigated by Boldi and Vigna [2005]. Overall, few examples of space-efficient **F-SEARCH** data structures exist in the literature, and their absence serves as motivation for much of the work presented in Chapter 6.

Rank and Select Queries

The **RANK** and **SELECT** operators are quickly becoming the operators of choice to build space-efficient data structures. The origins of **RANK** and **SELECT** come from the pioneering work of Jacobson [1988]. Jacobson was investigating data structures built on bitvectors which could support efficient **PREDECESSOR** and **SUCCESSOR** queries. From his work, **RANK** and **SELECT** were identified as the key primitive operators necessary to support a bevy of more complex operations. Jacobson proposed a data structure which supports **RANK** and **SELECT** and, as a consequence, **PREDECESSOR** and **SUCCESSOR** in $\mathcal{O}(\log \log u)$ time using $u + o(u)$ bits of space. Munro [1996] subsequently describes a data structure called a *indexable bitvector* which supports **RANK** and **SELECT** in $\mathcal{O}(1)$ time. There have been multiple improvements to augment the work of Munro and incrementally improve the space bounds (See [Munro and Rao, 2005, Navarro and Mäkinen, 2007, Raman and Rao, 1999, Raman et al., 2002, Miltersen, 2005, Golynski, 2006, Sadakane and Grossi, 2006, Golynski et al., 2007] and the references therein).

Another valuable application of **RANK** and **SELECT** is to navigate succinct tree representations. Preliminary work by Jacobson [1988] and Clark [1996] investigated succinct tree representations and efficient parent and child navigational operators built on **RANK** and **SELECT**. Generalizations to support ordinal, cardinal, and multi-way trees have been the subject of many investigations [Benoit et al., 1999, Munro and Raman, 2001, Munro et al., 2001b, Raman et al., 2002, Raman and Rao, 2003, Geary et al., 2004, Jansson et al., 2007, Barbay et al., 2007, He et al., 2007]. Work based on **RANK** and **SELECT** continues to be one of the most fruitful areas of fundamental algorithmic research to date.

3.4 Summary

Compression and information theory serve a fundamental role in designing efficient search algorithms of all varieties. Remarkable advances in efficient query support for compact data structures have been made over the last decade. However, an efficiency gap between theory and practice remains for many succinct data representations. Hidden constant factors can render a promising approach unusable in all but the simplest of applications. In the following chapters, we focus on practical improvements to compressed representations which allow efficient search queries. In particular, we devise variable length coding methods which utilize various alignment properties to support fast search operations in compressed sequences and sets. We conclusively show that searching in massive data sets can be simple and efficient.

Chapter 4

Practical Coding

Compression systems typically emit a stream of bits that represents the input message, but other representations are also possible. One interesting alternative is to use bytes instead of bits as the output unit. Byte-aligned codes have a number of interesting properties that make them attractive in practical compression systems: they are easy to construct; they decode quickly; and often byte-oriented algorithms, such as string searching, can be applied directly to the compressed data. The benefits of byte-aligned codes have been championed in several different domains recently. The canonical reference for applying byte-aligned codes in word-based applications is de Moura et al. [2000]. A comprehensive discussion of variable byte integer codes, and their role in inverted file compression was initially explored by Scholer et al. [2002], and referred to as `vbyte`.

In this chapter we present a new variation of byte code in which the first byte of each codeword completely specifies the length of the codeword. The new coding approach is a compromise between the strictly static models of the variable byte codes employed by Scholer et al., and the full power of a radix-256 Huffman code considered by de Moura et al. In particular, the set of constraints we describe provides better compression effectiveness than previous restricted byte codes, and better decoding throughput than a full Huffman decoder. We also explore alternative prelude representations in order to improve overall performance. One prelude representation in particular results in a

net gain in compression effectiveness and decoding efficiency, highlighting the key role preludes have in designing efficient coding techniques. *Semi-dense preludes*, described in Section 4.3.4, attempt to balance effectiveness and efficiency by generating a true probability ranking for a subset of "important" symbols, and using a default encoding for the remaining symbols.

4.1 Previous Work

Byte-aligned codes have been a staple in various database coding regimes for more than a decade. Early investigations in alignment constrained coding techniques were performed commercially to improve the performance of relational databases [Antoshenkov, 1994, Antoshenkov and Ziauddin, 1996]. The early coding methods typically traded up to 30% of the compression effectiveness in order to perform various join and intersect operations more efficiently. In particular, they require no additional information to be supplied to the decoder and finding codeword boundaries is trivial. These properties have led to several enhancements being proposed to capitalize on the flexibility of byte aligned codes. This section summarizes previous work on byte-aligned codes, starting with the seminal work of de Moura et al. [2000].

4.1.1 Tagged Huffman Codes

Perhaps the most straightforward way to produce byte-aligned codes is to apply length limiting techniques to Huffman codes. For example, Huffman-based codes which are byte-aligned can be constructed using trees of degree 256. Variations on constrained Huffman codes were investigated by de Moura et al. [2000]. A *byte Huffman code*, referred to here as `phc`, is a constrained Huffman code, in which all codewords have a length which is a multiple of 8 bits. A byte Huffman code is capable of more closely matching a given probability distribution, and is minimum-redundancy over all possible byte codes. If a fixed universe assumption is assumed, where all word identifiers are limited to 4-byte integer lengths, the code can be completely described by the 4-tuple

(h_1, h_2, h_3, h_4), where h_x is the number of x-byte codewords in the code.

This method is the most flexible approach to byte-aligned coding, but codeword boundaries are no longer easily detectable in the compressed sequence. In order to regain the ability to easily identify codeword boundaries, de Moura et al. [2000] suggest a *tagged Huffman coding* (`thc`) approach, in which only 7 bits in each byte are used to store information, and 1 bit is used to signify stopper and continuer bytes. The full potential of this method was realized when the codes were used to support compressed pattern matching algorithms. By combining tagged Huffman codes with a modified Boyer-Moore algorithms, searching directly in compressed text can be performed $2-8$ times faster than in uncompressed text [de Moura et al., 2000].

4.1.2 Variable Byte Codes

A more basic approach to byte-aligned coding is to use a radix-128 code, referred to here as `bc`. This method converts a sequence of integers into a uniquely decodable sequence of bytes as follows: for each x, where $x < 128$, is coded as itself in a single byte; otherwise, $(x \text{ div } 128) - 1$ is recursively coded, and then x mod 128 is appended as a single byte. Each output byte contains seven data bits. In order to ensure each codeword is prefix-free, the final byte of each codeword, called a *stopper*, is tagged with a leading zero bit. All other bytes in the codeword, called *continuers* are tagged with a leading one bit. For instance, the following example shows the encoding for some values of x:

$$
\begin{aligned}
1 &\rightarrow \texttt{001} \\
1{,}000 &\rightarrow \texttt{134-104} \\
1{,}000{,}000 &\rightarrow \texttt{188-131-064}
\end{aligned}
$$

Reconstructing the original codeword is simple. For example, to decode the three byte code `188-131-064`:

$$
\begin{aligned}
\texttt{188} \rightarrow (188-127) &= 61 \\
\texttt{131} \rightarrow (61 \times 128) + (131-127) &= 7{,}812 \\
\texttt{064} \rightarrow (7{,}812 \times 128) + 64 &= 1{,}000{,}000
\end{aligned}
$$

Bytes codes have received a considerable attention in the information retrieval domain. Experiments by Scholer et al. [2002] and later by Trotman [2003] show that processing queries using inverted indexes that are byte encoded are two times faster than bit-level Golomb codes, with modest compression loss. Their ease of implementation, as previously described in Section 2.5.3, has also contributed to their popularity.

4.1.3 Dense Byte Codes

Variable length byte codes work well when the underlying probability distribution is monotonic. However, data models do not necessarily generate integer sequences with a monotonic frequency ordering. To compensate for this risk, Brisaboa et al. [2003b] proposed a byte-aligned coding scheme using an alphabet mapping which ranks the symbols by frequency, ensuring that the most frequent items receive short codewords. Their *dense byte code*, referred to as `dbc`, is an extension of standard variable byte code `bc`, and introduces an alphabet ordering to match symbol ranks with symbol probabilities. Dense byte codes require the transmission of a *prelude* for decoding purposes. The prelude ensures that all mapped symbols appear in the message and each symbol is properly represented by its rank. A typical rank-based prelude operates as follows: Let a source alphabet $\Sigma = \{1 \ldots 15\}$. Now consider a text sequence, T, with the set of symbol frequencies

$$T_f = [20, 0, 1, 8, 11, 1, 0, 5, 1, 0, 0, 1, 2, 1, 2];$$

The corresponding ranked alphabet mapping for T would be

$$[1, 5, 4, 8, 13, 15, 3, 6, 9, 12, 14] \rightarrow [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]$$

in which, for example, symbol 5 is the second most frequent element, and therefore gets mapped to the output symbol 2.

This simple ranking mechanism opens up the possibility of new prelude representations because the alphabet ordering is not dependent on exact symbol probabilities. It

can also dramatically reduce the transmission costs for integer sequence models drawn from alphabets of symbol probabilities which are not monotonically decreasing. But care must be used when building the mapping, as large alphabets can quickly counter any gains achieved when requiring a prelude to be transmitted with the compressed sequence. We consider the implications of using a prelude in more detail in Section 4.3.

4.1.4 (S,C)-Dense Codes

Shortly thereafter, Brisaboa et al. [2003a] proposed another modification to variable byte codes. An (S,C)*-dense byte code*, referred to here as `scbc`, is an extension of a dense byte code, in which the partitioning value of 128 is made a variable S such that $1 \leq S \leq 256$. That is, a constraint $S + C = 256$ is added, where S is the number of stoppers, C is the number of continuers, and 256 is the radix. The tag bit which identifies each byte is essentially arithmetically coded now, allowing more of each byte to be available for actual “data” bits. The codewords assigned are still end-tagged, allowing codeword boundaries to be quickly found in the compressed sequence.

Decoding is still simple, and the only additional requirement is to know the partitioning value chosen for the sequence at encoding time. Encoding incurs the additional overhead of determining the best partitioning point. Brisaboa et al. [2003a] propose several methods to select the best partitioning, but a simple brute-force approach works well in practice. By using an array of pre-calculated cumulative frequencies for the mapped alphabet, the cost of any proposed partitioning can be evaluated quickly. Algorithm 3 shows a brute-force method, which can be used to quickly calculate the best partitioning point. First, the frequency array f is converted to a list of cumulative frequencies, which dramatically reduces the cost of testing each of the possible partitioning points. Then, each of the possible partitioning values are evaluated, and the s resulting in the smallest total size is returned.

The (S,C)-dense code works best when the underlying sequence follows a geometric probability distribution. So, it is conceptually equivalent to a byte-level variation of Golomb codes [Golomb, 1966]. The fraction of codespace used for

Algorithm 3 Calculating the partitioning value in `scbc`

INPUT: A list of symbol frequencies f, ordered from largest to smallest, the number of symbols in the list n, and a radix R (typically 256).
OUTPUT: The optimal partitioning value s.

```
 1: curr ← 0
 2: best ← ∞
 3: s ← R
 4: for i ← 1 to n − 1 do
 5:     f[i] ← f[i] + f[i − 1]
 6: end for
 7: for i ← 1 to R do
 8:     curr ← calculate_size(i, f, n, R)
 9:     if curr < best then
10:         best ← curr
11:         s ← i
12:     end if
13: end for
14: return s
```

function *calculate_size*(s, f, n, R)

```
1: set c ← R − s
2: set off1 ← s
3: set off2 ← off1 + s × c
4: set off3 ← off2 + s × c^2
5: set off4 ← off3 + s × c^3
6: set total ← (f[off1] − f[0]) × 1 + (f[off2] − f[off2]) × 2 + (f[off3] − f[off2]) ×
   3 + (f[off4] − f[off3]) × 4
7: return total
```

codewords of any given length is fixed once a value of S is chosen. For example, the number of one byte codewords is S; the number of two byte codewords is $C \times S$; and the number of three byte codewords is C^2S as with all static codes. The quality of compression that arises is then a function of the distance between the probability distribution implied by the set codeword lengths, and the actual probability distribution arising in the message. This leaves open the possibility of other byte coding variants which are better able to cope with other probability distributions.

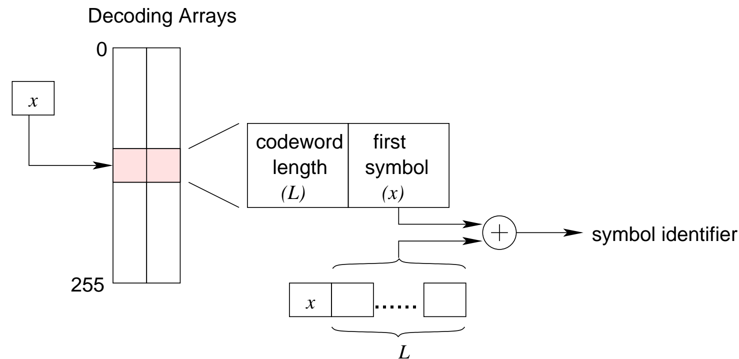


Figure 4.1: Decoding of an `rpbc` code. The first byte x is used to lookup the corresponding codeword length L. Then, the x byte and L trailing bytes are concatenated to produce a rank identifier, which can in turn be used to find the original symbol.

4.2 Restricted Prefix Byte Codes

In order to exactly match a probability distribution using byte-aligned sub-units, this section presents a variation on the the radix-256 Huffman code presented by de Moura et al. [2000]. The new code makes it possible to assign codeword lengths to different codes in a more flexible manner. In practice, any set of symbol identifiers $1 \ldots (n)$ can be partitioned into four contiguous subsets for any typical n-symbol monotonic probability distribution. This suggest any possible set can be defined by the tuple (h_1, h_2, h_3, h_4), where $n = h_1 + h_2 + h_3 + h_4$. A radix-256 variation of the Kraft inequality stipulates that possible tuples which are viable prefix codes with $R = 256$ must satisfy

$$\frac{h_1}{R} + \frac{h_2}{R^2} + \frac{h_3}{R^3} + \frac{h_4}{R^4} \leq 1\,.$$

These observations lead to a new variation of byte code. A *restricted prefix byte code* is a byte Huffman code with a single additional constraint: the first byte must completely define the suffix length. Instead of using a 2-tuple (S, C) as (S, C)-dense codes use, a 4-tuple (v_1, v_2, v_3, v_4) is defined to completely specify allowable code lengths. In the tuple, v_1 specifies the number of one-byte codes, v_2 the number of two-

byte codes, and so on. Also, $v_1 + v_2R + v_3R^2 + v_4R^3 \geq n$, where R is the radix and n is the cardinality of the source alphabet, must be satisfied. This still allows considerable flexibility in assignment of total codespace to different code lengths and ensures that the codes are prefix free. The 4-tuple of `rpbc` can be compared to the (S, C)-dense codes where the tuple is infinite, $(S, CS, C^2S, \ldots)$. The (S, C)-dense codes do not allow more of the codespace to be allocated to two-byte codewords than three-byte codewords, making models which generate flat distributions less compressible.

Figure 4.1 gives a high level overview of how `rpbc` encoded sequences are decoded. The first byte x is used via a lookup table to determine the codeword length, L. Then, L bytes are concatenated with a renormalized x to produce the rank symbol identifier. The rank symbol identifier can in turn be used to lookup the original symbol value via a prelude. Restricted prefix byte codes can be considered a derivative of the $\mathcal{K}$-flat code of Liddell and Moffat [2006]. Both methods use a constrained prefix for each code, such that the total codeword length can be identified unambiguously, without decoding the whole symbol. In a $\mathcal{K}$-flat code, each codeword commences with a k-bit binary prefix which uniquely determines the length of the suffix, for some fixed value k. In `rpbc`, the codeword lengths are uniquely determined by the first byte, and is fully described by a tuple (v_1, v_2, v_3, v_4), where the first byte of any one-byte codeword is in the range $0 \ldots (v_1-1)$; the first byte of any two-byte codeword is in the range $v_1 \ldots (v_1+v_2-1)$; the first byte of any three-byte codeword will lie between $(v_1+v_2) \ldots (v_1+v_2+v_3-1)$; and so on.

Algorithm 4 shows how to decode a restricted prefix byte code. The process begins with initialization of the *suffix* array and the *first* array. The *suffix* array can then quickly be used to determine the codeword length. Once the codeword length is known, the appropriate number of bytes are sequentially concatenated together to produce the symbol rank. Finally, the symbol rank is used to identify the original codeword.

It is worth noting that all of the codes discussed can be generalized to use other radix values (see, for instance Rautio et al. [2002]). Changing the radix would produce codes which are no longer byte aligned, but would work in the same manner. Typically,

Algorithm 4 Decoding an `rpbc` block

INPUT: A block-length m, a radix R (typically 256), and control parameters v_1, v_2, v_3, and v_4, with $v_1 + v_2 + v_3 + v_4 \leq R$.
OUTPUT: The m symbols coded into the message block are available in the array *output_block*

1: *create_tables*(v_1, v_2, v_3, v_4, R)
2: **for** $i \leftarrow 0$ **to** $m - 1$ **do**
3:    assign $b \leftarrow$ *get_byte*$()$ and *offset* $\leftarrow 0$
4:    **for** $i \leftarrow 1$ **to** *suffix*$[b]$ **do**
5:       assign *offset* $\leftarrow$ *offset* $\times R +$ *get_byte*$()$
6:    **end for**
7:    assign *output_block*$[i] \leftarrow$ *first*$[b] +$ *offset*
8: **end for**

function *create_tables*(v_1, v_2, v_3, v_4, R)
1: assign *start* $\leftarrow 0$
2: **for** $i \leftarrow 0$ **to** $v_1 - 1$ **do**
3:    assign *suffix*$[i] \leftarrow 0$ and *first*$[i] \leftarrow$ *start* and *start* $\leftarrow$ *start* $+ 1$
4: **end for**
5: **for** $i \leftarrow v_1$ **to** $v_1 + v_2 - 1$ **do**
6:    assign *suffix*$[i] \leftarrow 1$ and *first*$[i] \leftarrow$ *start* and *start* $\leftarrow$ *start* $+ R$
7: **end for**
8: **for** $i \leftarrow v_1 + v_2$ **to** $v_1 + v_2 + v_3 - 1$ **do**
9:    assign *suffix*$[i] \leftarrow 2$ and *first*$[i] \leftarrow$ *start* and *start* $\leftarrow$ *start* $+ R^2$
10: **end for**
11: **for** $i \leftarrow v_1 + v_2 + v_3$ **to** $v_1 + v_2 + v_3 - v_4 - 1$ **do**
12:    assign *suffix*$[i] \leftarrow 3$ and *first*$[i] \leftarrow$ *start* and *start* $\leftarrow$ *start* $+ R^3$
13: **end for**

higher radix codes are preferred when the alphabet size is large, while smaller radix codes are beneficial for shorter alphabets. For example, there is no benefit to using a byte-aligned code on sequences in which the maximum value is less than 256 (such as ASCII character text).

For instance, consider a small integer sequence derived from a snippet of text using a spaceless words model:

$$1, 3, 4, 5, 3, 4, 1, 6, 5, 7, 5, 6, 1, 7 \,.$$

Table 4.1 shows the codewords and corresponding frequencies for each symbol.

Sym.	Freq.	bc	phc	thc	dbc	scbc	rpbc
4	4	10 01	00	00 10	00	00	00
1	3	00	01	00 11	01	01	01
5	3	11 00	10	01 10 10	10 00	10	10
3	2	10 00	11 00	01 10 11	10 01	11 00	11 00
6	2	11 01	11 01	01 11 10	11 00	11 01	11 01
7	2	10 10 00	11 10	01 11 11	11 01	11 10	11 10
2	0	01	—	—	—	—	—

Table 4.1: Symbol assignments and corresponding radix-4 codewords generated for a 14 symbol sequence.

The radix-4 codeword assignments are also shown for each of the popular byte coding techniques. When using a fully static bc code, the frequency of each symbol is ignored. In a radix-4 version of bc, the values 00 and 01 would be continuers and the values 10 and 11 denote stoppers. So, the symbol 1 is assigned a two-bit representation of 10 as shown in Table 4.1. Note that not all symbols appear and the codeword assignments do not consider the probability distribution of the underlying text.

The dbc, scbc, and rpbc codes all employ a rank based prelude mapping. For instance, in Table 4.1, the symbol 5 is assigned a rank 3 and assigned the codeword 00 10. The (S, C)-dense codes fare a little better. If $R = 4$ then there are three possible (S, C)-dense code arrangements: $(1, 3)$, $(2, 2)$, $(3, 1)$. Note that for this simple example, the $(2, 2)$-arrangement corresponds to dbc. The best arrangement for the example frequency distribution is the $(3, 1)$-arrangement. For rpbc, the optimal code uses a $(1, 1, 1, 2)$-arrangement. Also, note that costs of thc are exaggerated in Table 4.1 as a result of the small radix. For larger radix values, thc generates codewords of reasonable length.

4.2.1 Minimum Cost Codes

There are a few approaches to determining the minimum-cost reduced prefix code. Dynamic programming approaches like those described by Liddell and Moffat [2006] for $\mathcal{K}$-flat codes can be used, and guarantee optimal codes subject to the given

Algorithm 5 Brute-force code calculation in `rpbc`.

INPUT: A set of n frequencies, $f[0 \ldots (n-1)]$, and a radix R, with $n \leq R^4$.
OUTPUT: The four partition sizes v_1, v_2, v_3, and v_4.

```
 1: assign C[0] ← 0
 2: for i ← 0 to n − 1 do
 3:    assign C[i + 1] ← C[i] + f[i]
 4: end for
 5: assign mincost ← partial_sum(0, n) × 4
 6: for i1 ← 0 to R do
 7:    for i2 ← 0 to R − i1 do
 8:       for i3 ← 0 to R − i1 − i2 do
 9:          assign i4 ← ⌈(n − i1 − i2R − i3R^2)/R^3⌉
10:          if i1 + i2 + i3 + i4 ≤ R and cost(i1, i2, i3, i4) < mincost then
11:             assign (v1, v2, v3, v4) ← (i1, i2, i3, i4) and mincost ← cost(i1, i2, i3, i4)
12:          end if
13:          if i1 + i2R + i3R^2 ≥ n then
14:             break
15:          end if
16:       end for
17:       if i1 + i2R ≥ n then
18:          break
19:       end if
20:    end for
21:    if i1 ≥ n then
22:       break
23:    end if
24: end for
```

function *partial_sum*(lo, hi):

```
1: if lo > n then
2:    assign lo ← n
3: end if
4: if hi > n then
5:    assign hi ← n
6: end if
7: return C[hi] − C[lo]
```

function *cost*(i_1, i_2, i_3, i_4)

```
1: return partial_sum(0, i1) × 1 +
      partial_sum(i1, i1 + i2R) × 2 +
      partial_sum(i1 + i2R, i1 + i2R + i3R^2) × 3 +
      partial_sum(i1 + i2R + i3R^2, i1 + i2R + i3R^2 + i4R^3) × 4
```

constant. However, the space requirements for high radix codes are non-trivial. Various approximate approaches can also be used but compression effectiveness is degraded when using simple greedy heuristics.

A simple brute force calculation works well in practice. Algorithm 5 describes a straightforward approach which evaluates all legal combinations of (v_1, v_2, v_3, v_4) and chooses the least cost arrangement. For any plausible alphabet size, the nested loops of the algorithm execute in a few hundredths of a second, and have no additional space overhead. The arrangement selected is optimal and little practical gain in efficiency is possible through the use of a dynamic programming approach.

4.3 Prelude Considerations

The `bc` byte codes are popular in many applications because of their simplicity and effectiveness. By virtue of being fully static, no additional information is required to transmit the sequence of encoded symbols. That is, the `bc` method is *on-line*, and no input buffering is necessary. However, dense coding methods are *off-line* mechanisms – they require the input message to be buffered into *message blocks* before processing is started, or two passes be made over the entire text. In this thesis, we opt for block-based semi-static methods over traditional two pass methods, as there are several benefits to block-based approaches. In particular, these methods allow unlimited stream lengths to be accommodated, and they exploit localized clustering of symbols. These and other benefits are discussed in detail by Moffat and Turpin [2002]. Block-based semi-static codes also require a *prelude* and a couple of scalar values, such as maximum allowable value u, and partitioning values to be transmitted to the decoder in order to decode the each block.

Figure 4.2 provides an example of semi-static message blocking, coupled with a spaceless-words model for natural language text. In this example, a text fragment from the popular children's book "Fox in Socks" [Seuss, 1965] is partitioned into an alternating sequence of words and non-words, as previously discussed in Section 2.4.3. The intermediate sequence of integers can then be partitioned further into message

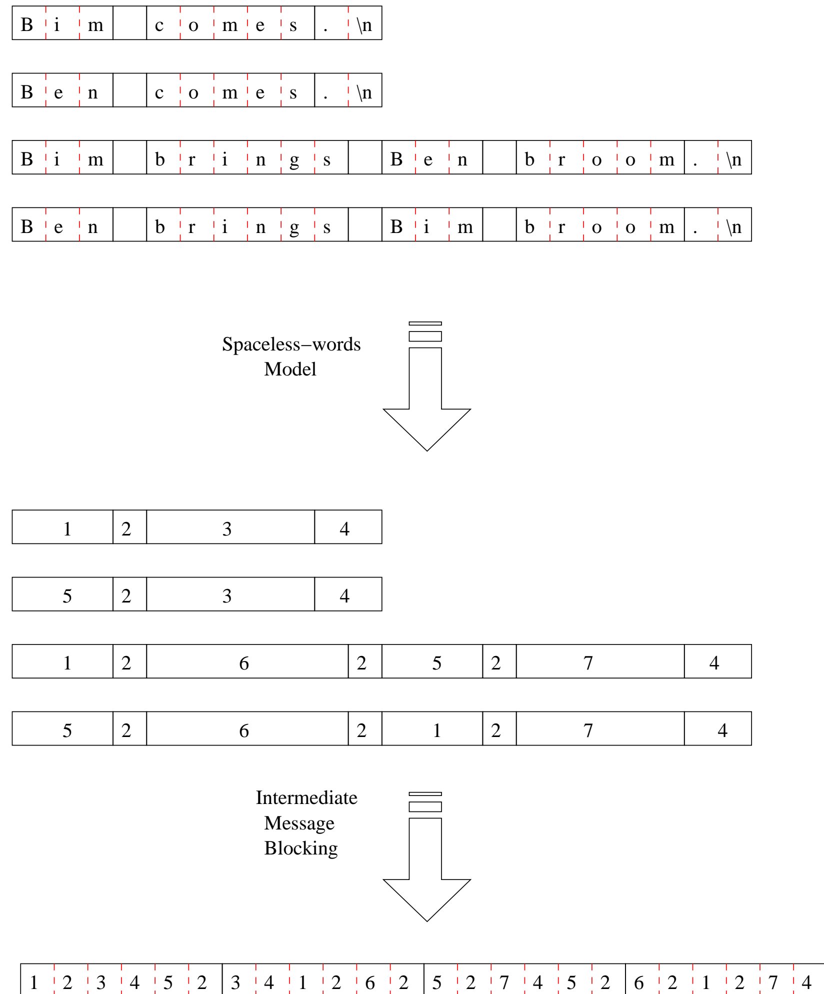


Figure 4.2: An example of spaceless-words modeling being combined with message block partitioning of integer intermediates. First, a spaceless-words model is applied to a text fragment from "Fox in Socks". Next, the resulting stream of integers is partitioned into 4 blocks of size $m = 6$. Each block can then be coded separately. Note that the cost of transmitting the *vocabulary* produced by the model, which maps English text to integers identifiers, must be accounted for, and is independent of the message blocking scheme. Additionally, each block requires the localized integer alphabet ranking, referred to as a *prelude*, to also be transmitted.

blocks. The message block size, arbitrarily chosen here to be $m = 6$, can be tuned to exploit localized symbol clustering. Moffat and Turpin [2002] provide a comprehensive analysis of the of block-based semi-static coding, and the reader is encouraged to consult this text for a more detailed discussion. Two key points are emphasized here. Firstly, the vocabulary mapping that results from the modeling phase must be transmitted explicitly or implicitly to the decoder. An example of implicit transmission is discussed in Section 2.4.3, and further by Moffat and Isal [2005]. An example of explicit transmission is discussed by Brisaboa et al. [2004], where a clear distinction between modeling and coding is not considered. It is important to realize that the transmission of the vocabulary is *independent* of block-based coding. Secondly, block-based coding also requires a prelude to be transmitted with each block to specify localized statistical information, in addition to the vocabulary. We now consider several different approaches to efficient prelude representation for block-based semi-static coding.

4.3.1 Permutation Preludes

The simplest way to represent the prelude is simply to transmit a permutation of the alphabet [Brisaboa et al., 2003a,b]. A *permutation approach* is the traditional method, where the complete alphabet is transmitted using a static binary code in a decreasing frequency ordering. The length of each static binary code is easily calculated as $\log u$, where u is the value of the largest symbol. The total cost of the prelude is roughly $\mathcal{O}(n \log u)$ bits and the overhead is $\mathcal{O}(n \log u/m)$ bits per message symbol. If the source alphabet is small, the cost is minimal. For more general applications, the cost can be non-trivial. For example, consider a character based prelude with $u = 256$ and $n \approx 100$. The overhead is less than 0.001 bits per symbol for a block of $m = 2^{20}$ symbols. However, for the same block size and $n \approx 10^5$ and $u \approx 10^6$ the overhead jumps to 1.9 bits per symbol.

4.3.2 Bitvector Preludes

An exact permutation is not actually necessary. The only information required to reconstruct the code is a list of symbols which appear in the block, and the length of the codeword to be assigned to each. So, we need to encode the n element subset of $0 \ldots u$ which appear in each block along with its corresponding length. A *bitvector approach* represents the subset of items from $0 \ldots u$ as a string of bits, where a 1 in the k th value means the symbol appears, and 0 means it does not appear. If the symbol appears, an additional two bits can be used to encode the symbol length when the maximum codeword length is 4. If we again consider the dense alphabet example where $n \approx 10^5$ and $u \approx 10^6$, the space required is $u + 2n \approx 1.2 \times 10^6$, or 1.14 bits per symbol of overhead on a message block of $m = 2^{20}$ symbols. This represents a significant gain from a relatively straightforward change.

4.3.3 Difference Gap Preludes

If the subset of items to be transmitted is sparse, the prelude can be stored more compactly than an unadorned bitvector allows. A *gap approach* is another alternative which eliminates bit operations, and attempts to exploit clustering in alphabet densities. Only four possible codeword lengths are necessary for all practical purposes, so the alphabet can be partitioned into four subsets. Each individual subset is then sorted and encoded as a sequence of d-gaps, in a manner previously discussed for inverted indexes in Section 3.2.1. Since the emphasis is easily decodable streams, the d-gaps are encoded with the standard byte coder `bc`. A similar approach to prelude description, in which a sub-alphabet is selected, and then the codeword lengths described, is presented by Turpin and Moffat [2000]. In their bit-based Huffman coding approach, the alphabet is chosen via interpolative coding Moffat and Stuiver [2000].

To estimate the costs of d-gap prelude arrangement, we suppose that all but a small minority of the gaps are less than 127, the largest value which can be encoded in a single byte. This is a reasonable assumption assuming that the sub-alphabet density does not drop below around 1%. Using this arrangement, the prelude costs approximately $8n$

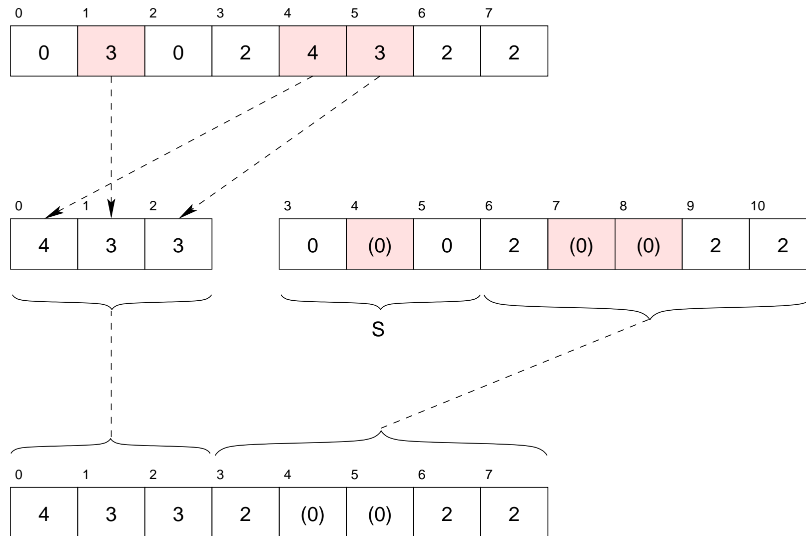


Figure 4.3: Construction of a semi-dense prelude for the sequence $T = 1, 3, 4, 5, 3, 4, 1, 6, 5, 7, 5, 6, 1, 7$. Here $t = 3$, resulting in the three most frequent items being assigned the smallest codewords. All zero frequency items up to the first item present (6) can be dropped since they never appear, resulting in the shift $s = 3$.

bits which corresponds to 0.76 bits per symbol of overhead when $n \approx 10^5$ and a block size of $m = 2^{20}$ is chosen.

4.3.4 Semi-Dense Preludes

Is it possible to reduce the cost of the prelude even more? We could use a nibble code, where each subunit is 4 bits instead of 8 bits, which could, in the right situation, halve the cost on average. A more promising approach is to limit the size of the prelude. Chen et al. [2003] propose a self-describing bucketed approach where a sparse alphabet is Huffman coded as binary offsets within a bucket, based on the aggregate symbols in the bucket. In contrast, it is also possible to take care of a subset of the most frequent items and the remaining symbols are encoded statically, reducing the per symbol overhead of prelude transmission on average. This approach is motivated by the fact that maintaining a prelude for rare items can be more expensive than simply assigning them a default sparse code. In a *semi-dense approach*, a subset of

"interesting" symbols (because of their prevalence) are assigned a second codeword, in addition to their normal "dense" codeword.

Figure 4.3 provides an example of how the semi-dense prelude mapping is accomplished. First, a threshold t is chosen, and the t most frequent items are identified and mapped into a sub-alphabet in decreasing frequency order, to ensure they receive short codewords. Then, the first non-zero element is identified in the sparse subset, resulting in a shift of S items which can be dropped from the final codeword assignment phase since they never appear. All other "zero" items must be explicitly encoded as part of the sparse subset. Finally, codeword assignment proceeds for all "dense" items and "sparse" items remaining in the set.

Algorithm 6 details the dual codeword assignment approach employed in a semi-dense prelude. Essentially, the codespace is separated into two parts. The partitioning value t determines the maximum size of the ranking map. A minimum-redundancy restricted prefix code is calculated for the full codespace which has been shifted by t positions. The highest frequency codes are assigned a default value as well as a dense value, which are assigned to the shortest codewords. The exact value of t can be adjusted such that compression effectiveness can be traded against prelude size or it can be set statically. There are several different heuristic approaches which can be used to select the value t.

- The cutoff could be a fixed value, such as $1{,}000$.
- The cutoff could be all symbols above a particular frequency threshold.
- The cutoff could be all symbols in the one or two byte range.

In practice, the last method is the most effective, and the method of choice in the experimental results presented here.

Algorithm 6 Calculating a Semi-Dense Prelude

INPUT: An integer u, and an unsorted array of symbol frequency counts, with $c[s]$ recording the frequency of s in the message block, $0 \leq s \leq u$; together with a threshold t.

OUTPUT: The coded message block.

1: assign $n \leftarrow 0$
2: **for** $s \leftarrow 0$ **to** u **do**
3:     assign $f[t+s].sym \leftarrow s$ and $f[t+s].freq \leftarrow c[s]$
4: **end for**
5: identify the t symbols with the largest *freq* components in $f[t \ldots (t+u)]$, and copy them and their corresponding frequencies into $f[0 \ldots (t-1)]$
6: **for** $s \leftarrow 0$ **to** $t-1$ **do**
7:     assign $f[t+f[s].sym].freq \leftarrow 0$
8: **end for**
9: assign *shift* $\leftarrow 0$
10: **while** $f[t + shift] = 0$ **do**
11:     assign *shift* $\leftarrow$ *shift* $+ 1$
12: **end while**
13: **for** $s \leftarrow t + shift$ **to** u **do**
14:     assign $f[s - shift] \leftarrow f[s]$
15: **end for**
16: use Algorithm 5 to compute v_1, v_2, v_3, and v_4 using the $t + u + 1 - shift$ elements now in $f[i].freq$
17: sort array $f[0 \ldots (t-1)]$ into increasing order of the *sym* component, keeping track of the corresponding codeword lengths as elements are exchanged
18: transmit v_1, v_2, v_3, and v_4 and the first t values $f[0 \ldots (t-1)].sym$ as a prelude, together with the matching codeword lengths for those t symbols
19: sort array $f[0 \ldots (t-1)]$ into increasing order of codeword length, with ties broken using the *sym* component
20: **for** each symbol s in the message block **do**
21:     **if** $\exists x < t : f[x].sym = s$ **then**
22:         code s as the integer x, using v_1, v_2, v_3, and v_4
23:     **else**
24:         code s as the integer $t + s - shift$, using v_1, v_2, v_3, and v_4
25:     **end if**
26: **end for**

File name	Total symbols	Maximum value	n/u ($m = 2^{20}$)	Self-information (bits/sym)
WSJ-WRD	58,421,983	222,577	22.5%	10.58
WSJ-IDX	41,389,467	173,252	10.4%	6.76
WSJ-BWT	58,421,996	222,578	20.8%	7.61
WSJ-REP	19,254,349	320,016	75.3%	17.63
ZIPFA	250,000,000	1,000,000	99.9%	10.89
ZIPFB	125,000,000	500,000	36.4%	4.92
GEOM	250,000,000	221,793	48.2%	14.73
UNIF	250,000,000	999,997	50.0%	18.93

Table 4.2: Statistical properties for all experimental collections used for benchmarking. The total number of symbols in the sequence, and the universe size, u, is shown in the first two columns. The third column shows the average sub-alphabet density for a block containing $m = 2^{20}$ symbols, where n is the number of distinct items in the block. The final column shows the zero-order self-information of the file.

4.4 Experiments

In order to test the algorithms developed, several large data sets were chosen for empirical study, with different statistical properties. Four different modeling intermediates of a text segment taken from the TREC data collection, as well as four synthetic sequence files using well-defined probability distributions, make up the test collection. In general, the modeling costs to generate the intermediates are not considered, since we wish to clearly delineate the *coding* costs from the complete compression process.

4.4.1 Experimental Setup

Table 4.2 shows various statistical properties for the compression benchmarking data collection. The first four files are derivatives of a 267 MB segment of SGML-tagged text, taken from the Wall Street Journal portion of the TREC data collection. The WSJ-WRD file was constructed using a spaceless words model similar to the one described in Section 2.4.3. The WSJ-IDX file represents a concatenated list of d-gaps of document identifiers for each vocabulary term, such as is typically found in an inverted index. The WSJ-BWT file represents an integer sequence generated from a word-parsed sequence which has been BWT'ed and MTF'ed. The WSJ-REP file represents phrase numbers

File	Method				
	shuff	`bc`	`dbc`	`scbc`	`rpbc`
WSJ-WRD	10.44	16.29	12.13	11.88	11.76
WSJ-IDX	6.66	9.35	9.28	9.00	8.99
WSJ-BWT	7.50	10.37	10.32	10.17	10.09
WSJ-REP	16.57	22.97	19.91	19.90	18.27
ZIPFA	10.59	23.92	12.81	12.70	12.57
ZIPFB	4.95	23.92	8.65	8.56	8.54
GEOM	14.71	23.41	17.38	17.38	15.95
UNIF	18.57	23.87	23.25	23.25	21.71

Table 4.3: Average codeword cost for different byte coding methods. Each input file is processed as a sequence of message blocks of $m = 2^{20}$ symbols, except at the end. Values listed are in terms of bits per source symbol, excluding any necessary prelude components. Only the column headed `bc` represents actual compression rates, since the simple byte coding approach is the only one that does not require a prelude.

from a word-based recursive byte-pair parser.

The remaining four data files (ZIPFA, ZIPFB, GEOM, and UNIF) were constructed to complement the above model-based intermediate files, which were derived from a "real" natural language text collection. The first two artificial sequences represent a Zipfian distribution where $w_f = 1/r^{\alpha}$, as previously discussed in Section 2.4.3. The file ZIPFA was generated using a $\alpha = 1.1$, $n = 500{,}000$, and $u = 1{,}000{,}000$. The file ZIPFB was generated using a $\alpha = 1.46$, $n = 220{,}000$, and $u = 500{,}000$. The file GEOM was generated using a probability $p = 0.0001$ where $p(x) = p(1-p)^x$, and UNIF represents an i.i.d. discrete uniform distribution with $n = 500{,}000$ and $u = 1{,}000{,}000$. After each of the artificial files was generated using a Zipfian, Geometric or Uniform distribution, the alphabet mappings were randomly shuffled such that the symbol frequencies were no longer monotonically decreasing. This ensures the various prelude method evaluation is unbiased. If the modeling stage produces a sequence of integers drawn from a symbol alphabet which is already monotonically decreasing, a prelude is not strictly necessary.

4.4.2 Efficiency and Effectiveness

Table 4.3 shows the base compression effectiveness for all the byte-code methods, when each file is processed in blocks of $m = 2^{20}$ symbols. No prelude costs are included, meaning that only the `bc` column reflects total compression costs. In addition to the various byte coding methods, a word-based Huffman coder, referred to here as `shuff`, is included as an efficient bit-aligned baseline [Turpin and Moffat, 2000]. The `shuff` column shows the compression effectiveness attainable using a bit-aligned, minimum-redundancy code, minus prelude costs.

For files which follow a Zipfian distribution, such as WSJ-WRD, ZIPFA, or ZIPFB, all of the methods benefit from a rank-based prelude mapping. There is a small, but consistent, improvement when moving from `dbc` to `scbc` to `rpbc`, based on increasing flexibility in codespace assignment. The gap in effectiveness widens between `scbc` and `rpbc` as the sub-alphabet density increases and the probability distribution flattens, as in the WSJ-REP and UNIF files. In contrast, the difference in effectiveness decreases as the probability distribution becomes more skewed, as in the WSJ-BWT and WSJ-IDX files. The `rpbc` method consistently performs better than all of the other byte-coding variants, but is less effective than a bit-aligned, minimum-redundancy code. As expected, the performance improvement is more pronounced in the files which have a flat distribution of symbol probabilities, such as WSJ-REP and UNIF.

4.4.3 Prelude and Block-Size Impact

Table 4.4 shows the prelude costs for the four different byte-aligned prelude representations and the bit-aligned prelude of `shuff`. The `shuff` method uses a binary interpolative code to compress the sub-alphabet [Moffat and Stuiver, 2000]. Binary interpolative codes are highly effective, but computationally intensive, relative to the byte coding methods described. When the sub-alphabet mapping is dense, as in ZIPFA and GEOM, the bitvector approach is the most effective representation for transmitting the complete mapping. As expected, the sparse sub-alphabet mappings in ZIPFB and WSJ-IDX benefit from gap-based approaches. In contrast, the semi-dense method

File	Prelude Representation				
	shuff	permutation	bitvector	gaps	semi-dense
WSJ-WRD	0.18	0.59	0.22	0.27	0.13+0.01
WSJ-IDX	0.09	0.31	0.20	0.14	0.08+0.00
WSJ-BWT	0.15	0.65	0.25	0.29	0.15+0.01
WSJ-REP	0.79	4.44	0.78	1.87	0.49+0.02
ZIPFA	0.83	2.71	1.23	1.36	0.21+0.06
ZIPFB	0.13	0.29	0.51	0.15	0.14+0.00
GEOM	0.38	0.90	0.31	0.50	0.45+0.00
UNIF	1.89	8.37	1.79	4.19	0.53+0.08

Table 4.4: Average prelude cost for four different representations. In all cases the input file is processed as a sequence of message blocks of $m = 2^{20}$ symbols, except for the last block in the file. Values listed represent the total cost of all of the block preludes, expressed in terms of bits per source symbol. In the column headed "semi-dense", the use of a partial prelude causes an increase in the cost of the message, the amount of which is shown (for the `rpbc` method) as a secondary component.

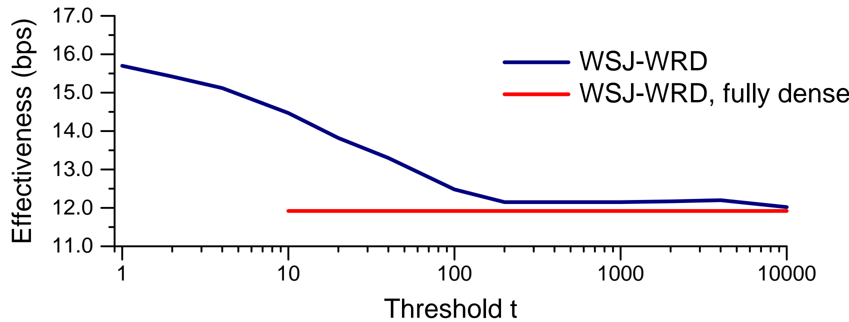


Figure 4.4: The impact on compression effectiveness of varying prelude thresholds, t, using the WSJ-WRD file, and the `rpbc` method.

represents a mapping where only the one and two byte codes are kept in the prelude, and all other codewords are assigned as is. This method provides a significant improvement in coding effectiveness, even when adding the small compression loss associated with the unmapped codewords. Surprisingly, the semi-dense prelude method is often more effective than the bit-based prelude representations of `shuff` and bitvectors, despite the fact that it uses a byte-aligned code to compress the sub-alphabet.

File	shuff	bc	dbc	scbc	rpbc	
		(none)	dense	dense	dense	semi-dense
WSJ-WRD	36	59	24	24	36	43
WSJ-IDX	44	68	30	30	47	59
WSJ-BWT	38	60	26	26	39	50
WSJ-REP	6	49	9	9	12	30
ZIPFA	11	20	12	12	15	21
ZIPFB	45	54	30	31	51	51
GEOM	20	20	16	15	22	23
UNIF	3	20	5	5	6	15

Table 4.5: Decoding speed on a 2.8 Ghz Intel Xeon with 2 GB of RAM, in millions of symbols per second, for complete compressed messages including a prelude in each message block, and with blocks of length $m = 2^{20}$. The bc method has no prelude requirement.

Figure 4.4 shows the impact on compression effectiveness of different cutoff thresholds t for the WSJ-WRD file. There is a big gain for every additional symbol added to the sub-alphabet reordering for $t < 100$; thereafter the return in effectiveness drops off beyond $t > 200$. The surprising observation here is that only a small number of items need to be remapped. All other items can receive a sparse code with relatively little impact on overall compression effectiveness.

Table 4.5 shows the decoding speed for all methods and test collections. The basic bc method is the most efficient, but suffers from poor compression effectiveness. The shuff method is surprisingly efficient, despite being bit-aligned. However, when comparing relative performance between shuff and rpbc on the files WSJ-WRD and ZIPFA, the performance gap increases between rpbc and shuff, as the total amount of text processed increases. When a prelude is added, rpbc consistently outperforms both dbc and scbc. The use of a semi-dense prelude further enhances the efficiency of rpbc, by reducing the size of the symbol rank lookup table. When a fully dense prelude is used, and the encoded sequence is drawn from a large alphabet, cache misses quickly have a negative impact on the overall decoding efficiency. So, using a semi-dense prelude in conjunction with any of the byte coding methods can greatly enhance decoding efficiency.

File and Prelude Method	Block Size				
	$m = 2^{20}$	$m = 2^{19}$	$m = 2^{18}$	$m = 2^{17}$	$m = 2^{16}$
WSJ-WRD	11.76	11.71	11.59	11.48	11.38
permutation	0.59	0.86	1.23	1.71	2.32
bitvector	**0.22**	**0.39**	0.73	1.37	2.60
full-gaps	0.27	0.49	**0.70**	**0.99**	**1.34**
semi-dense	0.15	0.28	0.64	0.91	1.23
WSJ-IDX	8.99	8.96	8.92	8.90	8.87
permutation	0.31	0.41	0.54	0.70	0.88
bitvector	0.20	0.38	0.72	1.39	2.57
full-gaps	**0.14**	**0.23**	**0.31**	**0.40**	**0.51**
semi-dense	0.08	0.20	0.28	0.38	0.53
WSJ-BWT	10.09	10.04	9.94	9.87	9.82
permutation	0.65	0.90	1.20	1.57	1.99
bitvector	**0.25**	**0.45**	0.82	1.54	2.90
full-gaps	0.29	0.50	**0.68**	**0.89**	**1.13**
semi-dense	0.16	0.40	0.61	0.80	1.04
WSJ-REP	18.27	17.91	17.40	16.25	15.61
permutation	4.44	6.22	8.07	9.90	11.67
bitvector	**0.78**	**1.26**	**2.06**	**3.44**	**6.02**
full-gaps	1.87	3.27	4.25	5.21	6.14
semi-dense	0.51	1.12	2.23	4.46	5.54
ZIPFA	12.57	12.48	12.16	11.62	11.43
permutation	2.71	3.29	3.90	4.56	5.27
bitvector	**1.23**	2.24	4.21	8.09	15.79
full-gaps	1.36	**1.64**	**1.95**	**2.32**	**2.86**
semi-dense	0.27	0.47	1.59	2.10	2.61
ZIPFB	8.54	8.52	8.50	8.49	8.48
permutation	0.29	0.37	0.48	0.61	0.77
bitvector	0.51	1.00	1.96	3.88	7.72
full-gaps	**0.15**	**0.21**	**0.29**	**0.41**	**0.57**
semi-dense	0.14	0.19	0.27	0.39	0.55
GEOM	15.95	15.92	15.89	15.84	15.78
permutation	0.90	1.56	2.64	4.33	6.76
bitvector	**0.31**	**0.60**	**1.14**	**2.18**	4.14
full-gaps	0.50	0.87	1.47	2.40	**3.75**
semi-dense	0.45	0.78	1.34	2.17	3.40
UNIF	21.71	21.15	20.26	19.07	15.99
permutation	8.37	12.39	15.57	17.60	18.75
bitvector	**1.79**	**3.15**	**5.37**	9.39	17.14
full-gaps	4.19	6.20	7.78	**8.80**	**9.38**
semi-dense	0.54	1.06	2.10	4.52	8.45

Table 4.6: Trade-offs in overall compression effectiveness when varying the block-size in an `rpbc` code. Small blocks typically exploit clustering better, resulting in improved effectiveness per symbol for the body, but gains are quickly offset by increased prelude costs.

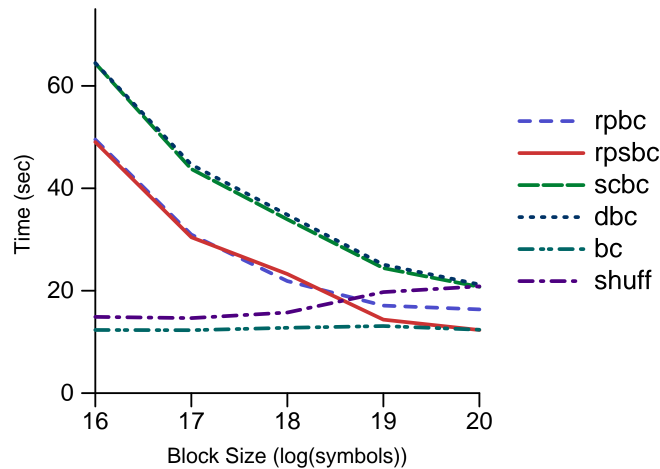


Figure 4.5: The impact of blocksize on decoding efficiency for all byte coding methods on the file ZIPFA. The block size shown is $\log_2$ of the number of symbols for clarity.

Table 4.6 shows the trade-off in effectiveness for differing block-sizes, for an `rpbc` code. As the block size decreases, the cost per symbol also decreases, but the cost to store the prelude increases. Larger blocks perform better on average, as the alphabet density grows. Using a permutation prelude is never a recommended, and smaller block sizes amplify the poor performance of this method. When blocks are large, and the sub-alphabet is dense, the bitvector method is the best fully dense choice. However, smaller blocksizes better exploit clustering, thus reducing the sub-alphabet density, which makes the d-gap methods more attractive. The most effective fully-dense prelude method shown in boldface for each file in order to highlight clustering effect of decreasing block-size. Once again, using a semi-dense prelude consistently results in better overall effectiveness, for any block-size and file combination. Moffat and Turpin [2002] provide a detailed discussion of clustering effects when block size is taken into account.

Figure 4.5 shows the impact of block-size on raw decoding efficiency. Block-size has little impact on the `bc` method, as no prelude information must be evaluated. The binary interpolative coding method employed in the `shuff` method results in a modest loss in efficiency as block size increases. In contrast, the gap-based prelude method

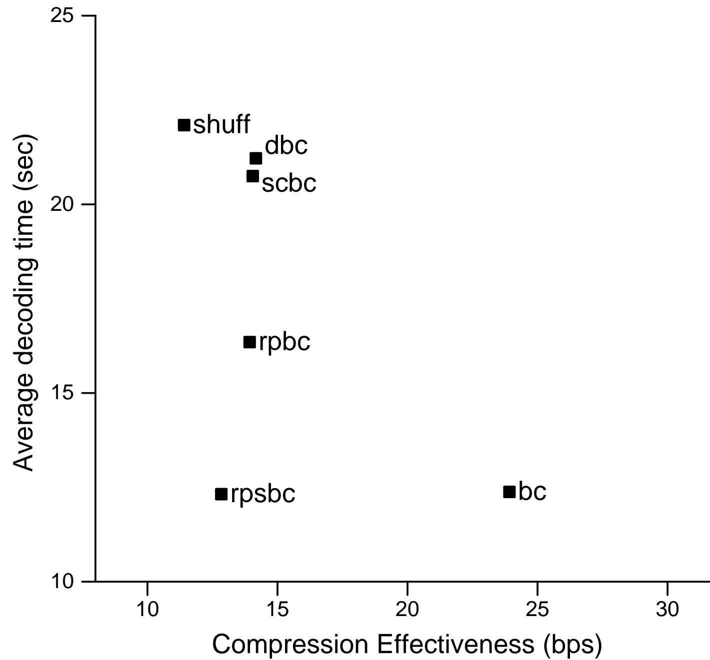


Figure 4.6: Comparison of effectiveness versus efficiency for all byte coding methods for the file ZIPFA. The baselines are `shuff` for effectiveness and `bc` for efficiency. The rpsbc data point represents the `rpbc` method in addition to a semi-dense prelude.

used in `rpbc`, `scbc`, and `dbc` performs poorly as block-size decreases as a consequence of the increase in the number of times the symbol mapping must be built. As the ratio between u and block-size increases, the impact on efficiency becomes more dramatic. Large block-sizes used in conjunction with a semi-dense prelude, as shown by the rpsbc method, results in decompression performance as efficient as that of `bc`.

Figure 4.6 shows effectiveness versus efficiency for byte coding variants on the 1 GB Zipfian test file. The bitwise Huffman coder, `shuff`, provides the best compression effectiveness. In contrast, `bc` uses no prelude and has excellent decoding efficiency. The dense methods `dbc` and `scbc` increase the effectiveness at the cost of efficiency, with `scbc` being modestly better in both regards. The `rpbc` method, using a fully-dense prelude, recoups some of the efficiency loss and increases the compression effectiveness slightly. The rpsbc method, using a semi-dense prelude, greatly enhances

efficiency and improves effectiveness. In fact, the rpsbc actually outperforms the bc method as a result of the reduced quantity of data which must be processed at decoding time.

4.5 Summary

Restricted prefix byte codes provide a more flexible approach to modeling diverse probability distributions without losing the attractive properties of current byte coding approaches. In addition, using byte codes with a prelude consistently improves compression effectiveness, but care must be taken with the representation. Semi-dense prelude representations provide a compelling alternative to traditional prelude representations. Regardless of probability distributions or alphabet size, semi-dense preludes outperform their fully-dense counterparts in every facet. In fact, the use of semi-dense preludes result in improved efficiency and effectiveness for all of the empirical studies conducted. In combination, these two improvements provide significant enhancements to decoding efficiency, and give improved overall compression effectiveness.

Chapter 5

Compact Sequences

Byte codes have unique properties which make them more attractive for various applications than traditional bit-aligned coding algorithms. This is particularly evident when considering the problem of searching and seeking in compressed text. The vast majority of entropy-based codes, such as Huffman or arithmetic codes, require compressed data to be evaluated in a sequential fashion. This limitation can greatly reduce their usefulness in applications in which only localized decoding is required. In contrast, byte codes typically contain codeword boundary markers which allow non-sequential searching algorithms to be applied directly to a compressed string with little or no modification.

The compressed pattern matching problem is defined as: given a *pattern* $\mathcal{P}$, a *text* $\mathcal{T}$, and a corresponding *compressed text* $\mathcal{Z}$, generated by a compression algorithm, find all occurrences of $\mathcal{P}$ in $\mathcal{T}$, that is, determine the set $\{|x| \mid T = x\mathcal{P}y\}$, using only $\mathcal{P}$ and $\mathcal{Z}$. The naive solution is simply to decompress the text $\mathcal{Z}$ before applying a pattern matching algorithm. In fact, this technique is employed by the standard UNIX program, zgrep and serves as a baseline which was difficult to beat in practice. However, the gap between CPU performance and I/O seek times continues to widen, allowing more complex data manipulation at runtime, and offer continued encouragement to the exploration of compressed searching techniques.

Much of the previous work in compressed pattern matching in byte-aligned codes has focused on the use of variants of the Boyer-Moore approach [Fariña, 2005]. In fact, any pattern matching algorithm can be employed, but few studies have considered how the extended alphabets which are a byproduct of word-based compression models affect the performance of the search algorithm. In this chapter, we explore the role of byte-aligned coding methods in designing efficient compressed pattern matching algorithms. We also consider the performance impact of removing redundancy from the search text, and the role of caching effects when using large alphabets. Finally, we show that using byte-aligned compression in conjunction with standard pattern matching algorithms allows text to be searched faster after compression than in raw, unprocessed form.

5.1 Previous Work

Early attempts to support pattern matching directly in compressed text met with limited success. Most were LZ-based and asymptotically efficient, but difficult to implement in practice [Amir et al., 1996, Kida et al., 1998, Rytter, 1999]. Some progress was made when simple character-based packing techniques were combined with standard pattern matching algorithms [Manber, 1997]. However, remarkable efficiency gains were not reported until de Moura et al. [2000] applied word-based modeling to the process, and then coupled the word-based model with byte-aligned codes. Word-based modeling techniques have been used with great success in information retrieval systems for years, so it should come as no surprise that it can improve compression effectiveness and dramatically simplify simple pattern matching approaches. In this section, we focus on byte-aligned compression approaches, as they are the most efficient algorithms in practice.

5.1.1 Byte-Pair Encoding

One of the earliest approaches to byte-aligned compressed pattern matching was proposed by Manber [1997]. Manber describes a method for compressed pattern

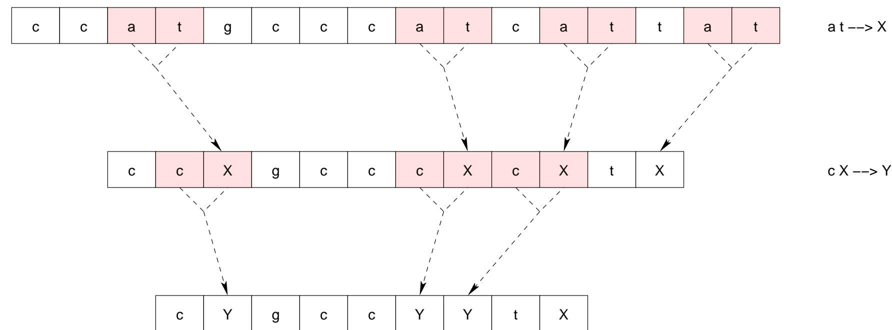


Figure 5.1: Codeword assignment proceeds via a greedy substitution heuristic in the byte pair encoding process.

matching which builds on the byte-pair encoding algorithm (`bpe`). The basic premise of `bpe` has been reinvented several times over the last thirty years [Snyderman and Hunt, 1970, Rubin, 1976, Jewell, 1976, Gage, 1994]. The compression effectiveness of Manber's `bpe` method relies on the fact that text files rarely use all 256 valid byte values. For instance, ASCII text uses 128 of the possible 256 codewords in the worst case, and can use as few as half that many, depending on the text. This allows the text size to be reduced by replacing common bigrams with unused byte values. Figure 5.1 shows a simple application of `bpe`. First, the most common bigram is identified and replaced with an unused symbol, $\mathsf{at} \mapsto \mathsf{X}$. The text is then evaluated again, resulting the most common remaining bigram substitution, $\mathsf{cX} \mapsto \mathsf{Y}$. The process continues until all 256 valid byte values are in use, resulting in algorithm termination. Later `bpe` methods, such as the one proposed by Larsson and Moffat [2000], allow alphabets larger than 256 and terminate the recursive coding when a fixed frequency threshold is reached.

The key insight of Manber was to devise a greedy substitution approach where recursive bigram mappings remain prefix-free. In other words, the same symbol cannot be both a prefix and a suffix in successive productions. This ensures that false matches can not occur when using a standard compressed pattern matching algorithm, such as BMH or KMP. In addition, Manber suggests breaking the raw text into blocks before compression, which ensures localized clustering is exploited.

Searching in bpe compressed text is the first published example of a technique which is significantly faster than the "decompress then search" baseline of Amir and Benson [1992]. Manber's experiments show that the compressed text can be searched $20 - 30\%$ faster than the corresponding uncompressed text. However, the algorithm does not give competitive compression effectiveness compared to word-based byte-coding methods, typically on the order of a 30% reduction. But from a historical point of view, the use of bytes as the smallest subunit proved to be a critical insight, and virtually all practical algorithms reported in the literature use variations of byte-aligned codes.

5.1.2 Tagged Huffman Codes

Building on lessons learned in information retrieval, de Moura et al. [2000] take a word-based approach to compressed pattern matching, and at the same time investigate variations of word-based Huffman codes which facilitate compressed pattern matching. Word-based Huffman codes offer excellent compression and benefit from years of efficiency tuning [Moffat, 1989, Moffat and Turpin, 1998, Turpin and Moffat, 2000]. The missing component was a modification which allows well-understood pattern matching algorithms to be applied directly to the compressed representation.

In order to facilitate direct pattern matching, de Moura et al. propose two byte-aligned Huffman codes. The first method, referred to as a *plain Huffman code* (phc), is based on a Huffman tree which has degree 256. The second method, referred to as a *tagged Huffman code* (thc), is based on a Huffman tree which has degree 128. The phc method is minimum redundancy across all possible byte-aligned combinations, but suffers from one serious drawback: codeword boundaries cannot be discovered without decoding the text. This means it is impossible to search directly in the compressed text, because one codeword can be a suffix of another codeword. Recall the codeword assignments for phc in Table 4.1 for the symbol 1 (01), and the symbol 6 (11 01). The codeword for the symbol 1 is a *suffix* of symbol 6. A search for the codeword 01 will match all instances of symbol 1 and symbol 6. In this example code, the ambiguity can

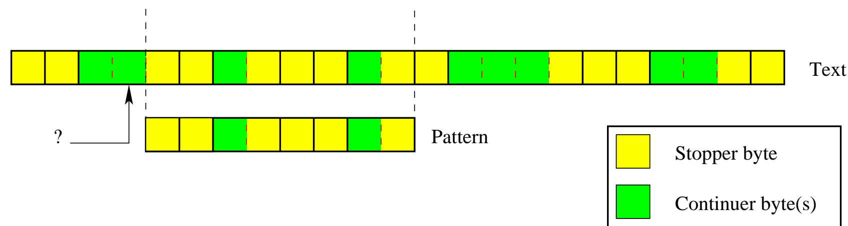


Figure 5.2: False match filtering in stopper-continuer codes requires an additional compare when a possible match is found.

be resolved by looking at the preceding "byte" to see if it contains a `11`, but in a larger code, direct searching is impossible, since codeword boundaries are not identifiable.

The codeword boundary limitation led directly to the `thc` variation. The `thc` method essentially uses 7 bits of each byte to encode the data and 1 bit as a codeword marker. This marker bit is used to uniquely identify the byte at the beginning of each codeword. The compression loss in `thc` is approximately 11% over the `phc` coding method, which typically reduces the overall text size by up to 70%. The real strength of `thc` is that it requires no modifications to the searching algorithm, resulting in searches which are often twice as fast as the corresponding uncompressed search.

5.1.3 Stopper-Continuer Byte Codes

Building on the insight of `thc`, Brisaboa et al. [2007] propose instead the use of variable byte codes (`bc`), in which the codewords are naturally end-tagged instead of start-tagged, as was the case with the `thc` method. Coding and decoding is significantly simplified when transitioning to this process. Furthermore, compression is further improved by using (S, C)-dense codes, as previously discussed in Section 4.1.4, with no loss in decoding efficiency. These properties make (S, C)-dense codes a sound choice when designing compressed pattern matching algorithms.

False matches can occur in stopper-continuer codes, but low-overhead filtering is possible. The only modification necessary to search directly in a stopper-continuer byte code is to add a false matching filter. Figure 5.2 shows an example of a false

match in a stopper-continuer based pattern matching scheme. Here, we see a perfect match for the compressed pattern in the corresponding compressed text. However, the first codeword in the patten is a single byte, while the first aligned byte in the text is actually the last byte of a three byte codeword: resulting in a *false match*. Thankfully, it is relatively simple to identify and ignore false matches in a stopper-continuer coded text. A typical filter consists of testing the byte in the text which immediately precedes the window of which a potential match has been found. If the byte is a continuer, then the alignment is a false match and processing continues, otherwise the position is added to the occurrence list. No other modifications are required, and the additional compare has minimal impact on searching in practice.

Brisaboa et al. [2007] show that when coupled with a spaceless-words model, (S, C)-dense codes are superior to tagged Huffman codes in terms of both efficiency and effectiveness and that for small to medium length texts, (S, C)-dense codes are $5-10\%$ faster than searching over `thc` compressed text. However, `thc`-based searching, in conjunction with `horspool`, results in faster searches than the corresponding `scbc` based search. Brisaboa et al. attribute this discrepancy to the fact that codewords are longer on average using `thc`, and `horspool` benefits from longer search patterns. However, shorter codewords lead to less text to evaluate overall. This paradox can make evaluation difficult, as pattern length and compressed text size appear to be in tension. A careful evaluation of pattern length and search performance in compressed pattern matching is carried out in Section 5.3.

5.2 Restricted Prefix Byte Codes

The restricted prefix byte codes described in Section 4.2 cannot use the same filtering methods as stopper-continuer codes for false-matches since more of the available codespace is used. A straightforward approach is to ensure that false matches can never occur. This can be accomplished by always aligning the first byte of the pattern on a codeword boundary in the text rather than allowing arbitrary assignments. To do this, we must ensure that each pattern shift is by a full codeword, to align with the next

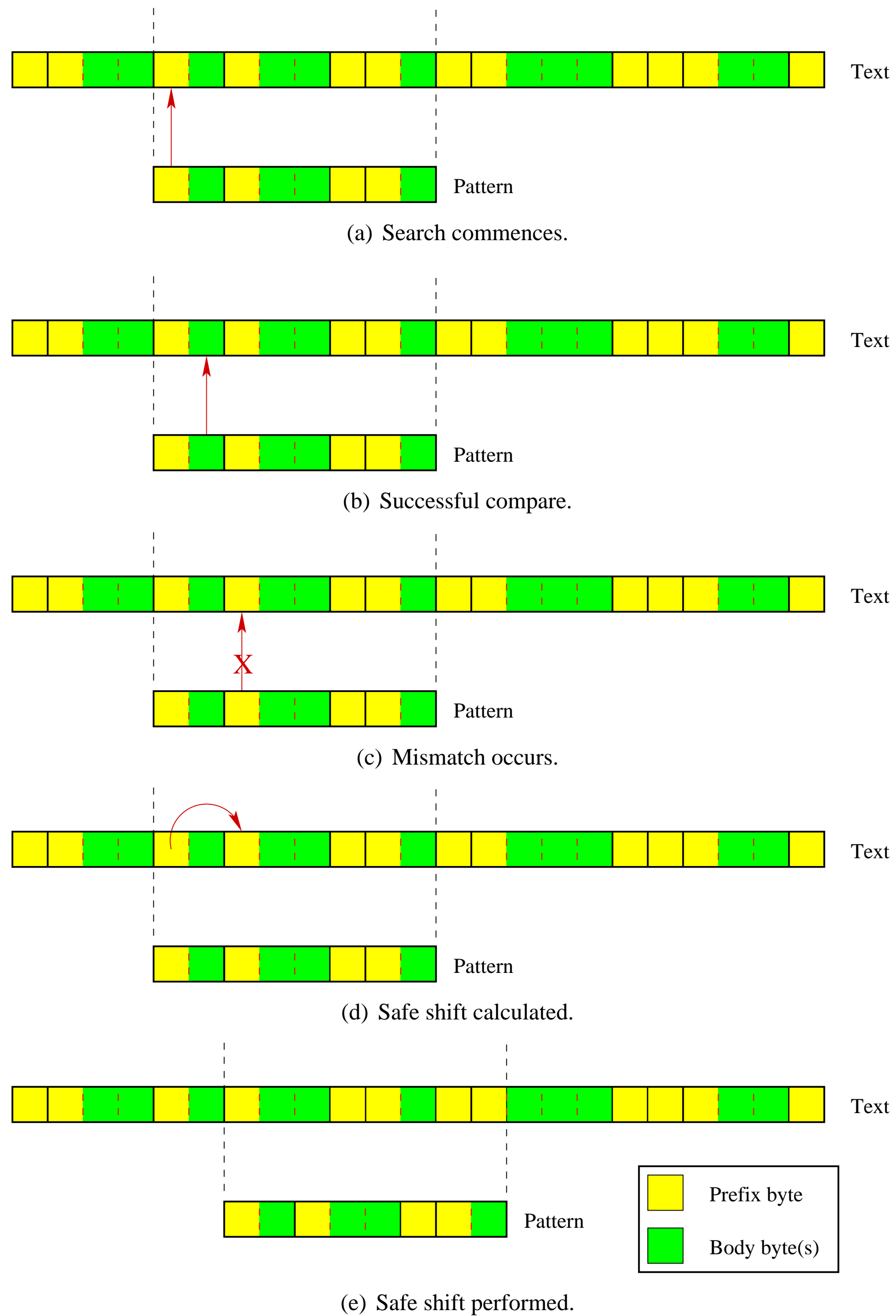


Figure 5.3: Brute-force searching in restricted prefix byte codes. False matches are filtered out by enforcing shifts which retain pattern alignment on codeword boundaries.

Algorithm 7 Seeking in Restricted Prefix Byte Codes.

INPUT: A seek offset s, and `rpbc` control parameters v_1, v_2, v_3, and v_4, with $v_1 + v_2 + v_3 + v_4 \leq R$, where R is the radix (typically 256). The function *create_tables* is the same as previously defined in Algorithm 4.
OUTPUT: A total of $s - 1$ codewords have been skipped over.

1: *create_tables*(v_1, v_2, v_3, v_4, R)
2: **for** $i \leftarrow 0$ **to** $s - 1$ **do**
3:     assign $b \leftarrow$ *get_byte*$()$
4:     adjust the input file pointer forwards by *suffix*$[b]$ bytes
5: **end for**

Algorithm 8 Brute-force searching in Restricted Prefix Byte Codes

INPUT: An `rpbc`-compressed array $\mathcal{Z}$ of compressed length n bytes, a compressed pattern $\mathcal{P}$ of length m bytes when compressed, and `rpbc` control parameters v_1, v_2, v_3, and v_4, with $v_1 + v_2 + v_3 + v_4 \leq R$, where R is the radix (typically 256). The function *create_tables* is the same as previously defined in Algorithm 4.
OUTPUT: The set of occurrences at which $\mathcal{P}$ appears in $\mathcal{Z}$, presented as a set of byte offsets in the compressed text $\mathcal{Z}$.

1: set $t \leftarrow 0$ and $p \leftarrow 0$ and $occ \leftarrow \{\}$
2: *create_tables*(v_1, v_2, v_3, v_4, R)
3: **while** $t \leq n - m$ **do**
4:     **while** $p < m$ **and** $\mathcal{P}[p] = \mathcal{Z}[t + p]$ **do**
5:         set $p \leftarrow p + 1$
6:     **end while**
7:     **if** $p = m$ **then**
8:         set $occ \leftarrow occ \cup \{t\}$
9:     **end if**
10:     set $t \leftarrow t +$ *suffix*$[\mathcal{Z}[t]] + 1$ and $p \leftarrow 0$
11: **end while**

available codeword boundary in the text. This is not too onerous, but it does require that one or more of the bytes in the aligned text to be inspected at each matching attempt, which can degrade performance. Figure 5.3 sketches the modified approach to pattern shifting on codeword boundaries. The method of shift determination is equivalent to "seeking" in `rpbc` compressed text. Algorithm 7 shows how to accomplish efficient skipping using an `rpbc` code.

Building on Algorithm 7, it is possible to devise a modified brute-force searching approach. In order to see this more clearly, consider a simple codeword alignment modification to a brute-force searching algorithm. Algorithm 8 shows a simple

preprocessing modification which can be used with `rpbc` since the prefix byte clearly defines the codeword boundaries. Now, each iteration of the outer-loop at statement 5 shifts directly to the next codeword boundary before re-commencing the matching. This approach is not possible with stopper-continuer byte codes, since there is no way to compute the codeword length without looking at each byte. That is, Algorithm 8 performs fewer comparisons than an equivalent search on `scbc` compressed text on average. In addition, false matches are impossible, since a shift never places the alignment between codeword boundaries. This allows efficient searching in `rpbc` coded text using any of the prefix-based to pattern matching.

A similar technique can be employed for suffix-based searching approaches, such as the highly effective `horspool` algorithm. Algorithm 9 shows the resulting modification to the original `horspool` algorithm. After the best shift from `horspool` is generated, the prefix bytes of the corresponding text are quickly traversed to find the next codeword boundary. If a shift value is returned from the generic `horspool` preprocessing table which falls in the middle of a codeword, the next codeword boundary is found, resulting in an increased shift. This modified approach guarantees that longer shifts are generated on average than in an equivalent `scbc` coded text. But, the approach has a small overhead in the form of additional lookups of text prefix bytes to find the next codeword boundary.

So, `rpbc` encoded texts can be searched using any standard pattern matching algorithm, with careful modifications which ensure each matching step is properly aligned on a codeword in the text. However, there are still limitations which must be addressed. For example, this approach does not allow moving backwards in a text compressed with a `rpbc` code to commence decoding at a point prior to a known codeword boundary.

5.2.1 Prefix Array Based Codes

To allow reverse scanning, either an implicit stack of past codeword boundaries must be maintained, or the underlying codes must be restructured. One such structural

Algorithm 9 BMH searching in `rpbc`

INPUT: A compressed text $\mathcal{Z}$ of length n and a compressed pattern $\mathcal{P}$ of length m and control parameters v_1, v_2, v_3, and v_4, with $v_1 + v_2 + v_3 + v_4 \leq R$, where R is the radix (typically 256). Assume initially that $\mathit{shift}[i] = 0$ and $\mathit{suffix}[i] = 0$ for $0 \leq i \leq 255$.
OUTPUT: The set of occurrences *occ* at which $\mathcal{P}$ appears in $\mathcal{Z}$, presented as a set of byte offsets in the compressed text $\mathcal{Z}$.

1: set $i \leftarrow m - 1$ and $j \leftarrow 0$ and $k \leftarrow 0$
2: set $s \leftarrow 0$ and $\mathit{occ} \leftarrow 0$
3: $\mathit{preprocess_bmh}(v_1, v_2, v_3, v_4, R, \mathit{suffix}, \mathcal{P}, m, \mathit{shift})$
4: **while** $i < n$ **do**
5:     set $w \leftarrow 0$ and $j \leftarrow i$ and $k = m - 1$
6:     **while** $k \geq 0$ **and** $\mathcal{Z}[j] = \mathcal{P}[k]$ **do**
7:         set $j \leftarrow j - 1$ and $k \leftarrow k - 1$
8:     **end while**
9:     **if** $k < 0$ **then**
10:         set $\mathit{occ} \leftarrow \mathit{occ} \cup \{i\}$
11:     **end if**
12:     set $s \leftarrow \mathit{shift}[\mathcal{Z}[i]]$
13:     **while** $w < s$ **do**
14:         set $w \leftarrow w + \mathit{suffix}[\mathcal{Z}[i - m + 1 + w]] + 1$
15:     **end while**
16:     set $i \leftarrow i + w$
17: **end while**

function $\mathit{preprocess_bmh}(v_1, v_2, v_3, v_4, R, \mathit{suffix}[], \mathcal{P}, m, \mathit{shift}[])$
The function *create_tables* is the same as previously defined in Algorithm 4.

1: set $i \leftarrow 0$
2: $\mathit{create_tables}(v_1, v_2, v_3, v_4, R)$
3: **for** $i \leftarrow 0$ **to** $R - 1$ **do**
4:     set $\mathit{shift}[i] \leftarrow m$
5: **end for**
6: **for** $i \leftarrow 0$ **to** $m - 1$ **do**
7:     set $\mathit{shift}[\mathcal{P}[i]] \leftarrow m - i - 1$
8: **end for**

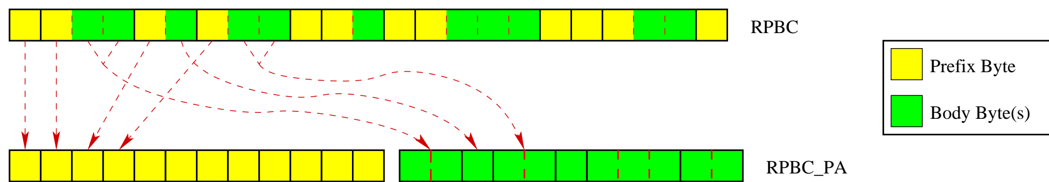


Figure 5.4: Construction and traversal of an array based prefix code representation. The prefix and suffix bytes of each codeword are partitioned into two separate arrays. The compressed sequence can then be traversed forwards and backwards by maintaining a suffix offset.

change to make traversing `rpbc` codes more flexible is to separate the prefix bytes and the suffix bytes into separate streams. This approach, referred to as `rpbc_pa`, offers additional pattern matching alternatives. By separating the prefix and suffix bytes, pattern matching algorithms can be applied to a subset of each codeword and the remainder is compared only to remove false matches. Essentially, the prefix bytes act as a simple codeword hash which is verified only when a possible match has been found.

Figure 5.4 outlines the basic approach to traversing a `rpbc_pa` code. The prefix array can be sequentially scanned forwards and backwards and a second pointer into the body array must be maintained in order to find the corresponding body bytes. Assuming the same codeword assignments are used as in Table 4.1 on page 72, the integer sequence,

$$1, 3, 4, 5, 3, 4, 1, 6, 5, 7, 5, 6, 1, 7$$

can be represented with a prefix array,

00 11 01 10 11 01 00 11 10 11 10 11 00 11

and the corresponding body array,

00 00 01 10 01 10.

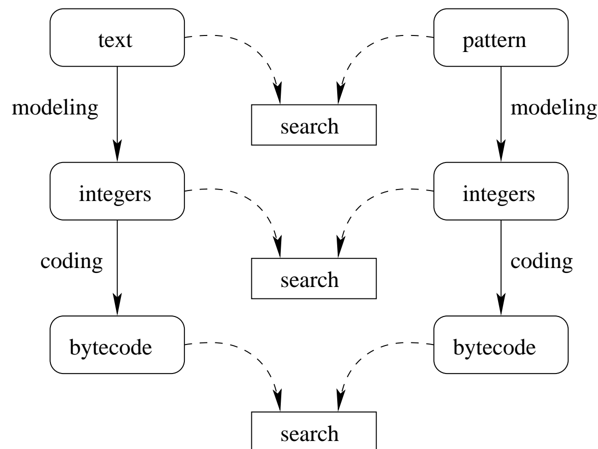


Figure 5.5: Modeling and coding-based decomposition for the compressed pattern matching problem. It is possible to initiate the search in one of three places: the uncompressed, character representation; the integer intermediate produced by the model; and the compressed, byte-aligned representation.

This approach lowers text and first pass pattern length to one byte per symbol. The additional suffix bytes are only used to filter out false matches. However, each prefix byte must be evaluated as the sequence is traversed in order to maintain the pointer into the body array, limiting the usefulness of this approach in pattern matching algorithms which generate large shifts on average. Sentinels can be used at predetermined locations in the prefix array to remove the requirement of inspecting each prefix byte, but at the cost of compression effectiveness.

5.3 Experiments

Byte codes have several interesting properties which allow searching and seeking to be performed *in situ*. Firstly, byte codes offer a unique advantage over other coding techniques: they allow straightforward decoding of individual symbols in a sequence. Secondly, byte codes allow standard pattern matching algorithms to be applied directly to the compressed sequence. Finally, searching in large alphabet “intermediates” has received little attention. Compression models often produce an intermediate stream

File name and origin	Total Symbols (n)	Alphabet ($\|\Sigma\|$)	Self-information (bits/sym)
WSJ-WRD: Spaceless words parsed file	56,908,910	222,329	10.60
WSJ-REP: Phrase numbers from a recursive byte-pair parser	19,254,349	319,757	17.63
ZIPFA: Synthetic Zipfian distribution, $\alpha = 1.1$	250,000,000	999,753	10.89
UNIF: Synthetic Uniform distribution	250,000,000	500,000	18.93

Table 5.1: Statistical characteristics and the origin of the test collection used in the search experiments. Additional information about the test collection can be found on page 81.

of integers, which makes it possible to also apply the pattern matching algorithm at the midpoint of the coding and modeling phase using word-aligned pattern matching algorithms.

Figure 5.5 outlines the three possible entry points for compressed pattern matching. The simplest approach is to decompress the text and initiate a character-based search in the original text. It is also possible to perform the search on the integer intermediate produced by in the modeling phase, or directly in the byte-aligned, encoded text. In order to better understand the trade-offs, we investigate algorithm performance using intermediate integer sequences drawn from the datasets previously discussed in Section 4.4.1 on page 81.

5.3.1 Experimental Setup

Experiments focused on word-based two intermediate representations of a 267 MB segment of SGML-tagged newspaper text previously discussed in Section 4.4.1, and two artificially generated files which use large, dense alphabets. Table 5.1 shows the basic statistical properties of each test file. The first file, WSJ-WRD, is a sequence of integers generated from a spaceless word model as described earlier. The second file, WSJ-REP, is a sequence of integers which represent common phrases generated via an

off-line, word-based, encoding method called RE-PAIR, similar to the method used by Wan [2003]. The data files WSJ-IDX and WSJ-BWT are not considered here as we are interested in streams of words for sequential search applications. The ZIPFA and UNIF files are representative of Zipfian and Uniform distributions respectively, and utilize large, dense alphabets.

The average length of queries in web based information retrieval systems is around 2.4 words per query [Spink et al., 2001]. So, we systematically performed our experiments using phrases of 1 to 5 words. In order to accurately measure the average search time, 100 queries of each length were generated from the uncompressed integer sequences by choosing a random position in the target file and recording the pattern. This ensured that each pattern appeared at least once and the experimental results were reproducible. The integer pattern can easily be converted to either the original character based pattern via a dictionary lookup, or to a compressed representation of the pattern using the corresponding encoding algorithm.

In the first set of experiments, the intermediate integer sequences are used to explore the impact of very large alphabet sizes on classical pattern matching algorithm performance. The files are then compressed using several different variants of the byte codes. The five coding algorithms investigated were `bc`, `dbc`, `scbc`, `rpbc`, and `rpbc_pa`. The prelude lookup costs are amortized, since the prelude is only used when building the compressed representation of the search pattern. In addition, the entire file (compressed or uncompressed) is transferred into memory before timings are taken.

Figure 5.6 shows an example of how different search pattern representations are generated from the intermediate integer pattern, using one of the queries applied to the file WSJ-WRD. The three word sequence $910, 2685, 153$, is the pattern being searched for, and reversal of the spaceless words parser reveals the original character based word phrase to be "`offer may be`". Then, by applying different encoding algorithms, a set of compressed search patterns are generated. The length of the resulting compressed patterns are not all the same. In fact, the length of the pattern directly corresponds to the effectiveness of the compression algorithm chosen, and better compression leads

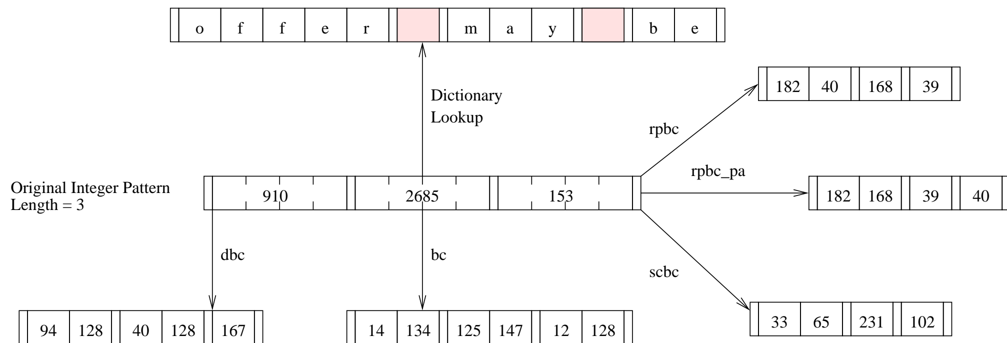


Figure 5.6: Examples of uncompressed and compressed patterns generated from an integer sequence pattern. The original integer sequence is from the spaceless words file WSJ-WRD, and represents the three word phrase "`offer may be`".

to shorter compressed patterns. The variance in pattern length can have a measurable impact on some of the search algorithms employed.

5.3.2 Seeking in Compressed Text

The first set of experiments were devised to determine how efficient each of the byte coding methods are at seeking to random positions and decoding the symbol. Figure 5.7 compares the efficiency of each byte code variant at decoding the n th item in the compressed sequence. The gap between the n th items is then slowly reduced, until all values in the sequence are being decompressed. So, in the first run, no values are decoded, and the whole sequence is traversed. All codeword boundaries are still found in the process, but no symbols are explicitly decoded. In the next iteration, a single value is decoded and the gap between the targeted extraction values is halved. This continues until all values are being decoded, with a gap of zero.

The `dbc`, `scbc`, and `rpbc` method all use a generic permutation prelude. The restricted prefix, semi-dense byte code, referred to here as `rpsbc`, uses a semi-dense prelude, in which only codewords of 1 or 2 bytes are densely mapped. The `bc` method requires no prelude. Using this approach, the `rpbc` method is more efficient than `dbc` or `scbc`, by a constant factor, since `rpbc` doesn't have to evaluate every byte during seeking, whereas stopper-continuer codes must. The `bc` and `rpsbc` methods benefit

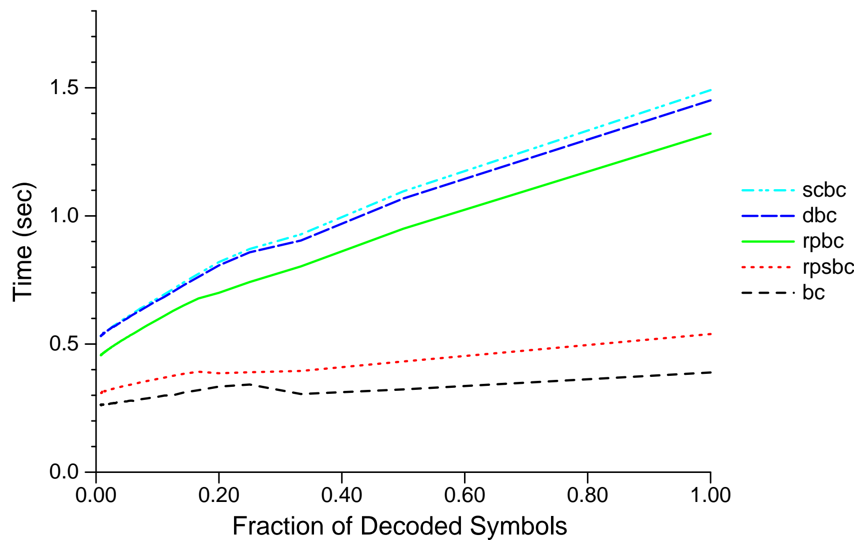


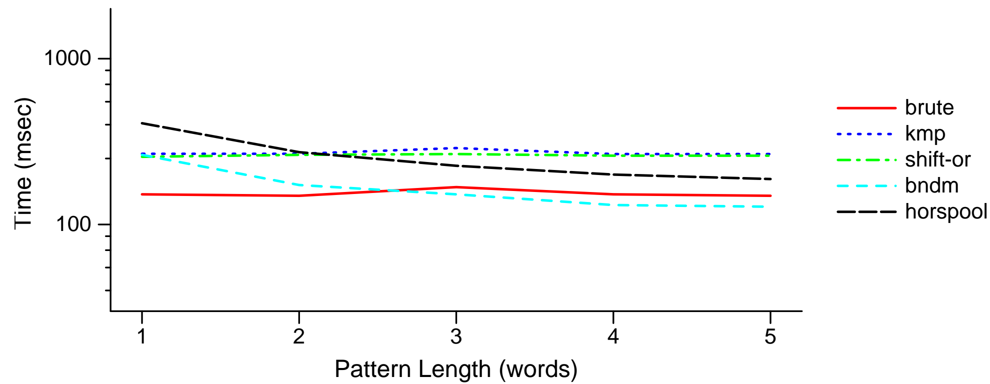
Figure 5.7: Seeking performance comparison for byte-aligned codes, represented as a percentage of values decoded, for the file WSJ-REP on 2.8 Ghz Intel Xeon with 2 GB of RAM.

from a reduction in cache misses, which are a byproduct of codeword lookups necessary to resolve the final output symbol. The performance penalty of maintaining a complete rank mapping is particularly noticeable when the percentage of decoded text is high.

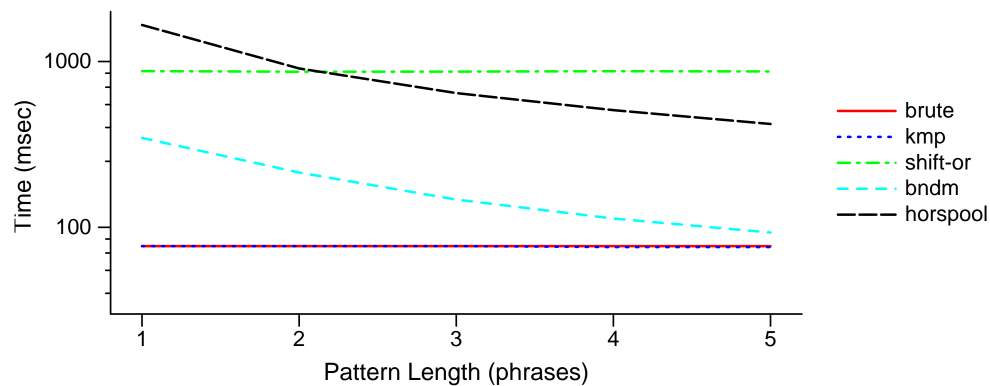
There is a clear penalty for maintaining a complete prelude. The `rpbc`-based methods out-perform stopper-continuer based methods for sequential seeking and decoding. On-line pattern matching algorithms process the text sequentially, and, therefore, `rpbc` based codes can be used in conjunction with many of the standard search algorithms.

5.3.3 Integer Intermediate Pattern Matching

The next set of experiments were designed to evaluate the cost of using large alphabets in the intermediate, integer-aligned text sequence. In order to facilitate integer-on-integer searching, the implementation of each pattern matching algorithm was modified



(a) Results for WSJ-WRD

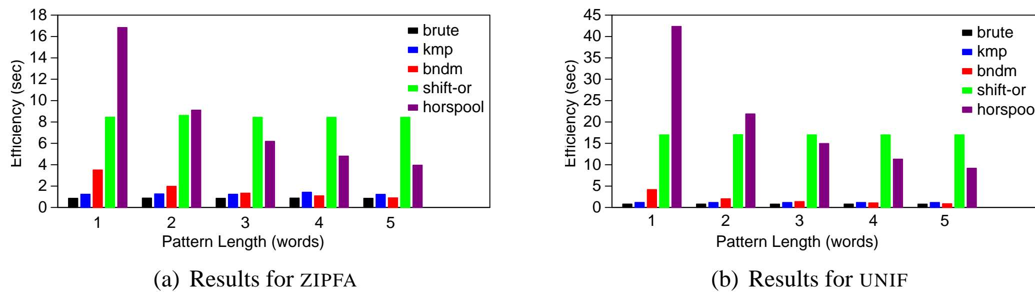


(b) Results for WSJ-REP

Figure 5.8: The impact of large alphabets on intermediate, integer-on-integer pattern matching, using a 2.8 Ghz Intel Xeon with 2 GB of RAM. The methods used were brute-force; the Knuth-Morris-Pratt method; Shift-Or; Backward Nondeterministic DAWG matching; and the Horspool variant of the Boyer-Moore method.

to process integers instead of characters (bytes).[1] Figure 5.8 shows the relative performance of several different searching methods using integer-on-integer pattern matching. Algorithms which use preprocessed lookup tables based on the size of

[1] One of the examiners of this thesis suggested that the integers can be partitioned into bytes or short integers, and false matches filtered out by enforcing typesize alignments. This reduces the likelihood of cache misses in algorithms such as `horspool` by limiting the alphabet size to 256, plus mismatch filtering. For example, the byte-aligned approach reduces the average search time using `horspool` for all patterns in the file WSJ-WRD from 259 ms to 188 ms. Combining short (2 byte) integers with false match filtering performs slightly worse than standard integer aligned methods, averaging 353 ms. All of these approaches fall short of `horspool` combined with dense byte-aligned coding methods, which average 101 ms over all patterns in WSJ-WRD, and are discussed futher in Section 5.3.4.



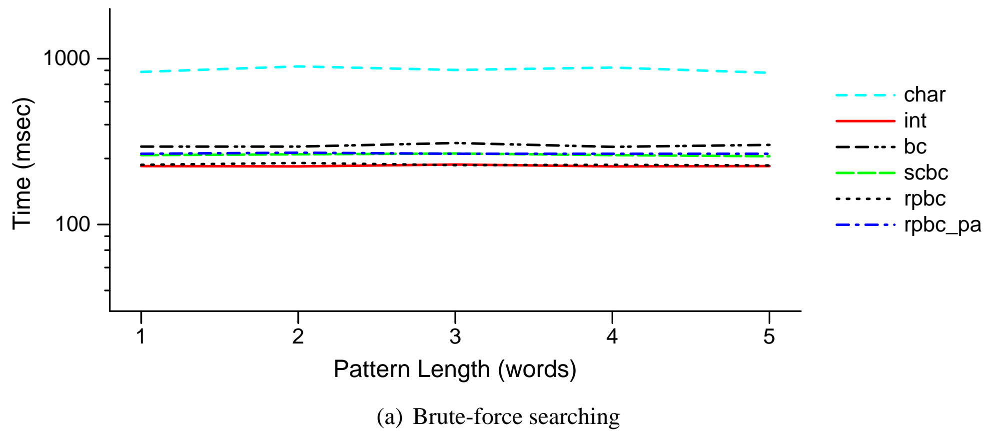
(a) Results for ZIPFA (b) Results for UNIF

Figure 5.9: The impact of pattern length on intermediate, integer-on-integer pattern matching algorithms when $\sigma \gg 256$, using a 2.8 Ghz Intel Xeon with 2 GB of RAM. Two 1 GB artificially generated sequences are used, and the timings reflect the average of 100 random queries for each pattern length.

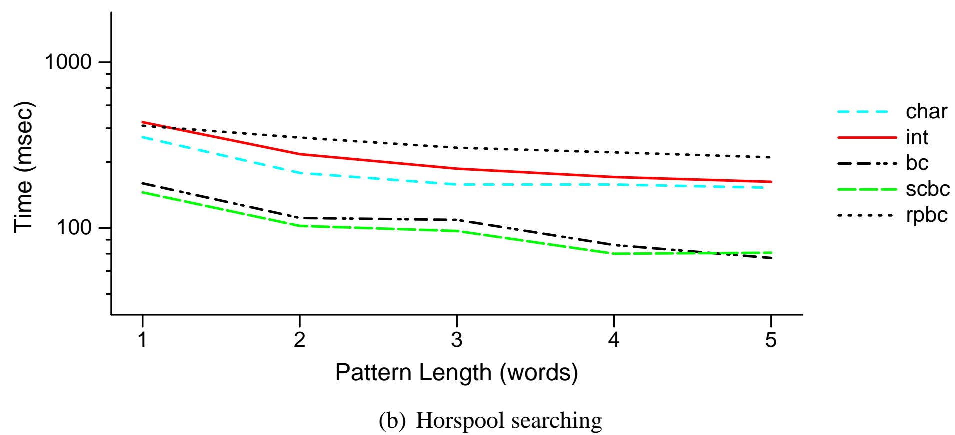
the alphabet, such as `horspool`, tend to suffer considerably when pattern lengths are short. The large lookup tables result in cache misses, which tend to offset any gains achieved by the improved shifts, until the search patterns are of sufficient length. Algorithms which preprocess the pattern instead, such as KMP, fare better for short patterns. Somewhat surprisingly, brute-force approaches outperform all of the more elegant algorithms for patterns of three words or less, irrespective of the input file's probability distribution.

These affects are even more noticeable when considering the statistical differences between WSJ-WRD and WSJ-REP. The WSJ-REP file has a much larger and dense alphabet, resulting in more cache misses in alphabet-based lookup tables. Also, the `brute` method performs even better on WSJ-REP than on WSJ-WRD, as the probability distribution is flat, shedding light on its surprising performance. Brute-force search has a $\mathcal{O}(mn)$ worst case performance bound, but the average case complexity approaches $\mathcal{O}(n)$ time on *random* text [Cormen et al., 2001].

Figure 5.9 shows the efficiency of the intermediate, integer-on-integer search algorithms on the artificial files ZIPFA and UNIF. The two 1 GB artificially generated files ZIPFA and UNIF are used, and the timings reflect the average of 100 random queries for each pattern length. The impact of large, dense alphabets is even more revealing here. Again, `horspool` suffers a severe performance penalty, caused by short



(a) Brute-force searching



(b) Horspool searching

Figure 5.10: Performance comparisons for `brute` and `horspool` pattern matching methods in the WSJ-WRD file on a 2.8 Ghz Intel Xeon with 2 GB of RAM. The character-based, integer-based, and byte-aligned, compressed representations are all shown.

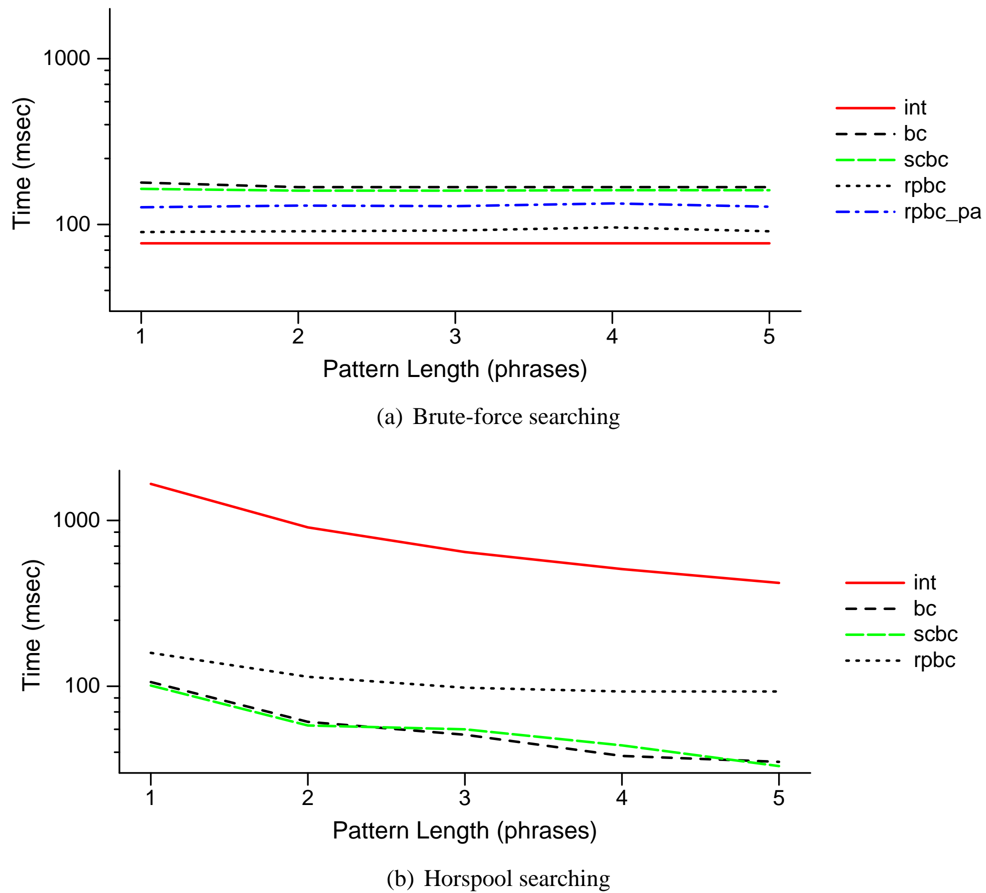
patterns, and cache misses in the shift offset table. The pattern-based preprocessing in `kmp` produces consistent efficiency, and `bndm` shows good performance which improves with pattern length. The brute-force method outperforms the other methods for all pattern lengths for these test files, reinforcing that it should not be ignored as a potential solution.

5.3.4 Byte-Aligned Pattern Matching

The pattern matching results for searching directly in the integer-based intermediates reveal a few interesting empirical facts about standard search algorithms. Brute-force works well in a variety of cases, and very large alphabets can produce unexpected side-effects when measuring practical efficiency. Of course, these revelations may not necessarily carry over to byte-aligned pattern matching, where the alphabet size is always 256. So, the next question is: how do these algorithms perform when the properties of the underlying text are changed by the compression algorithms?

Figure 5.10 shows the compressed searching results for the `brute` and `horspool` methods, on the spaceless words file, WSJ-WRD. In order to do perform the experiment 100 random patterns for each length, of 1 to 5 English words, are selected and a corresponding representation is created, as previously discussed in Figure 5.6. The `char` method represents searches for the character-based representation of the pattern, in the original, unprocessed text file, using the corresponding unmodified search algorithm. The `int` method represents searching in the processed, but uncompressed intermediate sequence of integers, as discussed in Section 5.3.3. When using the brute-force method, `rpbc` performs better than all of the other byte coding methods, but searching in the uncompressed integer intermediate file is more efficient. However, if you include decompression costs in the `char` or `int` baselines, all byte coding methods perform considerably better than the "decompress then search" baseline. The stopper-continuer byte codes perform considerably better than `rpbc`-based methods when coupled with the `horspool` method, as longer shifts generated by the `rpbc` approach are offset by the additional compares necessary to locate the next codeword boundary.

In Figure 5.11, we see the same experiment on the WSJ-REP file. In the WSJ-WRD file, the symbol distribution is essentially flat, and the large, dense alphabet of the integer-aligned `horspool` method performs poorly. But, using the relatively small alphabet of the byte-aligned methods increases the performance dramatically, as fewer cache misses occur in the shift array. Again, `scbc` coupled with `horspool` searching clearly outperforms the equivalent `rpbc`-based methods.

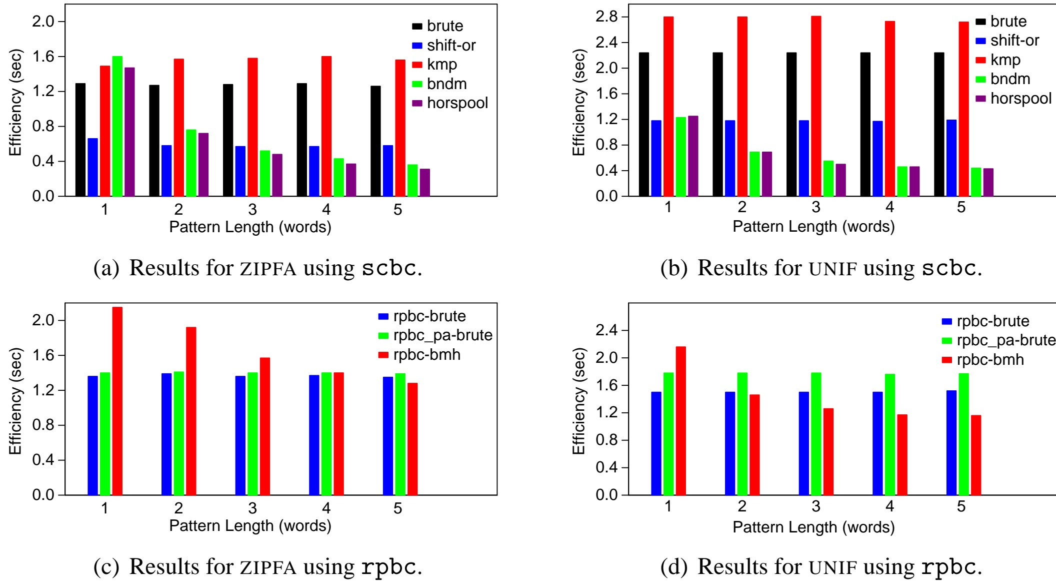


(a) Brute-force searching

(b) Horspool searching

Figure 5.11: Performance comparisons for `brute` and `horspool` pattern matching methods in the WSJ-REP file on a 2.8 Ghz Intel Xeon with 2 GB of RAM.

Figure 5.12 shows the effects of increasing the pattern length on matching efficiency in compressed, byte-aligned text. The two 1 GB artificially generated files ZIPFA and UNIF are used, and the timings reflect the average of 100 random queries for each pattern length. The top two graphs compare five different pattern matching algorithms when used in conjunction with the `scbc` method. For short patterns, `shift-or` is superior. As the pattern length increases, suffix-based pattern matching algorithms such as `horspool` and `bndm` become more attractive. The bottom two graphs show the same experiment using modified `rpbc` pattern matching approaches. The `rpbc` brute-force search methods have the same efficiency as their `scbc` counterparts, but the suffix-



(a) Results for ZIPFA using `scbc`.

(b) Results for UNIF using `scbc`.

(c) Results for ZIPFA using `rpbc`.

(d) Results for UNIF using `rpbc`.

Figure 5.12: The impact of pattern length on compressed, byte-aligned pattern matching algorithms when $\sigma = 256$, using a 2.8 Ghz Intel Xeon with 2 GB of RAM. Two 1 GB artificially generated sequences are used, and the timings reflect the average of 100 random queries for each pattern length.

based methods no longer benefit from increased pattern length, presumably because more compares are necessary in the inner-loops to calculate legal shifts.

In hindsight, the poor performance of `rpbc` relative to equivalent stopper-continuer searches can be attributed to one key observation. The `rpbc` based search algorithms must perform one or more additional compares for every *mismatch*, while stopper-continuer based search algorithms add a single compare for every possible *match*. The number of possible matches in the text is typically much smaller than the number of mismatches, leading to a clear performance penalty. This discrepancy highlights one key property which facilitates efficient compressed pattern matching: *every* byte should be marked in such a way as to support random seeks. Without such markings, pattern matching algorithms are required to re-calculate shifts on every mismatch.

5.4 Summary

Recent advances in the pattern matching conclusively show the critical role compression can have in reducing space usage while actually *improving* algorithm efficiency. We have described several techniques for searching directly in a text compressed using various byte codes. Additionally, we have shown how large alphabets can have an unexpected impact on algorithms which are in common use, such as BMH. However, efficient searching is possible on integer intermediates, and may be useful in certain applications.

When the average length of a phrase-based query is taken into account, brute-force algorithms often perform remarkably well, and carry no additional preprocessing overhead. On average, brute-force algorithms perform best when the underlying search text is random, and individual bytes in compressed text are far more randomized than the underlying natural language text. All of the byte code variants perform considerably better than the generic "decompress then search" baseline proposed by Amir and Benson [1992], rendering them useful in applications today. In particular, the stopper-continuer coding methods, such as `scbc`, when combined with `horspool` or `bndm`, are considerably more efficient than searching in uncompressed text with the same pattern matching algorithm, and allow the text to be stored in compressed form. All things considered, combining the `scbc` coding method with the `horspool` search algorithm remains the best choice for an application which must support basic pattern matching directly in compressed natural language text.

Random access is a crucial property in any byte-aligned coding scheme which hopes to support efficient suffix-based search algorithms like BMH. Codes which do not allow codeword boundaries to be discovered after random length shifting require the search algorithms to be modified. This is a bad idea in practice. Minor changes to the tight loops in classic pattern matching algorithms often have a large impact on performance, and should be avoided. Coding algorithms which can combine the flexible codeword assignment of the `rpbc` method, and are also able to support codeword boundary marking, such as the `scbc` method allows, remains an open problem.

Chapter 6

Compact Sets

Efficient set representation is a fundamental problem in computer science. There are a wealth of possible approaches, each with their own unique trade-offs. Possible choices include traditional pointer-based data structures such as hash tables, various tree-based representations, and less structured approaches, such as bitvectors and ordered arrays. The tension between query performance and space usage is clearly evident in many data structures, and the most appropriate choice for a particular application is dictated by the operations which are performed most often. For instance, hash table approaches guarantee $\mathcal{O}(1)$ **MEMBER** queries, at the cost of using a sparse array representation. Alternately, we may prefer to constrain space usage instead of query performance. To accomplish this, a compressed list of d-gaps may be used, but the cost to perform **MEMBER** queries now grows to $\mathcal{O}(n)$.

A deceptively simple question embodies the motivation behind the interest in designing new space-efficient data structures: Is it possible to use minimal space in an information theoretic sense, while supporting optimal query performance? More formally, given an ordered set $S = \{s_1, s_2, \ldots, s_n\}$, where $s_i < s_{i+1}$, drawn from a universe $U = \{0, 1, \ldots, u-1\}$, devise a representation of S which supports efficient operations and uses $\lceil \log \binom{n}{u} \rceil$ bits of space. In order to achieve this goal, the representation choice is driven by the subset of primitive operations which are necessary to support the required query operations.

In this chapter, we investigate practical trade-offs in space consumption and query performance for ordered sets of integers. As was discussed in Section 3.2.1, ordered sets are commonly used in inverted indexes. One important operation on sets is the **INTERSECT** operator, since it plays a key role in conjunctive Boolean queries. Such queries play an important role in traditional AND-mode search queries as well as ranked search query environments such as PageRank [Zobel and Moffat, 2006]. A conjunctive Boolean query can be abstractly conceptualized as a $|q|$-way intersection of $|q|$ ordered sets of integers. This abstraction allows flexibility when choosing an intersection algorithm or data representation. Unfortunately, the most efficient intersection algorithms depend on non-sequential access to elements of the set and are not amenable to standard sequential decompression methods.

6.1 Set Operations For Inverted Lists

Set manipulation can be reduced to the classical *dictionary problem*, with three key operations needed: **INSERT**, **DELETE**, and **MEMBER**. However, search engines typically amortize the cost of modifying operators such as **DELETE** and **INSERT** over a whole stream, in order to facilitate faster **MEMBER** queries. If we are allowed to focus on static data representations and are not required to support modifying operations in an on-line fashion, it is much simpler to manipulate the underlying succinct data structures.

More complex query operations can be built on top of a subset of navigation-based operations. In inverted indexes, intermediate operations such as **INTERSECT**, **UNION**, and **DIFFERENCE** can be built on top of **F-SEARCH**, **PREDECESSOR**, and **SUCCESSOR** operations. There is a notion of the "current" element in these operations, and, as the sequence of operations unfolds, activity proceeds from one element to another. In some representations, these operations can in turn be built on top of **RANK** and **SELECT**. For instance, **SUCCESSOR**(S) can be implemented as **SELECT**$(S, 1 + $ **RANK**$(c))$, where c is the value of the current item. If **RANK** and **SELECT** operations are available, strategically chosen indices in a sequence of **SELECT** operations can be used to provide an efficient non-sequential implementation

of the **F-SEARCH** operation. For a more comprehensive discussion of the relationship between various set operations, refer to Section 3.3. Our focus in this chapter is primarily the **INTERSECT** operation, implemented via a combination of **SUCCESSOR** and **F-SEARCH** calls.

6.2 Set Representations

In this section, we review various approaches to to represent sets, and discuss the trade-offs between space usage and operator efficiency. A more general discussion of set representations is in Section 3.3.

6.2.1 Array of Integers

Perhaps the simplest way to represent a set is to store it as an array of sorted integers. For example, the following set S can easily be manipulated in various ways:

$$\{1, 4, 5, 6, 8, 12, 15, 16, 18, 20, 25, 26, 27, 28, 30\}$$

The array-of-integer representation is not particularly space-efficient, but offers flexibility in operator support. Implementations of **MEMBER**, **RANK**, and **F-SEARCH** require $\mathcal{O}(\log n)$ time, $\mathcal{O}(\log n)$, and $\mathcal{O}(\log d)$ time respectively, where d is the number of elements between each successive search. Operations such as **UNION** and **DIFFERENCE**, which potentially include all members of the largest set, take $\mathcal{O}(n_2)$ time, where $n_1 \leq n_2$ are the sizes of the two sets involved. The **INTERSECT** operator, which will be considered in more detail shortly, requires $\mathcal{O}(n_1 \log(n_2/n_1))$ time.

6.2.2 Bitvectors

A set can also be represented succinctly as a bitvector – a u-bit sequence where the xth bit is 1 iff $x \in S$. If the set S is dense over a *universe* $U = 1 \ldots u$, bitvectors are a good choice. Bitvectors represent an intriguing trade-off as the space required is proportional to u rather than to n, the total number of elements present in the set. This

unique property makes bitvectors very space efficient when $n \approx u$ and unacceptably inefficient when $n \ll u$. The set S in Section 6.2.1 can be represented as the bitvector:

$$10011101000100110101000011110100,$$

where u is assumed to be 32. Using a bitvector shifts the cost balance of various set operations. Operations such as **INSERT**, **DELETE**, **MEMBER**, and **F-SEARCH** all take $\mathcal{O}(1)$ time; but **UNION**, **DIFFERENCE**, and **INTERSECT** now take $\mathcal{O}(u)$ time. In practice, **UNION**, **DIFFERENCE**, and **INTERSECT** can use *bit-parallelism* to obtain significant constant factor speedups.

Control operations such as **RANK** and **SELECT** are also expensive if unadorned bitvector representations are used. If the application requires these or other control operations such as **PREDECESSOR** and **SUCCESSOR**, the basic bitvector representation is no longer sufficient. Jacobson [1989] showed that addition of a controlled amount of extra space allows **RANK** and **SELECT** to be supported using $\mathcal{O}(\log u)$ bit probes, and, thus, **SUCCESSOR** and **PREDECESSOR** can also be efficient. Munro [1996] later showed that these operations can be supported in $O(1)$ time. The basic idea proposed, called an "indexable" bitvector, depends on the use of partial lookup tables which store cumulative ranks for carefully constrained blocks of elements. But, this method requires an additional space overhead on top of the basic combinatorial cost of Equation 2.3.

Additional improvements to the original approach have been reported recently, including some that address the space overhead associated with sparse bitvectors (for example, [Clark, 1996, Pagh, 2001, Raman et al., 2002]). However, the space overhead for many the approaches is still unacceptable in some cases. Many of these advances are driven by the desire for faster **RANK** and **SELECT** operations, neither of which is strictly necessary for efficient set intersection.

6.2.3 Compressed Representations

The simplest way to produce a compact set representation is to encode each value using $\log u$ bits for each member. However, as was discussed briefly in Section 3.2.1, it is possible to achieve more disciplined compression if the data is transformed into a set of *first-order differences* or d-gaps. Compact representations of sets are often built on top of d-gaps. So, our previous example can be represented as the following list of d-gaps:

$$\{1, 3, 1, 1, 2, 4, 3, 1, 2, 2, 5, 1, 1, 1, 2\}.$$

The transformed list can then efficiently be encoded using one of many variable length codes such as Huffman, Golomb, Elias γ and δ, or static byte and nibble based codes. Choosing the appropriate code often depends on the context. Section 2.5.3 provides a detailed discussion of variable length codes.

Unfortunately, the sequential nature of d-gap based representations means that operations other than **SUCCESSOR** can be prohibitively expensive. Operations such as **INTERSECT** which require efficient **F-SEARCH** can now only be processed in a linear fashion. Even simple **MEMBER** queries incur an $\mathcal{O}(n)$ cost. However, it is possible to improve the efficiency of **F-SEARCH** by adding additional information to the compressed set representation. For instance, *synchronization points* can be added at some constant interval, perhaps every $\mathcal{O}(\sqrt{n})$ positions, to improve performance [Moffat and Zobel, 1996].

More ambitious approaches have been proposed recently which attempt to offset the costs of the linear scan dictated by the d-gap model by using variations of balanced binary search trees. Gupta et al. [2006] describe a two-level data structure in which each level is itself searchable and compressed, extending earlier work by Blandford and Blelloch [2004]. In the Gupta et al. method, each block of elements at the lower level is represented as a balanced binary search tree, stored implicitly via a pre-order traversal, with additional skip pointers to find each child's left subtree. Sitting on top of each "tree" block is a structure which allows the correct tree to be quickly identified.

Algorithm 10 Binary Set Intersection

INPUT: Two ordered sets S and T.
OUTPUT: An ordered set of answers A.

1: without loss of generality assume that $n_1 = |S| \leq n_2 = |T|$
2: set $A \leftarrow \{\,\}$
3: set $x \leftarrow$ **FIRST**(S)
4: **while** x is defined **do**
5: set $y \leftarrow$ **F-SEARCH**(T, x)
6: **if** $x = y$ **then**
7: **APPEND**(A, x)
8: **end if**
9: set $x \leftarrow$ **SUCCESSOR**(S)
10: **end while**

By balancing the sizes and performance of the two structures, good performance is achieved.

6.3 Set Intersection

There has been considerable interest in developing novel approaches to array-based set intersection. In fact, robust data abstractions have been developed for many of the algorithms which allow flexibility in selecting an appropriate **F-SEARCH** algorithm. A *finger search* (**F-SEARCH**) is a query for an element x on an ordered set S, which returns a pointer to the least element $y \in S$ for which $y \geq x$, where x is greater than the value of the current element. In this section, we will review the various set intersection algorithms, as well as different approaches to efficiently implement **F-SEARCH**. We also investigate an intriguing duality between integer compression and finger searching.

When there are exactly two sets, intersection is a straightforward problem. The simplest and most effective approach is to perform an iterative search for the items in the smaller set. Algorithm 10 shows a simple two set intersection where each element in the smaller set S is searched for in the larger set. The search always moves forward and the eliminator item x chosen is monotonically increasing as we proceed from $i = 0 \ldots n$. The general template of Algorithm 10 leaves us free to choose from a range of options for implementing **F-SEARCH**, which will be explored in detail shortly.

Algorithm 11 Set versus Set Intersection (svs)

INPUT: A list of q ordered sets $S_1 \ldots S_{|q|}$. The function *binary_intersect* is the same as previously defined in Algorithm 10.

OUTPUT: An ordered set of answers A.

1: without loss of generality assume that $|S_1| \leq |S_2| \leq \cdots \leq |S_{|q|}|$
2: set $A \leftarrow S_1$
3: **for** $i = 2$ to $|q|$ **do**
4:     set $A \leftarrow$ *binary_intersect*(A, S_i)
5: **end for**

6.3.1 Intersecting Multiple Sets

When there are more than two sets which must be intersected, the operations can be implemented as a sequence of pairwise set intersections. Algorithm 11 shows the set versus set (svs) method, which starts with the smallest set S_1, and in turn intersects it against each of the others, in increasing order of size. Since the candidate set can only get smaller, the worst-case cost of this approach in an array-based implementation, assuming that **F-SEARCH** takes $\mathcal{O}(n_1 \log n_2/n_1)$ time, is

$$\sum_{i=2}^{|q|} n_1 \log \frac{n_i}{n_1} \leq n_1(|q| - 1) \log \frac{n_{|q|}}{n_1} ,$$

where the ordering on the sets is $n_1 \leq n_2 \leq \cdots \leq n_{|q|}$. This method is both simple and effective, and, since the sets are processed two at a time, operations benefit from spatial locality.

Baeza-Yates Intersection (bya)

An svs-based divide and conquer approach to intersection proposed by Baeza-Yates [2004] is shown in Algorithm 12. Again, sets are ordered from smallest to largest. The median value of the smallest set is selected as the initial eliminator. An **F-SEARCH** is performed on the next smallest set and the current eliminator is added to a candidate list if a match occurs. The remaining two subsets are then evaluated recursively in the same manner. After first two sets have been evaluated, the candidate set is sorted and intersected against the next unevaluated set. When all of the sets have been evaluated,

Algorithm 12 Baeza-Yates Intersection Algorithm (bya)

INPUT: A list of q ordered sets $S_1 \ldots S_{|q|}$.
OUTPUT: An ordered set of answers A.

1: without loss of generality assume that $|S_1| \leq |S_2| \leq \cdots \leq |S_{|q|}|$
2: set $C \leftarrow S_1$
3: **for** $i = 1$ to $|q|$ **do**
4:    $C \leftarrow$ **BAEZA_YATES**(C, S_i)
5: **end for**
6: **SORT**(C)

function BAEZA_YATES(C, S_i)
1: **if** $C = \emptyset$ **or** $S_i = \emptyset$ **then**
2:    **return** $\emptyset$
3: **end if**
4: set $A \leftarrow \{\}$
5: set $m \leftarrow |C|/2$
6: set $x \leftarrow c_m$
7: set $y \leftarrow$ **BINARY_SEARCH**(S_i, x)
8: **if** $x = y$ **then**
9:    **APPEND**(A, x)
10: **end if**
11: set $r \leftarrow$ **RANK**(S_i, x)
12: **RECURSE**(C, S_i, m, r)
13: **return** A

the final candidate set is sorted and returned. This approach has a distinct advantage when one of the sets is significantly smaller than the other, but does not fare well against sets of roughly equal size.

Holistic Multi-Set Intersection

Another approach is to process all of the sets at one time and determine the intersection *holistically*. Holistic algorithms hope to achieve better performance than simple svs approaches by exploiting the adaptive complexity of a problem. The basic premise of adaptive complexity is that some problems are harder to solve than others. For example, if all elements in S_1 happen to be smaller than the first element in S_2, then **F-SEARCH** can execute in $\mathcal{O}(n_1)$ total time.

Algorithm 13 Adaptive Intersection Algorithm (adp)

INPUT: A list of q ordered sets $S_1 \dots S_{|q|}$.
OUTPUT: An ordered set of answers A.

```
1: set A ← { }
2: while x is defined do
3:    INSERTION_SORT(S_1 ... S_|q|) based on total remaining elements.
4:    set x ← SUCCESSOR(S_1)
5:    for i = 1 to |q| do
6:       set y ← F-SEARCH(S_i, x)
7:       if x ≠ y then
8:          break
9:       else if i = |q| then
10:         APPEND(A, x)
11:      end if
12:   end for
13: end while
```

Assuming the sets are ordered, the simplest holistic approach is simply to treat each item in the smallest set as an *eliminator* and search for it in the remaining sets in order of size, in an interleaved version of svs. Several adaptive approaches have been proposed recently which differ in the way an eliminator is chosen or by the **F-SEARCH** algorithm used. It is possible to mix and match **F-SEARCH** algorithms with all of the eliminator selection algorithms. Barbay et al. [2006] provide a detailed overview of how such combinations interact, and summarize a range of previous work.

Adaptive Intersection (adp)

Demaine et al. [2000] propose an adaptive method which depends on a dynamic set ordering. Algorithm 13 shows the Adaptive method. In this algorithm, the sets are initially ordered by increasing size. The first unexplored item drawn is from S_1 to act as an *eliminator*. If a mismatch occurs before all $|q|$ sets have been examined, the sets are reordered based on the total number of unexamined items remaining in each set, and the successor in S_1 is used as the new eliminator. For each iteration, the eliminator is selected from the smallest remaining list instead of the smallest overall list. The algorithm terminates when any of the sets have been examined entirely. In practice, the

Algorithm 14 Sequential Intersection Algorithm (seq)

INPUT: A list of q ordered sets $S_1 \dots S_{|q|}$.
OUTPUT: An ordered set of answers A.

```
1: without loss of generality assume that |S_1| ≤ |S_2| ≤ ··· ≤ |S_|q||
2: set A ← { }
3: set x ← FIRST(S_1)
4: set i ← 0
5: while x is defined do
6:    set y ← F-SEARCH(S_i, x)
7:    if x = y then
8:       set k ← k + 1
9:       if k = |q| then
10:         APPEND(A, x)
11:      end if
12:   end if
13:   if x ≠ y or k = |q| then
14:      set x ← SUCCESSOR(S_i)
15:      set k ← 0
16:   end if
17:   set i ← (i + 1) mod |q|
18: end while
```

reordering between each mismatch becomes progressively expensive as $|q|$ increases.

Sequential Intersection (seq)

Building on the notion of adaptive complexity, Barbay and Kenyon [2002] introduced a modification to the adp algorithm which they refer to as the Sequential algorithm. Algorithm 14 outlines the Sequential method. To begin, the sets are ordered in increasing size. The algorithm then cycles through the sets searching for an eliminator (initially set to the first value in the smallest set) until a mismatch occurs, or, all $|q|$ sets have been evaluated. If all sets have been evaluated, the eliminator is added to the answer list, otherwise, a new eliminator is chosen via the **SUCCESSOR** operation in the current set being evaluated. This approach has the potential advantage that now the sets do not need to be reordered, and each set has an opportunity to provide the next eliminator if it causes the most recent mismatch.

Algorithm 15 Max Successor Intersection Algorithm (max)

INPUT: A list of q ordered sets $S_1 \dots S_{|q|}$.
OUTPUT: An ordered set of answers A.

```
1:  without loss of generality assume that |S_1| ≤ |S_2| ≤ ··· ≤ |S_|q||
2:  set A ← { }
3:  set x ← FIRST(S_1)
4:  while x is defined do
5:     if skip then
6:        set startat ← 2
7:     else
8:        set startat ← 1
9:     end if
10:    for i = startat to |q| do
11:       set y ← F-SEARCH(S_i, x)
12:       if x ≠ y then
13:          set x ← SUCCESSOR(S_1)
14:          if y > x then
15:             set skip ← FALSE
16:             set x ← y
17:          else
18:             set skip ← TRUE
19:          end if
20:          break
21:       else if i = |q| then
22:          APPEND(A, x)
23:          set x ← SUCCESSOR(S_1)
24:          set skip ← TRUE
25:       end if
26:    end for
27: end while
```

Max Successor Intersection (max)

Each of the previous methods performs well in certain situations, but none of them are able to take advantage of localized access patterns, like the svs is capable of doing. To overcome this deficiency, we consider a faster approach to the problem. This approach initially draws the eliminator from the smallest set. If a mismatch occurs, the next eliminator is the maximum of the mismatched value or the successor of the smallest set. Processing then starts in S_2 if the eliminator is taken from S_1, otherwise processing begins in S_1. Algorithm 15 describes this new approach.

The intuition behind this approach is two-fold. First, the most discriminating set is the smallest, which suggests that most mismatches will occur between the two smallest sets. Secondly, if most of the compares can be localized in the two smallest sets, fewer cache misses will occur on average.

6.3.2 Efficient F-SEARCH

One of the most attractive features of performing **INTERSECT** operations on an array-of-integers is the wide variety of **F-SEARCH** algorithms which are available. For instance, if the sets are stored as a sorted array, we may choose binary search for simple and fast **MEMBER** queries on the ordered set. A binary **F-SEARCH** in a sorted array representation can be performed using $1 + \lfloor \log n \rfloor$ comparisons. Or, we might choose another search method such as interpolative, Fibonacci, exponential (also referred to as galloping by some authors), or even a simple linear search, all of which are possible on a sorted array representation of sets.

An exponential **F-SEARCH** works well when the next search item is likely to be close to the current location in the larger list. Exponential search reduces the number of comparisons by finding a range r of $2^{\lfloor \log d \rfloor}$ elements, where d is the *rank* difference in the set between the *finger* (pointer to the last accessed position) and the key. The range must contain the search key and is often smaller than the total set length n, allowing more efficient binary searches which find the final position. Exponential search requires $2 + 2\lfloor \log d \rfloor$ comparisons. Bentley and Yao [1976] describe an optimal static **F-SEARCH** algorithm based on exponential search.

Search algorithms which perform better in the average case are described by Gonnet et al. [1980]. Interpolation-based search algorithms attempt to reduce the number of compares using more controlled jumps than a traditional binary search. Interpolation search has an average run time of $\mathcal{O}(\log \log n)$ over uniformly random data and a worst case of $\mathcal{O}(n)$ [Gonnet et al., 1980]. In practice, the running time of interpolation search is often worse than binary search, despite fewer key comparisons, because the cost of calculating the jumps is non-trivial. However, modern processors are quickly closing

Algorithm 16 Golomb Search Algorithm

INPUT : a sorted list L of n elements, a pre-calculated Golomb parameter b, and a search key k.
OUTPUT: an offest in the sorted list L if k is found, the offset of the *successor* of k if not found, or -1 if $k > L[0 \dots n-1]$.

```
1:  set pos ← b
2:  while pos < n and L[pos] < k do
3:     prev ← pos
4:     pos ← pos + b
5:  end while
6:  result ← BINARY_SEARCH(L[prev ... prev + b], pos − prev, k)
7:  if result = −1 then
8:     return −1
9:  else
10:    return result + prev
11: end if
```

the gap between interpolation and binary search, as more complex calculations are becoming cheaper than navigating across cache lines [Hennessy and Patterson, 2006].

Another jump search variation which has received little attention in recent work is the *Golomb search*. Golomb-based searches were initially proposed by Hwang and Lin [1972] as a compliment to svs intersection. Algorithm 16 describes an efficient implementation of Golomb search. The search proceeds in a manner similar to exponential searching. A straddling range r is calculated iteratively in the target set which is then used to bound a binary search. The key difference being that each iterative range search in L increases by a constant factor b instead of doubling. When iteratively searching through a set of size n_2 for the elements of a set of size n_1, the value for b is be pre-computed as $b = 2^{\lceil \log n_2/n_1 \rceil}$ [Gallager and van Voorhis, 1975].

6.3.3 Duality of Searching and Coding

There is an interesting duality between **F-SEARCH** and integer compression. For example, consider the simple intersection shown in Algorithm 10 where $n_1 = |S| \leq n_2 = |T|$, and each of the elements of the smaller set S is searched for in turn the larger set, with the search always moving forward. If we choose to implement

F-SEARCH using a full binary search, the costs per operation would be $\mathcal{O}(\log n_2)$, and $\mathcal{O}(n_1 \log n_2)$ time overall. This approach is the dual of the $\mathcal{O}(n \log u)$ cost of using a static binary code to store an n-item set over the universe $1 \ldots u$. We could also choose to implement **F-SEARCH** as a linear search such that forward traversal over d elements requires $\mathcal{O}(d)$ time. This approach is the dual of storing a set using a Unary code, which in turn is equivalent to the use of a bitvector, which requires $\mathcal{O}(u)$ bits. Other integer codes have duals. The search method of Hwang and Lin is equivalent to Golomb coding (see Witten et al. [1999], Moffat and Turpin [2002]). The Elias γ code is the dual of the exponential search mechanism of Bentley and Yao [1976], which was used as a basis for efficient **F-SEARCH** operations. Likewise, the searching method of Baeza-Yates [2004] is the dual of the interpolative coding of Moffat and Stuiver [2000], which is strikingly similar to divide and conquer techniques explored by Moffat and Port [1990] In the Elias γ code, the representation for a gap of d requires $1 + 2\lfloor \log d \rfloor = \mathcal{O}(\log d)$ bits, and a subset of n of u elements requires at most $n\,(2\log(u/n) + 1)$ bits.

6.4 Practical Indexing

In order to obtain reasonable efficiency, all of the intersection methods discussed in Section 6.3 depend on **F-SEARCH** methods with non-sequential access. However, applications which depend on large sets rarely store the information uncompressed. So, we need an approach which allows fast traversal in compressed sets of d-gaps. In order to facilitate efficient traversal, it is possible to build a partial index into the list of compressed d-gaps. Figure 6.1 shows the modified arrangement proposed. Every p th d-gap is extracted from the compressed list and added to an auxiliary array, stored as an uncompressed integer. In addition, a bit offset (or byte offset, in the case of byte code compression techniques) is also stored as satellite data in the auxiliary table.

We are now able to apply any of the **F-SEARCH** possibilities, and the **INTERSECT** operations that use them, with only minor modification. An **F-SEARCH** in the modified data structure proceeds by first searching the auxiliary array. If the value is found, it is returned. Otherwise, the **PREDECESSOR** is returned to identify the appropriate

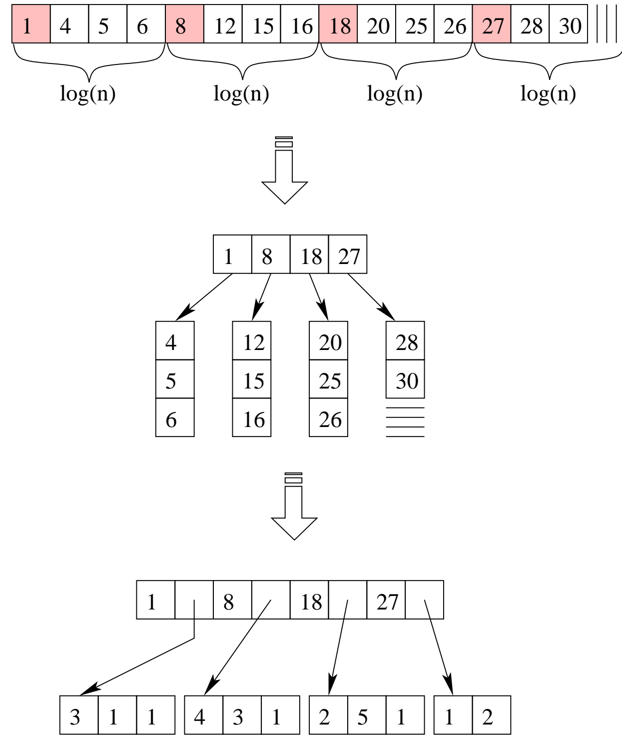


Figure 6.1: The array is initially partitioned into blocks of size $p = \lceil\log(n)\rceil$. Every p th element is then stored in an auxiliary array and the remaining elements in each block are encoded as d-gaps. In this example $p = 4$, so every fourth element from the original list (top row, shaded entries) is extracted and stored in the auxiliary array, together with a byte offset to the block of $p-1$ remaining elements. The block elements are then encoded as d-gaps and compressed (bottom representation).

sub-block. The sub-block is subsequently decompressed, and can in turn be searched using any of the available search methods, or is linearly searched as is decompressed sequentially. The cost of searching a set of n values is at most $\mathcal{O}(\log(n/p))$ for a binary search in the auxiliary index, plus $\mathcal{O}(p)$ linear decodings within the sub-block. If we assume $p = k \log n$, for some constant k, the search cost is $\mathcal{O}(\log n)$.

In order to estimate the storage cost of the new structure, suppose that a set of n elements encoded as d-gaps occupies $n \log(u/n)$ bits. Removing every p th gap multiplies that by $(p-1)/p$. Therefore, each of the n/p elements in the auxiliary array requires $\log u$ bits for the original element, and $\log(n \log(u/n)) \leq \log n + \log \log u$

bits for the access pointer. Adding all of the costs reveals a total cost of

$$\frac{p-1}{p} n \log \frac{u}{n} + \frac{n}{p} (\log u + \log n + \log \log u)$$

bits. If we again assume $p = k \log n$, this simplifies to

$$n \log \frac{u}{n} + \frac{n}{k} \left(2 + \frac{\log \log u}{\log n} \right) .$$

When $n \geq \log u$, the overhead cost of the auxiliary array is $\mathcal{O}(n/k)$ bits, with a search cost of $\mathcal{O}(k \log n)$ time. In real terms, when $k = 1$, the cost of the auxiliary index is two bits per pointer, in addition to the compressed storage cost of the index lists.

A potential drawback of the hybrid approach is that **F-SEARCH** operations over a distance of d are no longer guaranteed to take $\mathcal{O}(\log d)$ time. In the worst case, sequential decoding of a block of d-gaps can lead to $\mathcal{O}(\log n)$ time instead. From a practical standpoint, this is generally not a significant problem, as the block based approach is well suited to modern blockwise hierarchical memory models. When compared with the inline approach described by Moffat and Zobel [1996], our blocks are smaller on average and the skip pointers are maintained separately from the compressed d-gap sequence. This approach allows faster **F-SEARCH** operations, and thus faster **INTERSECT** computations at the cost of a small amount of space overhead. In recent independent work, Sanders and Transier [2007] also investigate two-level representations to improve intersection in compact sets. The main focus of their work is a variation on most significant bit tabling to create buckets of roughly uniform size. Sanders and Transier also consider the possibility of deterministic bucket sizes, in a method similar to the approach proposed here.

6.5 Experiments

This section introduces the approach used to measure the execution costs and storage efficiency for the various set representations explored experimentally. First, we

query length $\lvert q \rvert$	total queries	matches ('000)	average n_1 ('000)	average $\lvert n_i \rvert$ ('000)
2	15,517	124	338	1,698
3	7,014	78	325	5,725
4	2,678	56	348	10,311
5	1,002	41	356	14,317
6	384	27	288	15,927
7	169	15	248	17,365
8	94	10	165	17,958
9+	146	5	146	18,986

Table 6.1: Length distribution of the $27{,}004$ queries. The average query length is 2.73 terms.

describe the data collection used, and then discuss properties of the query set. We then compare and contrast the various **INTERSECT** and **F-SEARCH** combinations available in array based intersection. Finally, we will look at storage costs for various hybrid representations of sets and investigate the efficiency of **INTERSECT** in these arrangements.

6.5.1 Collection and Queries

All of the experiments discussed here are based on the GOV2 collection of $426\,$GB of web data provided by the TREC Terabyte Track (see `http://trec.nist.gov`). The total collection contains just over 25 million documents, and roughly 87 million unique words. The vocabulary and inverted lists were constructed using the freely available `Zettair` search engine (see `http://www.seg.rmit.edu.au/zettair`). For simplicity, words that appeared only once or twice were ignored, since they are easily handled within the vocabulary itself. This reduction resulted in a total of $19{,}783{,}975$ index lists being considered. Each inverted list can be represented as an ordered sequence of document numbers from 1 to $u = 25{,}205{,}181$.

The query set used for experimentation was extracted from a set supplied by Microsoft. Each query present in the list had at least one of the top three document answers present in the `.gov` domain, as of early 2005. There were a total of $27{,}004$

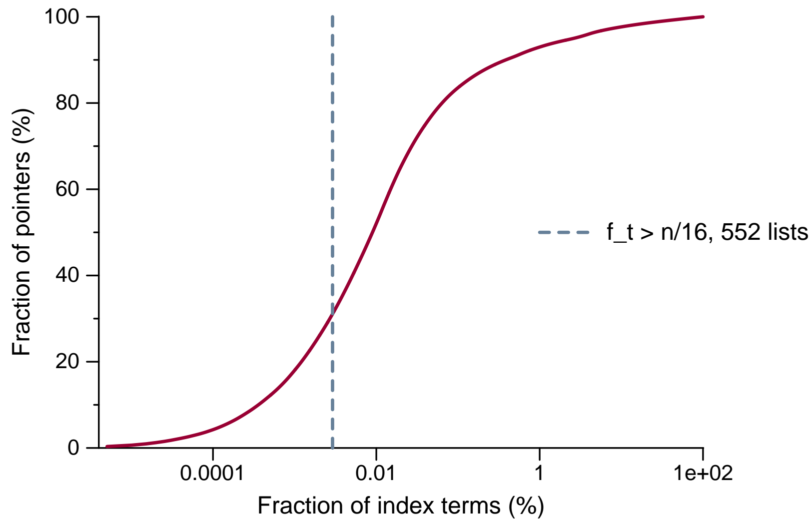


Figure 6.2: Relative costs of terms in the GOV2 data set. A small percentage of terms contain the majority of the pointers in the collection. The vertical dotted line shows that approximately 30% of the pointers in the index appear in the set of 552 inverted lists representing the most common terms.

unique queries in the final set, each of which had at least one conjunctive Boolean query match in the GOV2 collection. Table 6.1 shows the distribution of query lengths in the collection, and the range of index list lengths used in each query. The average query length tested was 2.73, which is near the expected average query length of 2.4 [Spink et al., 2001]. Note how, even in two-term queries, the most common term appears in more than 5% of the documents. In addition to showing the average number of matches for each query length, the average number of items in the most discriminating set n_1, and the average length for all sets in a query $|n_i| = (\sum_1^{|q|} n_i)/|q|$.

Figure 6.2 shows the skewed nature of term occurrences in the query text. A small percentage of terms, in this case 188 lists, represent 50% of all document pointers in the inverted index. A significant portion of these words are *stop* words such as “and” or “the”. In some instances, it can be beneficial to drop these words from the inverted

index to improve performance and reduce space overhead. However, modern systems rarely remove stop words because doing so can severely limit multi-word query support.

To carry out the experiments, the following methodology was used. First, the index lists for the $15{,}208$ unique words which appeared in the query set were extracted from the index into a separate file. The set of queries was then executed using the various intersection algorithms. To process a single query, the set of lists pertinent to the query were read into memory while the execution clock was stopped. Each query was then executed five times in a row, each time generating a candidate set of document numbers. The CPU time for each run was then added to a running total, according to the length of the query evaluated. For example, the time recorded for queries of length two is the average of $5 \times 15{,}517 = 77{,}585$ query executions. The average number of comparisons was also recorded in order to further evaluate the various **F-SEARCH** algorithms.

6.5.2 Array-Based Intersection

The first set of experiments were devised to determine how well the holistic methods perform relative to standard approaches. There are two levels of adaptivity which are possible. The **F-SEARCH** method can be modified, or the method of selecting an eliminator can be varied. For instance, we might choose the use an exponential search in the **F-SEARCH** function, rather than a worst-case optimal Golomb search, and we might choose to use the adp intersection method. Methods for enhanced eliminator selection were discussed in Section 6.3.1.

Figure 6.3 compares the average number of comparisons per query for each of the intersection methods, with the underlying set represented as an array of integers throughout. All search methods perform successive **F-SEARCH** operations over the unexplored region in each set. The first graph shows binary search combined with three holistic methods and the svs approach. The top right graph shows the results of a worst-case optimal Golomb search combined with the same intersection methods. The bottom left graph and bottom right graph again show the same experiment, with the underlying search method being exponential search and interpolative search

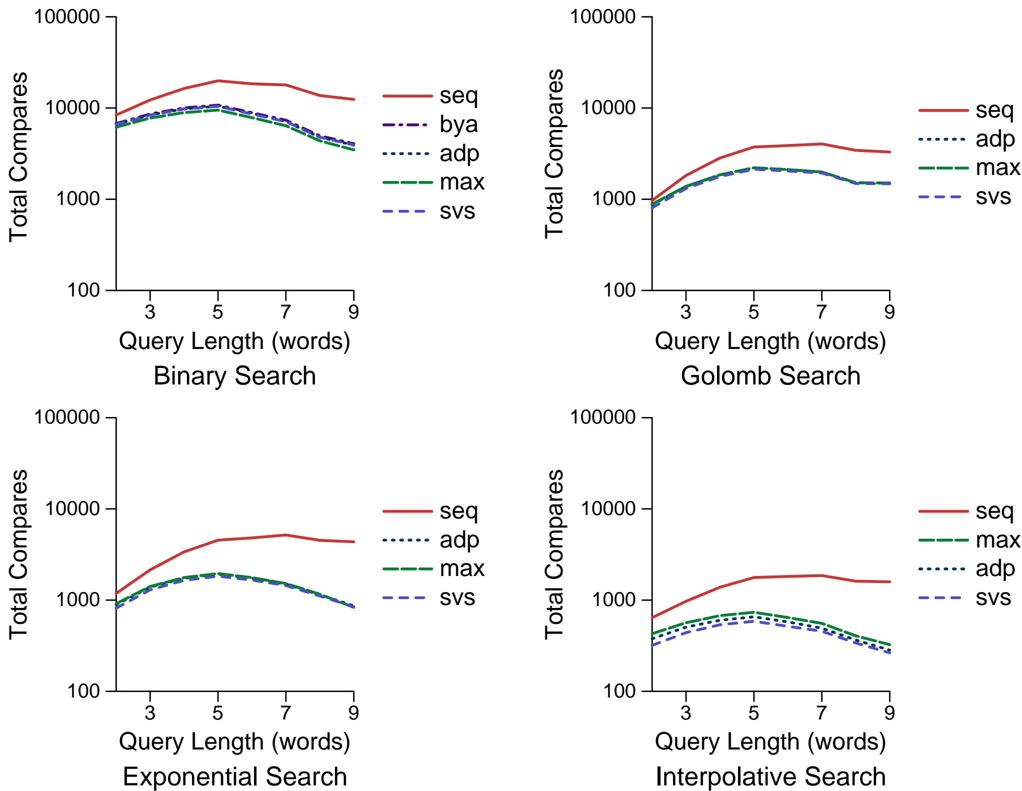


Figure 6.3: Average number of comparisons per query for different intersection algorithms with non-sequential search mechanisms, for different query lengths, in the TREC GOV2 dataset, and an array set representation.

respectively. For all possible search methods, interpolative search performs the fewest number of comparisons per query, by a large margin. The seq algorithm appears quite competitive with the svs approach when coupled with an interpolative **F-SEARCH**.

Figure 6.3 also highlights the important role adaptive search methods play in efficient **INTERSECT**. Binary search is less flexible in this regard, and as a result performs more comparisons than any of the other methods. In general, the number of comparisons performed by svs, max, and adp are tightly related, as each of these methods rely on intersecting the sets from smallest to largest, and they differ only in details.

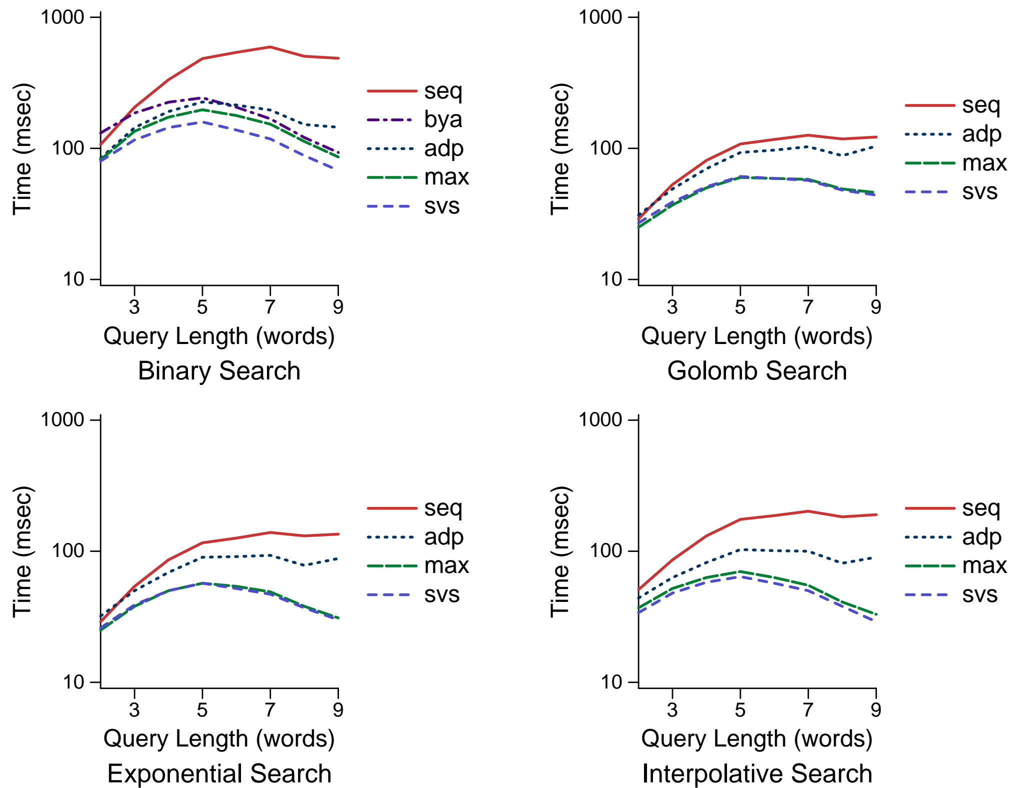


Figure 6.4: Efficiency (in CPU time) of the different intersection algorithms with non-sequential search mechanisms, for different query lengths, in the TREC GOV2 dataset, and an array set representation.

If we choose another metric to compare performance, such as the average CPU time per query, other patterns begin to emerge. In Figure 6.3, we see the same experiment, where the average CPU time is measured instead of the average number of comparisons per query. The top left graph is once again a binary search. The binary search method is surprisingly ineffective, as the method generates far more cache misses than the other methods, resulting in a noticeable performance degradation, relative to methods which exhibit localized access. Also, interpolative search performs significantly fewer comparisons on average across all **INTERSECT** methods. However, the cost of calculating each jump is far more expensive than is the case with the other approaches, making it less attractive than would be indicated by the comparison

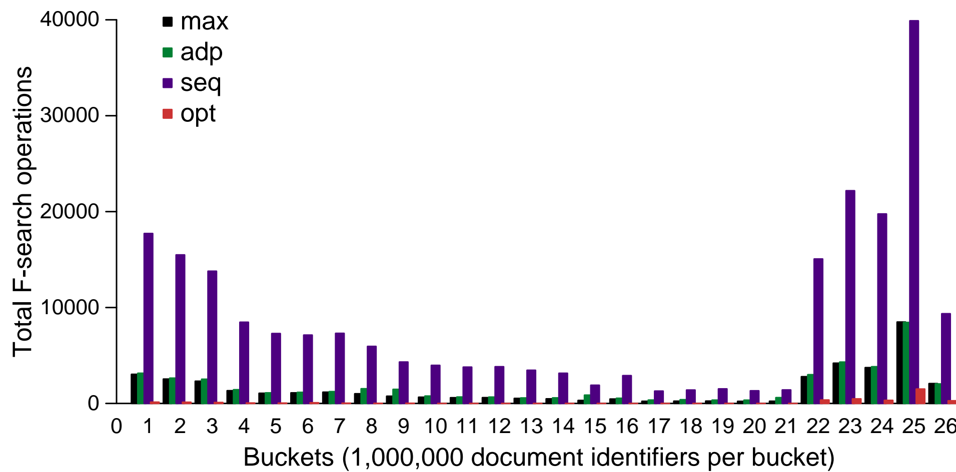


Figure 6.5: The total number of **F-SEARCH** operations initiated in per $1,000,000$ documents for each holistic search algorithm for the query "`staffing issues in the crime lab national survey of forensic`". From this comparison we can see that the max and seq algorithms initiate significantly fewer **F-SEARCH** operations across the whole search space.

counting alone. In terms of CPU performance, the svs method outperforms the multi-set approaches, as a consequence of a more tightly localized memory access pattern.

When comparing all of the graphs based on CPU performance in Figure 6.4, exponential search offers the best performance for all of the possible eliminator selection choices. Golomb and interpolative search are also quite competitive. Note that the execution time of svs-based method tends not to grow as more terms are added to the query. This is because the cost is largely determined by the frequency of the rarest element, and long queries are likely to use at least one low-frequency term. The lower left graph also shows the binary search-based method of Baeza-Yates [2004], which is adaptive by virtue of the search sequence. It operates on two sets at a time, but within those two sets has little locality of reference, when compared to the svs approach.

query length	max	adp	seq	svs
2	314,537	337,865	439,901	337,867
3	396,007	428,944	631,547	429,582
4	456,729	501,990	830,429	503,222
5	488,507	539,280	997,092	542,081
6	410,887	452,662	927,326	453,734
7	337,984	375,395	894,055	378,408
8	235,338	258,878	692,983	295,834
9+	191,113	216,114	630,750	217,608

Table 6.2: Average number of **F-SEARCH** calls for each of the search algorithms over all $27{,}004$ queries, based on query length.

Search Depth

In order to better understand the performance of the multi-way intersection methods, it is valuable to compare how effective each method is at selecting eliminators. One of the design goals of the max algorithm is to localize access to the pair of most discriminating sets whenever possible. In Figure 6.5, we see how effective the max approach is at minimizing the total number of **F-SEARCH** operations for each eliminator. In this experiment, a word query (“`staffing issues in the crime lab national survey of forensic`”) was tested against each of the intersection methods. For each eliminator selected, the total number of **F-SEARCH** operations initiated is recorded and added to a bucket relative to the eliminator value. For example, if the eliminator $12{,}111{,}010$ is selected, and generates seven **F-SEARCH** operations that are successful before it is eliminated by an eighth, then 8 is counted as part of bucket 12.

The minimal number of comparisons can be calculated for the query and is shown as opt. In order to calculate the value, the final answer list is determined and $q - 1$ comparisons are added to an appropriate bucket, based on the document identifier. An algorithm which has a perfect eliminator selection method would require an equivalent number of comparisons to validate the candidate set, as shown for the opt method. This value serves as a lower bound for a comparison based algorithm, as this is the total number of comparisons necessary simply to validate the answer set. Both max and adp initiate significantly fewer **F-SEARCH** operations than seq. However, none

Data Representation	TREC GOV2 19,783,975 words	Query Set 15,208 words
bitvector	58,051.4	44.6
integer (32-bit)	22.6	14.1
d-gaps, byte code, and auxiliary index, $k = 1$	9.5	4.9
d-gaps, byte code, and auxiliary index, $k = 2$	8.5	4.4
d-gaps, byte code, and auxiliary index, $k = 4$	8.0	4.1
d-gaps and byte code	7.4	3.8
combinatorial cost	5.9	2.7

Table 6.3: Total space cost in gigabytes to store (in the center column) all of the inverted lists, which appear in more than two documents, for the 426 GB TREC GOV2 collection, and (in the right column) the subset of the inverted lists referred to by the experimental query set.

of the algorithms are close to the best possible result of opt, suggesting more efficient approaches are possible.

Table 6.2 shows the average number of **F-SEARCH** calls initiated for all 27,004 queries, relative to query length. Again we see max, adp, and svs initiate roughly the same number of **F-SEARCH** operations. The max algorithm does a little better on average, since it occasionally selects a more discriminating eliminator from a set other than the default smallest set. When only considering an **F-SEARCH** metric, both max and adp outperform svs. However, svs is still more CPU efficient as it is performing its **F-SEARCH** operations in on list at any given stage.

6.5.3 Compressed Indexing

Table 6.3 shows the relative cost of storing the inverted lists using different representations. For example, stored as uncompressed 32-bit integers, the index requires 23 GB, compared to a combinatorial set cost (as defined in Equation 2.3 on page 12), summed over all the lists, of 6 GB. Byte codes do not attain the latter target, nevertheless they provide an attractive space saving compared to uncompressed integers, and an even greater saving compared to bitvectors. If we must store all of the lists, the cost of using a bitvector representation is prohibitively expensive. However, low frequency

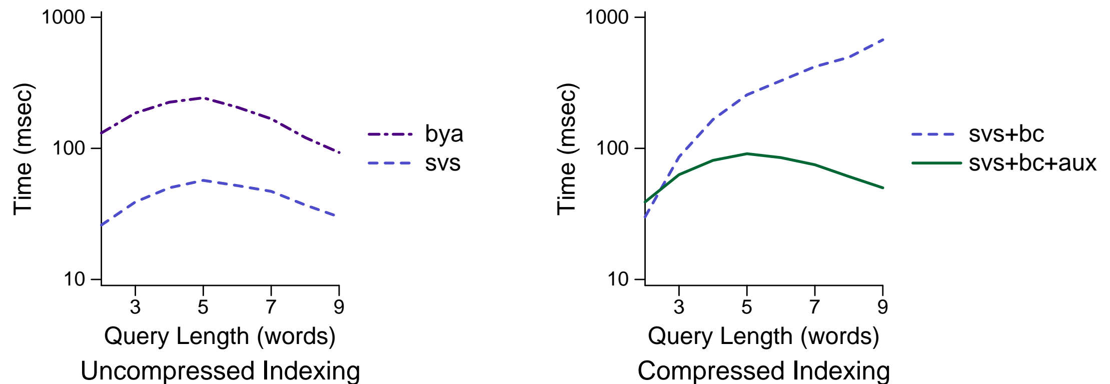


Figure 6.6: Efficiency of intersection algorithms for different query lengths in the TREC GOV2 dataset on a 2.8 Ghz Intel Xeon with 2 GB of RAM: (a) fast methods, including the use of random access, where svs is set-vs-set, using exponential search, and bya is set-vs-set using the Baeza-Yates method and binary search; and (b) two methods that use compressed data formats, where svs+bc involves sequential processing of byte codes, and svs+bc+aux makes use of byte codes indexed by an auxiliary array with $k = 2$.

terms rarely appear in the Microsoft query list and, as a result, the cost of the bitvector representation drops dramatically in the right hand column. On the other hand, the relative fractions of the compressed representations suggest that more than half of the full index is still manipulated.

Figure 6.6 compares the costs of using uncompressed binary representations in the left-hand graph with the cost of using compressed representations in the right-hand graph. Both graphs use svs methods as a baseline since they are the most efficient array-of-integer representation in practice. The left-hand graph shows the results for the uncompressed array representations with non-sequential searching methods. In the right-hand graph, both lines refer to compact index representations compressed using byte codes. In the svs+bc approach, **F-SEARCH** operations are handled via sequential access and linear search; and in the svs+bc+aux method, through the use of an auxiliary index array. Use of the auxiliary array greatly increases processing speed on long queries, and allows intersections to be handled in times very close to the uncompressed svs cost in the left-hand graph.

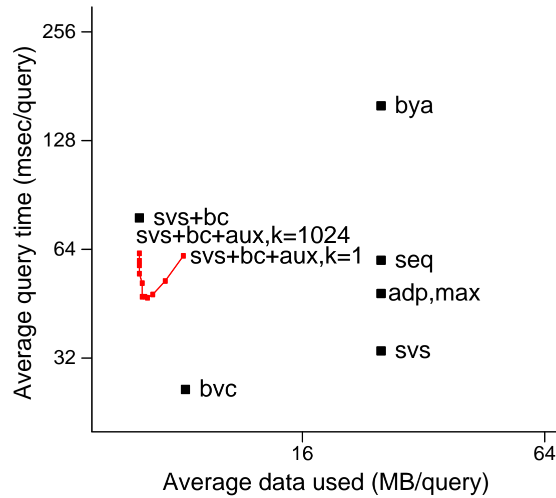


Figure 6.7: Tradeoffs between index cost and query throughput, plotted as the average amount of data processed in MB per query over all 27,004 queries, versus the average time taken in milliseconds per query.

6.5.4 Time and Space Comparisons

Figure 6.7 shows data transfer volumes plotted against query time, in both cases averaged over the full sequence of 27,004 queries. The svs+bc+aux methods, using the auxiliary array, require slightly more disk traffic than the fully-compressed svs+bc approach, but execute in as little as half the time. The indexed compressed methods are slower than the uncompressed svs+exp method, using exponential search, but the latter involves much more disk traffic.

The bvc bitvector approach provides astonishing efficiency in this query set, because most query terms are relatively common in the collection, and benefit because a large proportion of the queries are short. Bitvectors are also able to store common query terms in a small amount of space. In fact, their only real deficiency is the exorbitant storage costs for rare terms. This surprising revelation leads to a new line of attack. Is it possible to use bitvectors to store and process common query terms and use a compressed, d-gap representation to store the rare items? Would a hybrid approach taking the best of both approaches work well in practice?

Method	Bitvector terms	Size (GB)
Fully byte coded	0	7.41
Hybrid, $f_t > n/8$	188	6.97
Hybrid, $f_t > n/10$	277	7.00
Hybrid, $f_t > n/12$	367	7.07
Hybrid, $f_t > n/16$	552	7.31
Hybrid, $f_t > n/20$	775	7.67
Hybrid, $f_t > n/24$	982	8.05
Hybrid, $f_t > n/32$	1,382	8.88

Table 6.4: Total space cost in gigabytes to store an index from the 426 GB TREC GOV2 as a mixture of bitvector and byte codes. The lists are partitioned so that large lists are represented as bitvectors and smaller lists are represented as byte coded sequences of d-gaps, based on the frequency $f_t > n/\alpha$, where α is a tuning parameter.

6.5.5 Engineering a Hybrid Representation

The surprising blend of efficiency and effectiveness for the bvc bitvector approach in Figure 6.7 suggests other practical compromises are possible. Bitvectors are clearly preferred for common words, because of their superior compression effectiveness and efficient query processing. On the other hand, byte coded lists of d-gaps are more effective for shorter term lists. These observations suggest a simple partitioning approach to query processing, in which short lists are represented as byte coded lists of d-gaps, while longer lists are represented as bitvectors.

Finding the best partitioning point requires balancing the cost of storing the index and the cost of using it. The minimum length of any item in a byte coded sequence is 8 bits. Hence, if term lists which have more than $f_t > n/8$ items are coded as bitvectors, and all others using a byte code and d-gap representation, the index cannot increase in size.[1]

Table 6.4 shows the cost of compressing all of the TREC GOV2 inverted lists, using different partitioning points, with the tuning parameter $\alpha = 8$ being the best of those tested. In Figure 6.8, we see a different representation of the average processing cost

[1] One of the examiners of this thesis suggested simply taking each list and encoding as both a bitvector and a byte encoded list of d-gaps, and storing the shortest. This pragmatic approach is, broadly speaking, equivalent to the $f_t > n/8$ method described here.

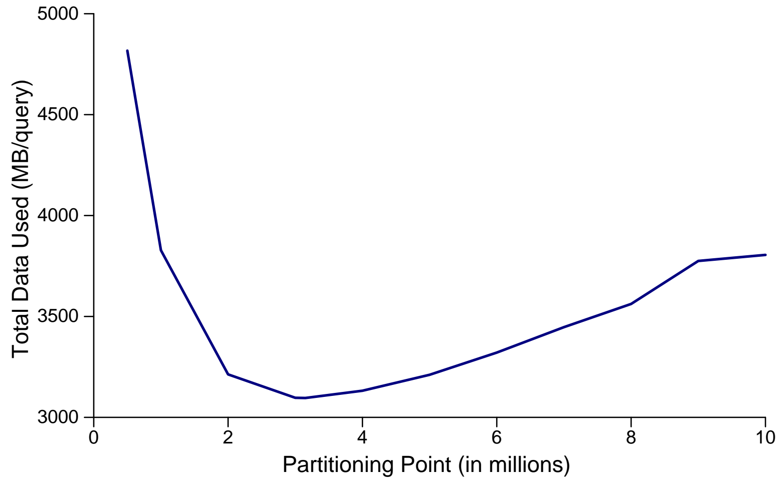


Figure 6.8: The total data used per query for the TREC GOV2 collection when using a hyb+m2 approach for varying partitions.

when the partitioning point is varied. Essentially, as f_t approaches $n/8$, the total data used is drops very quickly. Beyond this point, the total data use slowly starts to increase again.

There are a several approaches to multi-set intersection using hybrid list representations. In addition to effective compression in dense sets, bitvectors also offer constant time membership queries. In order to maximize the benefit of using the bitvectors, two hybrid multi-set intersection methods are considered here. In Hybrid Intersection Method One (hyb+m1), all byte coded and bitvector lists are intersected separately. Each remaining item from the byte code intersection is searched for in the remaining bitvector, using a **MEMBER** query. Algorithm 17 shows the first method. First, if there are any bitvector sets, intersect them to yield a candidate bitvector B. If there are no byte coded sets, then return the expanded set of document numbers derived from B. Next, intersect the byte coded sets using the svs approach, to yield a candidate set C. If there are no bitvector sets, return C. Otherwise, for each $x \in C$, if $B[x] = 1$, then x is appended to the answer set.

Algorithm 17 Hybrid Intersection Algorithm 1 (hyb+m1)

INPUT: A list of byte coded d-gaps $BCS_1 \ldots BCS_x$, and a list of bitvectors $BVS_1 \ldots BVS_y$, where $x + y = |q|$

OUTPUT: An ordered set of answers, as 32-bit integers.

1: set $A \leftarrow \{\}$
2: set $B \leftarrow BVS_1, i \leftarrow 2$
3: **while** $i \leq y$ **do**
4:    $B \leftarrow B$ **and**$_b$ BVS_i
5: **end while**
6: **if** $x = 0$ **then**
7:    return **DECODE_BITVECTOR**(B)
8: **else**
9:    $C \leftarrow$ **BC_DECODE**(BCS_1)
10:    **while** $i \leq x$ **do**
11:       $C \leftarrow$ **SVS_INTERSECT**$(C,$ **BC_DECODE**$(BCS_i))$
12:    **end while**
13:    **if** $y = 0$ **then**
14:       return C
15:    **end if**
16:    **for** each $d \in C$ **do**
17:       **if MEMBER**(B, d) **then**
18:          **APPEND**(A, d)
19:       **end if**
20:    **end for**
21: **end if**
22: return A

In Hybrid Intersection Method 2 (hyb+m2), all byte coded lists are intersected to produce a candidate list.[2]. Then, all remaining bitvector lists are iteratively searched via a **MEMBER** query to produce the final answer list. The motivation for this alternative comes from the realization that intersection of the byte coded sets should result in a small set of potential candidates, generating fewer total **MEMBER** queries. Algorithm 18 shows the modified approach. Firstly, if there are no byte coded sets,

[2]One of the examiners of this thesis suggested another variation to hyb+m2: "When you have to try a candidate (surviving the gap intersections) in the bitmaps, you get the corresponding computer words of all the bitmaps and **AND**$_b$ them into a register r. Later you check the presence of your bit in r. If the next position x' to check is still within the same computer word, you have r already computed and don't have to intersect again. This is not at all unlikely because of the possible clustering of symbols. Even if it never happens, doing the **AND**$_b$ might be faster, because of the pipeline, than a sequence of if's". It might be interesting to explore this alternative at a future time; here we retain the arrangement that was proposed in Moffat and Culpepper [2007]

Algorithm 18 Hybrid Intersection Algorithm 2 (hyb+m2)

INPUT: A list of byte coded d-gaps $BCS_1 \ldots BCS_x$, and a list of bitvectors $BVS_1 \ldots BVS_y$, where $x + y = |q|$
OUTPUT: An ordered set of answers, as 32-bit integers.

```
 1: set A ← { }
 2: if x = 0 then
 3:   set B ← BVS_1, i ← 2
 4:   while i ≤ y do
 5:     B ← B and_b BVS_i
 6:   end while
 7:   return DECODE_BITVECTOR(B)
 8: else
 9:   C ← BC_DECODE(BCS_1)
10:   while i ≤ x do
11:     C ← SVS_INTERSECT(C, BC_DECODE(BCS_i))
12:   end while
13:   if y = 0 then
14:     return C
15:   end if
16:   while i ≤ y do
17:     set B ← BVS_i
18:     for each d ∈ C do
19:       if MEMBER(B, d) then
20:         APPEND(A, d)
21:       end if
22:     end for
23:     set C ← A
24:   end while
25: end if
26: return A
```

then the bitvector sets are intersected to yield a candidate bitvector B which, in turn, is expanded to a set of document numbers derived from B and returned. Next, any byte coded sets are intersected using the svs approach, to yield a candidate set C. Now, for each remaining bitvector set, each element in $x \in C$ is checked against the current bitvector, and retained in C (via an intermediary A) only if $B_i[x] = 1$. Finally, when all bitvector terms have been processed, the answer set A is returned.

Motivation for the proposed hybrid approaches is reinforced when considering the percentage of long lists relative to short lists with varying query length. Figure 6.9

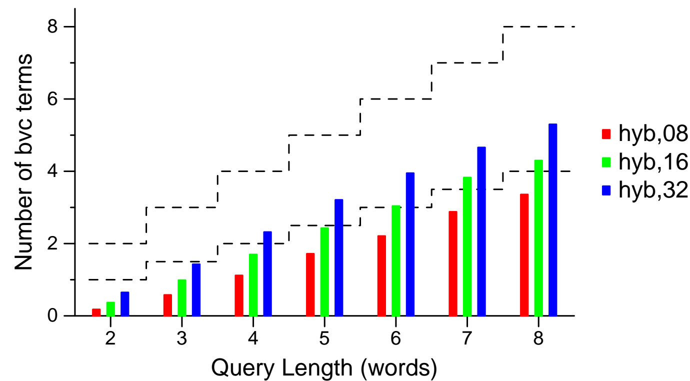


Figure 6.9: A comparison of compressed term representations when varying the query length for different partitioning values. The upper dashed line shows the total number of terms in the query and the lower dashed line shows half of the terms. When $\alpha = 32$, more than half of the terms in queries of length three or more are represented as bitvectors.

shows the fraction of byte coded lists, relative to the number of bitvector lists when varying query size and partitioning value. If a partitioning value $\alpha = 32$ is used, less than half of all lists are byte encoded for all queries longer than 3 words. In fact, more than half of all terms in queries longer than 3 are composed of only $1,382$ words, which are more efficiently handled as bitvectors.

Figure 6.10 shows the efficiency of the hyb+m1 method for three different partitioning thresholds $\alpha = \{8, 16, 32\}$. The results for the svs+lin+aux,k=4 method and the pure bitvector approach bvc are also shown for comparison. All three versions outperform the auxiliary array method, and are competitive with the pure bitvector approach, with $\alpha = 32$ being most efficient for all query lengths.

Figure 6.11 shows the efficiency of the hyb+m2 method for three different partitioning thresholds $\alpha = \{8, 16, 32\}$. The results for the svs+lin+aux,k=4 method and the pure bitvector approach bvc are again shown for comparison. The hyb+m2 method performs significantly better than the hyb+m1 method over all tested query lengths and partitioning permutations. Additionally, the hybrid methods offer more compression effectiveness than using straight byte coded lists, pure bitvector approaches, or auxiliary

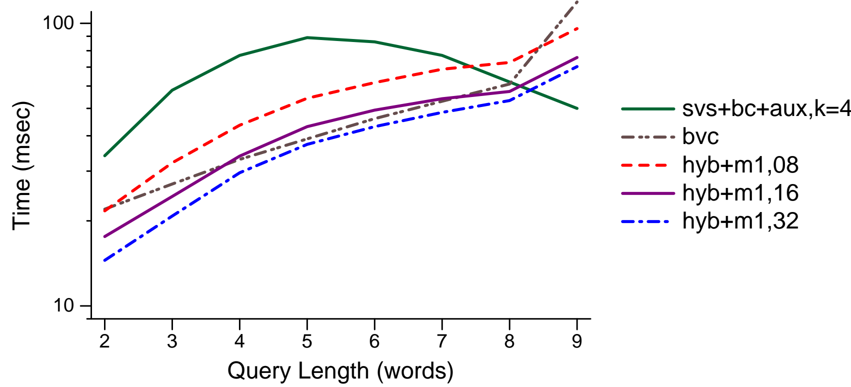


Figure 6.10: Efficiency of the hyb+m1 algorithm and three different partitioning thresholds $\alpha = \{8, 16, 32\}$. For $\alpha = 32$, the hybrid approach is faster than the pure bitvector approach, shown as a solid black line here, for all tested query lengths.

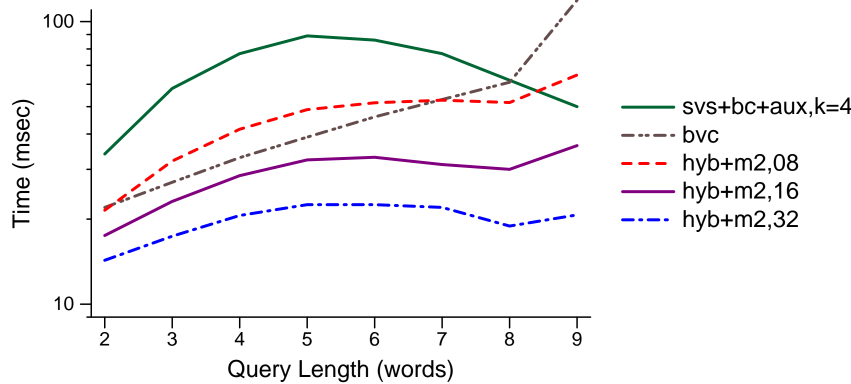


Figure 6.11: Efficiency of the hyb+m2 algorithm and three different partitioning thresholds $\alpha = \{8, 16, 32\}$. For $\alpha = 16$, the hybrid approach is faster than the pure bitvector approach, shown as a solid black line here, for all tested query lengths.

array representations when $\alpha < 16$.

Figure 6.12 conclusively shows the remarkable performance efficiency and compression effectiveness for hyb+m2. This simple intersection technique is nearly twice as fast as the traditional svs array-of-integer technique, while processing only 25% as much data, reinforcing the pivotal role compression, and the engineering trade-offs that are possible when implementing it, has in improving algorithm performance.

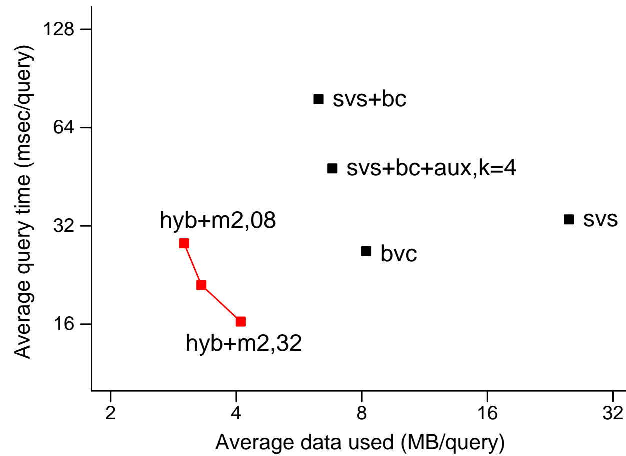


Figure 6.12: Space versus CPU costs when using hybrid bitvector representations for conjunctive Boolean queries.

6.6 Summary

This chapter has shown that data compression can be used to greatly enhance the performance of various set operations. If flexible coding methods which allow selective decompression are coupled with these approaches, space overheads can be dramatically reduced, while still maintaining efficient set intersection support. In fact, simple hybrid representations can dramatically reduce the storage and processing costs of massive data sets, such as the TREC GOV2 collection, while supporting set operations, such as **INTERSECT** twice as fast as their uncompressed counterparts. Future research may produce additional space-efficient data structures which exploit the inherent parallelism of bitvectors, but limit the excessive space requirements in sparse sets, in a single, unified set representation.

Chapter 7

Conclusions

In this thesis we have shown the key role data compression serves in designing space-efficient algorithms. Processing massive datasets and extracting relevant information remains a fundamental research problem in computer science, as well as interdisciplinary research areas such as molecular biology. Virtually every area of scientific research is affected in some manner by aspects of practical algorithmic design. Compression is arguably the most versatile method in use today which can bridge the gap between generalist algorithms which are asymptotically efficient and their practical counterparts. We conclusively show that a combination of careful algorithmic design, which exploits system constraints and leverages byte-aligned codes, consistently produces solutions which are more efficient and more effective than their traditional algorithmic counterparts.

Exploiting the Hardware

One of the key ingredients to designing practical algorithms relies on an intimate understanding of the *hardware* which processes the information. The most promising computational models are beginning reflect this reality. For instance, cache-oblivious and external memory models are becoming the dominant theoretical frameworks. They more accurately reflect limitations in real systems than their pure abstract counterparts, and encourage algorithms which can be evaluated empirically. These

models lead to higher quality research, and more accurately portrays the scientific method. However, there are few detailed descriptions of systematic enhancements which can be consistently used to improve algorithmic design. Here, we describe several techniques which have widespread applications across all areas of algorithmics:

- Tune algorithms to take maximum advantage of time and space trade-offs. The relationship between time and space is not necessarily linear, meaning small increases in processing time can lead to large reductions in space usage, and vice versa. Measurable gains are often possible by using simple techniques such as pre-computation, lazy evaluation, incremental computation, and batch processing.
- Tune algorithms for the expected case.
- Know your hardware. Leverage algorithms which support locality of reference to maximize cache performance. Use strength reduction, word-parallelism, memory interleaving, and pipelining whenever possible. Set ordering and block-based tabling are simple and effective solutions which exploit this principle.
- Use models which facilitate a finite universe assumption. For example, produce integer intermediate representations of bounded size early in the processing phase in order to reap maximum benefit from efficient representations such as bitvectors.
- Minimize the amount of information which must be processed. For instance, data compression allows more information to be kept close to the processor, which often leads to better performance.
- Relax algorithmic constraints. It is often possible to trade certainty for time or space if approximate solutions are acceptable. This is the basis of significant algorithmic improvements in everything from lossy compression to bloom filters.

It is rare to find instances where all of these principles can be applied consistently. It is equally as rare to find instances where none of these principles are applicable. The

key to success is understanding the problem at hand, developing a formal description of what must be accomplished, and devising a solution which is based on sound algorithmic engineering. Data compression is an underutilized technique in algorithm design which spans several of these algorithmic design principles: use it whenever possible.

Why Alignment Matters

The success of byte-aligned codes can be attributed to one key property: length discovery without decoding. This allows them to be easily integrated into sophisticated data structures and algorithms with little impact or effort. Furthermore, subunits of bytes are more favorable to modern hardware optimizations and programming language designs. These properties promote flexibility and algorithm reuse. The ability to reuse classic algorithms for pattern matching as well as dictionary-based data structures as building blocks to solve complex problems make byte codes a sound practical choice. No other coding algorithm has attained the level of flexibility byte codes possess to date.

Thesis Summary

This thesis presents novel approaches for efficient data representation of sequences and sets. First, a new approach to byte-aligned coding, called restricted prefix byte coding is presented, which is more effective and more efficient than other prelude-based byte coding variants. Along the same lines, we also investigate new approaches to maximize efficieny in prelude representations. When combined with byte-aligned coding methods, these new prelude approaches significantly improve coding efficiency, without reducing overall effectiveness. Next, new algorithms for searching directly in compressed sequences are developed which directly leverage unique properties of various byte-aligned codes.

Finally, we devise several new approaches to the classic set intersection problem. Here, we presented a set intersection algorithm, called Max Successor, which is

competitive with other recently introduced holistic intersection algorithms. We then propose two additional approaches which leverage compression to further improve efficiency: auxiliary arrays and d-gap+bitvector hybrids. These succinct intersection algorithms show a significant efficiency boost over traditional integer-based set intersection methods, and have widespread applicability in current text retrieval systems. Together, these contributions accentuate the fundamental role data compression has in designing practical space-efficient data structures.

Future Directions

Opportunities still exist to design alternative codes which implicitly store codeword lengths or block offsets. These codes are highly sought after as a basic foundation to space-efficient algorithms. There are also few examples of compressed, cache-oblivious algorithms which allow efficient secondary memory usage. Another path to explore is the use of approximation in compact sets and sequences. There are few examples of compressed pattern matching solutions which tightly integrate approximate string searching and efficient coding techniques. More disciplined approaches to length limited subset mappings, such as the ideas developed for semi-dense prelude transmission, also appear promising.

Finally, we close with an open problem of significant interest: Is there a *practical* data representation which can support the operations **INSERT**, **DELETE**, and **MEMBER**, using $\mathcal{O}(1)$ time and minimum space (in an information theoretic sense) in terms of the number of items in the set, instead of the maximum size of a bounded universe? In other words, does a simple dictionary representation which uses $\mathcal{O}(\lceil \log \binom{u}{n} \rceil)$ bits of space, with no lower order terms or hidden constant factors, exist? Can the representation be extended to support ordering, and, as a consequence, produce new routes to supporting **F-SEARCH**, **SUCCESSOR**, or **RANGE** in $\mathcal{O}(1)$ time?

Bibliography

J. Åberg. *A universal source coding perspective on PPM*. PhD thesis, Lund University, Sweden, October 1999.

J. Åberg, Y. M. Shtarkov, and B. J. M. Smeets. Towards understanding and improving escape probabilities in PPM. In J. A. Storer and M. Cohn, editors, *Proceedings of the 7th IEEE Data Compression Conference (DCC 1997)*, pages 22–31, Los Alamitos, California, March 1997. IEEE Computer Society Press.

M. I. Abouelhoda, E. Ohlebusch, and S. Kurtz. Optimal exact string matching based on suffix arrays. In A. H. F. Laender and A. L. Oliveira, editors, *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE 2002)*, volume 2476 of *LNCS*, pages 31–43. Springer, September 2002.

J. Adiego and P. de la Fuente. Mapping words into codewords on PPM. In F. Crestani, P. Ferragina, and M. Sanderson, editors, *Proceedings of the 13th International Symposium on String Processing and Information Retrieval (SPIRE 2006)*, volume 4209 of *LNCS*, pages 181–192. Springer, October 2006.

A. V. Aho and M. J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, June 1975.

C. Allauzen and M. Raffinot. Factor oracle of a set of words. Technical Report 99-11, Institut Gaspard-Monge, Université de Marne-la-Vallée, 1999.

C. Allauzen, M. Crochemore, and M. Raffinot. Efficient experimental string matching by weak factor recognition. In A. Amir and G. M. Landau, editors, *Proceedings of the 12th*

Annual Symposium on Combinatorial Pattern Matching, number 2089 in LNCS, pages 51–72. Springer, July 2001.

A. Amir and G. Benson. Efficient two-dimensional compressed matching. In J. A. Storer and M. Cohn, editors, *Proceedings of the 2nd IEEE Data Compression Conference (DCC 1992)*, pages 279–288, Los Alamitos, California, March 1992. IEEE Computer Society Press.

A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, April 1996.

A. Andersson. Sublogarithmic searching without multiplication. In *Proceedings of the 36th Annual IEE Symposium on Foundations of Computer Science (FOCS 1995)*, pages 655–665. IEEE Computer Society, October 1995.

A. Andersson and M. Thorup. Tight(er) worst-case bounds on dynamic searching and priority queues. In *Proceedings of the 32rd Annual ACM Symposium on Theory of Computing (STOC 2000)*, pages 335–442. ACM Press, May 2000.

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

V. N. Anh and A. Moffat. Improved word-aligned binary compression for text indexing. *IEEE Transactions on Knowledge and Data Engineering*, 18(6):857–861, June 2006.

V. N. Anh and A. Moffat. Compressed inverted files with reduced coding overheads. In W. B. Croft, A. Moffat, C. J. van Rijsbergen, R. Wilkinson, and J. Zobel, editors, *Proceedings of the 21st International ACM Conference on Research and Development in Information Retrieval (SIGIR 1998)*, pages 291–298. ACM Press, August 1998.

G. Antoshenkov. Byte-aligned bitmap compression. Technical report, Oracle Corporation, 1994. U.S. Patent Number 5,363,098.

G. Antoshenkov and M. Ziauddin. Query processing and optimization in Oracle RDB. *The VLDB Journal*, 5:229–237, 1996.

A. Apostolico and A. S. Fraenkel. Robust transmission of unbounded strings using Fibonacci representations. *IEEE Transactions on Information Theory*, IT-33(2):238–245, March 1987.

K. Asanovic, R. Bodik, B. C. Catanzaro, K. K. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, K. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report EECS-2006-183, University of California at Berkeley, December 2006.

R. Baeza-Yates and B. Ribeiro-Neto. *Modern information retrieval.* Addison Wesley, first edition, 1999.

R. A. Baeza-Yates. A fast set intersection algorithm for sorted sequences. In S. C. Sahinalp, S. Muthukrishnan, and U. Dogrusöz, editors, *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM 2004)*, volume 3109 of *LNCS*, pages 400–408. Springer, July 2004.

R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

R. A. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2): 127–158, 1999.

J. Barbay and C. Kenyon. Adaptive intersection and t-threshold problems. In D. Eppstein, editor, *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 390–399, January 2002.

J. Barbay, A. López-Ortiz, and T. Lu. Faster adaptive set intersections for text searching. In C. Àlvarez and M. J. Serna, editors, *Proceedings of the 5th International Workshop on Experimental Algorithms (WEA 2006)*, volume 4007 of *LNCS*, pages 146–157, May 2006.

J. Barbay, M. He, J. I. Munro, and S. S. Rao. Succinct indexes for strings, binary relations and multi-labeled trees. In N. Bansal, K. Pruhs, and C. Stein, editors, *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2007)*, pages 680–689. SIAM, January 2007.

L. A. Barroso, J. Dean, and U. Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 23(2):22–28, 2003.

R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173 – 189, 1973.

P. Beame and F. E. Fich. Optimal bounds for the predecessor problem and related problems. *Journal of Computer and System Sciences*, 65(1):38–72, August 2002.

T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

T. C. Bell, A. Moffat, C. G. Nevill-Manning, I. H. Witten, and J. Zobel. Data compression in full-text retrieval systems. *Journal of the American Society for Information Science*, 44(9): 508–531, October 1993.

D. Benoit, E. D. Demaine, J. I. Munro, and V. Raman. Representing trees of higher degree. In F. K. H. A. Dehne, A. Gupta, J.-R. Sack, and R. Tamassia, editors, *Proceedings of the 6th International Workshop on Algorithms and Data Structures (WADS 1999)*, volume 1663 of *LNCS*, pages 169–180. Springer, August 1999.

J. Bentley and A. C-C. Yao. An almost optimal algorithm for unbounded searching. *Information Processing Letters*, 5(3):82–87, August 1976.

J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, 1986.

D. Blandford and G. Blelloch. Index compression through document reordering. In J. A. Storer and M. Cohn, editors, *Proceedings of the 12th IEEE Data Compression Conference (DCC 2002)*, pages 342–351, Los Alamitos, California, March 2002. IEEE Computer Society Press.

D. K. Blandford and G. E. Blelloch. Compact representations of ordered sets. In J. I. Munro, editor, *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 11–19, January 2004.

R. E. Bleier. Treating hierarchical data structures in the SDC time-shared data management system (TDMS). In *Proceedings of the 22nd National Conference of the ACM*, pages 41–49, 1967.

G. E. Blelloch, B. M. Maggs, and S. L. M. Woo. Space-efficient finger search on degree-balanced search trees. In *Proceedings of the 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2003)*, pages 374–383. SIAM, January 2003.

A. Blumer, J. Blumer, D. Haussler, A. Ehrenfeucht, M. T. Chen, and J. Seiferas. The smallest automaton recognizing a subword. *Theoretical Computer Science*, 40:31–55, 1985.

A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM*, 34(3):578–595, July 1987.

P. Boldi and S. Vigna. Compressed perfect embedded skip lists for quick inverted-index lookups. In M. Consens and G. Navarro, editors, *Proceedings of the 12th International Conference on String Processing and Information Retrieval (SPIRE 2005)*, volume 3772 of *LNCS*, pages 25–28, November 2005.

A. Bookstein and S. T. Klein. Using bitmaps for medium sized information retrieval systems. *Information Processing & Management*, 26:525–533, 1990.

A. Bookstein and S. T. Klein. Flexible compression for bitmap sets. In J. A. Storer and J. H. Reif, editors, *Proceedings of the 1st IEEE Data Compression Conference (DCC 1991)*, pages 402–410, Los Alamitos, California, March 1991a. IEEE Computer Society Press.

A. Bookstein and S. T. Klein. Compression of correlated bit-vectors. *Information Systems*, 16 (4):387–400, 1991b.

A. Bookstein and S. T. Klein. Generative models for bitmap sets with compression applications. In A. Bookstein, Y. Chiaramella, G. Salton, and V. V. Raghavan, editors, *Proceedings of the 14th Annual International Conference on Research and Development in Information Retrieval (SIGIR 1991)*, pages 63–71. ACM Press, 1991c.

A. Bookstein and S. T. Klein. Models of bitmap generation: A systematic approach to bitmap compression. *Information Processing & Management*, 28(6):735–748, 1992.

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

N. R. Brisaboa, A. Fariñna, G. Navarro, and M. F. Esteller. (S, C)-dense coding: An optimized compression code for natural language text databases. In M. A. Nascimento, editor, *Proceedings of the 10th International Symposium on String Processing and Information Retrieval*, volume 2857 of *LNCS*, pages 122–136, October 2003a.

N. R. Brisaboa, E. L. Iglesias, G. Navarro, and J. Paramá. An efficient compression code for text databases. In F. Sebastiani, editor, *Proceedings of the 25th European Conference on Information Retrieval Research*, volume 2633 of *LNCS*, pages 468–481, April 2003b.

N. R. Brisaboa, A. Fariña, G. Navarro, and J. Paramá. Simple, fast, and efficient natural language adaptive compression. In A. Apostolico and M. Melucci, editors, *Proceedings of the 11th International Symposium on String Processing and Information Retrieval*, volume 3246 of *LNCS*, pages 230–241, October 2004.

N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. Efficiently decodable and searchable natural language adaptive compression. In R. A. Baeza-Yates, N. Ziviani, G. Marchionini, A. Moffat, and J. Tait, editors, *Proceedings of the 28th Annual International Conference on Research and Development in Information Retrieval (SIGIR 2005)*, pages 234–241, New York, August 2005. ACM Press.

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

A. Brodnik and J. Munro. Membership in constant time and almost-minimum space. *SIAM Journal on Computing*, 28(5):1627–1640, 1999.

H. Buhrman, P. B. Miltersen, J. Radhakrishnan, and S. Venkatesh. Are bitvectors optimal? *SIAM Journal on Computing*, 31(6):1723–1744, 2002.

S. Bunton. Semantically motivated improvements for PPM variants. *Computer Journal*, 40 (2/3):76–93, 1997.

S. Bunton. *On-line stochastic processes in data compression*. PhD thesis, University of Washington, March 1996.

M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation, Palo Alto, California, May 1994.

J. L. Carter and M. N. Wigman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

B. Chapin. Switching between two on-line list update algorithms for higher compression of Burrows-Wheeler transformed text. In J. A. Storer and M. Cohn, editors, *Proceedings of*

the 10th IEEE Data Compression Conference (DCC 2000), pages 183–192, Los Alamitos, California, March 2000. IEEE Computer Society Press.

D. Chen, Y.-J. Chiang, N. Memon, and X. Wu. Optimal alphabet partitioning for semi-adaptive coding of sources of unknown sparse distributions. In J. A. Storer and M. Cohn, editors, *Proceedings of the 13th IEEE Data Compression Conference (DCC 2003)*, pages 372–381, Los Alamitos, California, 2003. IEEE Computer Society Press.

G. Chen, S. J. Puglisi, and W. F. Smyth. Fast and practical algorithms for computing all the runs in a string. In B. Ma and K. Zhang, editors, *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM 2007)*, volume 4580 of *LNCS*, pages 307–315. Springer, June 2007.

Y. Choueka, S. T. Klein, and Y. Perl. Efficient variants of Huffman codes in high level languages. In *Proceedings of the 8th Annual International Conference on Research and Development in Information Retrieval (SIGIR 1985)*, pages 122–130, New York, June 1985. ACM Press.

Y. Choueka, A. S. Fraenkel, S. T. Klein, and E. Segal. Improved hierarchical bitvector compression in document retrieval systems. In F. Rabitti, editor, *Proceedings of the 9th Annual International Conference on Research and Development in Information Retrieval (SIGIR 1986)*, pages 88–97. ACM Press, 1986.

Y. Choueka, A. S. Fraenkel, S. T. Klein, and E. Segal. Improved techniques for processing queries in full-text systems. In C. T. Yu and C. J. V. Rijsbergen, editors, *Proceedings of the 10th Annual International Conference on Research and Development in Information Retrieval (SIGIR 1987)*, pages 306–315. ACM Press, 1987.

Y. Choueka, A. S. Fraenkel, and S. T. Klein. Compression of concordances in full-text retrieval systems. In Y. Chiaramella, editor, *Proceedings of the 11th Annual International Conference on Research and Development in Information Retrieval (SIGIR 1988)*, pages 597–612. ACM Press, 1988.

D. Clark. *Compact PAT trees.* PhD thesis, University of Waterloo, 1996.

D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 1996)*, pages 383–391, January 1996.

J. G. Cleary and W. J. Teahan. Unbounded length contexts for PPM. *Computer Journal*, 40 (2/3):67–75, 1997.

J. G. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, COM-32(2):306–315, April 1984.

R. Cole. On the dynamic finger conjecture for splay trees. Part II: The proof. *SIAM Journal on Computing*, 30(1):44–85, 2000.

R. Cole, B. Mishra, J. Schmidt, and A. Siegel. On the dynamic finger conjecture for splay trees. Part I: Splay sorting $\log n$ block sequences. *SIAM Journal on Computing*, 30(1):1–43, 2000.

J. Connell. A Huffman-Shannon-Fano code. *Proceedings of IEEE*, 61(7):1046–1047, July 1973.

G. V. Cormack and N. S. Horspool. Data compression using dynamic Markov modelling. *Computer Journal*, 30(6):541–550, 1987.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, second edition, 2001.

M. Crocehmore and R. Vérin. On compact directed acyclic word graphs. In J. Mycielski, G. Rozenberg, and A. Salomaa, editors, *Structures in Logic and Computer Science*, volume 1261 of *LNCS*, pages 192–211. Springer, 1997.

M. Crochemore. Transducers and repetitions. *Theoretical Computer Science*, 45:63–86, 1986.

M. Crochemore and W. Rytter. *Jewels of stringology*. World Scientific Publishing Co. Pte. Ltd., Singapore, first edition, 2002.

M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.

M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings*. Cambridge University Press, Cambridge, United Kingdom, first edition, 2007.

J. S. Culpepper and A. Moffat. Enhanced byte codes with restricted prefix properties. In M. Consens and G. Navarro, editors, *Proceedings of the 12th International Conference on*

String Processing and Information Retrieval (SPIRE 2005), volume 3772 of *LNCS*, pages 1–12, November 2005.

J. S. Culpepper and A. Moffat. Phrase-based searching in compressed text. In F. Crestani, P. Ferragina, and M. Sanderson, editors, *Proceedings of the 13th International Symposium on String Processing and Information Retrieval (SPIRE 2006)*, volume 4209 of *LNCS*, pages 337–345. Springer, October 2006.

J. S. Culpepper and A. Moffat. Compact set representation for information retrieval. In N. Ziviani and R. Baeza-Yates, editors, *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE 2007)*, volume 4726 of *LNCS*, pages 124–135. Springer, October 2007.

Z. J. Czech, G. Havas, and B. S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1-2):1–143, 1997.

E. S. de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, 2000.

S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6): 391–407, 1990.

E. D. Demaine, A. López-Ortiz, and J. I. Munro. Adaptive set intersections, unions, and differences. In *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2000)*, pages 743–752, January 2000.

S. Deorowicz. Improvements to Burrows-Wheeler compression algorithm. *Software Practice and Experience*, 30(13):1465–1483, November 2000.

S. Deorowicz. Second step algorithms in Burrows-Wheeler compression algorithm. *Software Practice and Experience*, 32(2):99–111, November 2002.

L. Devroye and P. Morin. Cuckoo hashing. Further analysis. *Information Processing Letters*, 86(4):215–219, 2002.

M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In W. Kuich, editor, *Proceedings of the 19th International Colloquium on*

Automata, Languages, and Programming (ICALP 1992), volume 623 of *LNCS*, pages 235–246. Springer, July 1992.

M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4): 738–761, 1994.

M. Dietzfelbinger, T. Hagerup, and J. Katajainen M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *Journal of Algorithms*, 25(1):19–51, 1997.

A. I. Dumey. Indexing for rapid random access memory systems. *Computers and Automation*, 5(12):6–9, 1956.

M. Effros. PPM performance with BWT complexity: A new method for lossless data compression. In J. A. Storer and M. Cohn, editors, *Proceedings of the 10th IEEE Data Compression Conference (DCC 2000)*, pages 203–212, Los Alamitos, California, March 2000. IEEE Computer Society Press.

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

A. P. Ershov. On programming of arithmetic operations. *Doklady Akademii Nauk SSSR*, 118 (3):427–430, 1958.

S. Even and M. Rodeh. Economical encoding of commas between strings. *Communications of the ACM*, 21(4):315–317, April 1978.

C. Faloutsos and H. V. Jagadish. Hybrid index organizations for text databases. In A. Pirotte, C. Delobel, and G. Gottlob, editors, *Proceedings of the International Conference on Extending Database Technology*, volume 580 of *LNCS*, pages 310–327, 1992.

R. M. Fano. The transmission of information. Technical Report 65, Massachusetts Institute of Technology, 1949.

M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20(4):388–404, 1998.

A. Fariña. *New compression codes for text databases*. PhD thesis, Universidade de Coruña, April 2005.

P. Fenwick. Universal codes. In K. Sayood, editor, *Lossless Compression Handbook*, pages 55–78, Boston, 2003. Academic Press. ISBN 0-12-620861-1.

P. Fenwick. A new data structure for cumulative frequency tables. *Software Practice and Experience*, 24(3):327–336, March 1994.

P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *Proceedings of the 41st IEEE Annual Symposium on Foundatations of Computer Science (FOCS 2000)*, pages 390–398. IEEE Computer Society Press, November 2000.

P. Ferragina and G. Manzini. Compression boosting in optimal linear time using the Burrows-Wheeler transform. In J. I. Munro, editor, *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 655–663. SIAM, January 2004.

P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 53(4):552–581, 2005.

P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. An alphabet-friendly FM-index. In A. Apostolico and M. Melucci, editors, *Proceedings of the 11th International Conference on String Processing and Information Retrieval (SPIRE 2004)*, volume 3246 of *LNCS*, pages 150–160, October 2004.

J. Fischer, V. Heun, and S. Kramer. Fast frequent mining using suffix arrays. In *Proceedings of the 5th IEEE International Conference on Data Mining (ICDM 2005)*, pages 609–612. IEEE Computer Society, November 2005.

A. S. Fraenkel and S. T. Klein. Novel compression of sparse bit-strings—Preliminary report. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words, Volume 12*, NATO ASI Series F, pages 169–183, Berlin, 1985. Springer.

A. S. Fraenkel and S. T. Klein. Bounding the depth of search trees. *The Computer Journal*, 36 (7):668–678, 1993.

A. S. Fraenkel and S. T. Klein. Robust universal complete codes for transmission and compression. *Discrete Applied Mathematics*, 64(1):31–55, January 1996.

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

M. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with $o(1)$ worst case access time. *Journal of the ACM*, 31:538–544, 1984.

M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *Journal of Computer and System Sciences*, 47(3):424–436, December 1993.

K. Fredriksson and S. Grabowski. A general compression algorithm that supports fast searching. *Information Processing Letters*, 100(6):226–232, 2006.

K. Fredriksson and G. Navarro. Average-optimal single and multiple approximate string matching. *ACM Journal of Experimental Algorithmics*, 9:1.4–1 – 1.4–47, 2004.

K. Fredriksson and J. Tarhio. Processing of Huffman compressed texts with a super-alphabet. In M. A. Nascimento, E. S. de Moura, and A. L. Oliveira, editors, *Proceedings of the 10th International Symposium on String Processing and Information Retrieval (SPIRE 2003)*, volume 2857 of *LNCS*, pages 108–121, October 2003.

K. Fredriksson and J. Tarhio. Efficient string matching in Huffman compressed texts. *Fundamenta Informaticae*, 62(1):1–16, 2004.

M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In P. Beame, editor, *Proceedings of the 40th Symposium on Foundations of Computer Science (FOCS 1999)*, pages 285–298. IEEE Computer Society, October 1999.

P. Gage. A new algorithm for data compression. *The C Users Journal*, 12(2):23–38, February 1994.

R. G. Gallager and D. C. van Voorhis. Optimal source codes for geometrically distributed integer alphabets. *IEEE Transactions on Information Theory*, IT-21(2):228–230, March 1975.

R. F. Geary, R. Raman, and V. Raman. Succinct ordinal trees with level-ancestor queries. In J. I. Munro, editor, *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, pages 1–10, January 2004.

P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. In J. M. Abello and J. S. Vitter, editors, *External Memory Algorithms*, volume 50 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 39–70. AMS, 1999.

E. N. Gilbert and E. F. Moore. Generalized Kraft inequality and arithmetic coding. *Bell Systems Technical Journal*, 38:933–967, July 1959.

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

A. Golynski. Optimal lower bounds for rank and select indexes. In M. Bugliesi, B. Preneel, V. Sassone, and I. Wegener, editors, *Proceedings of the 33th International Colloquium on Automata, Languages, and Programming (ICALP 2006) Part 1*, volume 4051 of *LNCS*, pages 370–381. Springer, July 2006.

A. Golynski, R. Grossi, A. Gupta, R. Raman, and S. S. Rao. On the size of succinct indices. In L. Arge, M. Hoffmann, and E. Welzl, editors, *Proceedings of the 15th Annual European Symposium on Algorithms (ESA 2007)*, volume 4698 of *LNCS*, pages 371–382. Springer, October 2007.

G. H. Gonnet, L. D. Rogers, and J. A. George. An algorithmic and complexity analysis of iterpolation search. *Acta Informatica*, 13(1):39–52, January 1980.

R. Grossi and G. F. Italiano. Suffix trees and their applications in string algorithms. In R. Baeza-Yates and N. Ziviani, editors, *Proceedings of the 1st South American Workshop on String Processing (WSP 1993)*, pages 57–76, September 1993.

R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *Proceedings of the 32rd Annual ACM Symposium on Theory of Computing (STOC 2000)*, pages 397–406. ACM Press, May 2000.

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. A preliminary version appeared in STOC 2000.

D. A. Grossman and O. Frieder. *Information Retrieval: Algorithms and Heuristics.* Springer, second edition, 2004.

L. Guibas, J. Hershberger, D. Leven, M. Sharir, and R. Tarjan. Linear time algorithms for visibility and shortest path problems inside simple polygons. *Algorithmica*, 2:209–233, 1987.

A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In A. Ellis and T. Hagino, editors, *Proceedings of the 14th International World Wide Web Conference (WWW05)*, pages 902–903. ACM, May 2005.

A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed dictionaries: Space measures, data sets, and experiments. In C. Àlvarez and M. J. Serna, editors, *Proceedings of the 5th International Workshop on Experimental Algorithms (WEA 2006)*, volume 4007 of *LNCS*, pages 158–169, May 2006.

D. Gusfield. *Algorithms on Strings, Trees, and Sequences.* Cambridge University Press, New York, New York, USA, 1997.

T. Hagerup. Sorting and searching on the word RAM. In M. Morvan, C. Meinel, and D. Krob, editors, *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1998*, volume 1373 of *LNCS*, pages 366–398. Springer, February 1998.

D. Hankerson, G. A. Harris, and P. D. Johnson Jr. *Introduction to Information Theory and Data Compression*. Chapman and Hall/CRC, Boca Raton, Florida, second edition, 2003.

M. He, J. I. Munro, and S. S. Rao. Succinct ordinal trees based on tree covering. In L. Arge, C. Cachin, T. Jurdzinski, and A. Tarlecki, editors, *Proceedings of the 34th International Colloquium on Automata, Languages, and Programming (ICALP 2007)*, volume 4596 of *LNCS*, pages 509–520. Springer, July 2007.

H. S. Heaps. *Information Retrieval - Computational and Theoretical Aspects.* Academic Press, 1978.

J. L. Hennessy and D. A. Patterson. *Computer architecture: A quantitative approach*. Morgan Kaufmann, San Francisco, CA, fourth edition, 2006.

D. Hirschberg and D. Lelewer. Efficient decoding of prefix codes. *Communications of the ACM*, 33(4):449–459, April 1990.

R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10(6): 501–506, 1980.

R. N. Horspool and G. V. Cormack. Comments on "A locally adaptive data compression scheme". *Communications of the ACM*, 30(9):792–793, September 1987.

P. G. Howard and J. S. Vitter. Practical implementations of arithmentic coding. In J. A. Storer, editor, *Image and Text Compression*, pages 85–112. Kluwer Academic, 1992.

S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

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

E. Hunt, M. P. Atkinson, and R. W. Irving. A database index to large biological sequences. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB 2001)*, pages 139–148. Morgan Kaufmann, September 2001.

F. K. Hwang and S. Lin. A simple algorithm for merging two disjoint linearly ordered list. *SIAM Journal on Computing*, 1:31–39, 1972.

H. Hyyrö. Explaining and extending the bit-parallel approximate string matching algorithm of Myers. Technical Report A-2001-10, Department of Computer and Information Systems, University of Tampere, Tampere, Finland, 2001.

H. Hyyrö, K. Fredriksson, and G. Navarro. Increased bit-parallelism for approximate and multiple string matching. *ACM Journal of Experimental Algorithmics*, 10:2.6–1 – 2.6–27, 2006.

S. Inenaga, H. Hoshino, A. Shinohara, M. Takeda, S. Arikawa, G. Mauri, and G. Pavesi. On-line construction of compact directed acyclic word graphs. *Discrete Applied Mathematics*, 146(2):156–179, 2005.

G. Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, 1988.

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

M. Jakobsson. Huffman coding in bit-vector compression. *Information Processing Letters*, 7 (6):304–307, October 1978.

J. Jansson, K. Sadakane, and W.-K. Sung. Ultra-succinct representation of ordered trees. In N. Bansal, K. Pruhs, and C. Stein, editors, *Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2007)*, pages 575–584. SIAM, January 2007.

G. C. Jewell. Text compression for information retrieval systems. *IEEE SMC Newsletter*, 5, February 1976.

J. Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In Z. Galil and E. Ukkonen, editors, *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM 1995)*, volume 937 of *LNCS*, pages 191–204. Springer, July 1995.

J. Kärkkäinen and E. Sutinen. Lempel-Ziv index for q-grams. *Algorithmica*, 22(1):137–154, 1998.

J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In J.-Y. Cai and C. K. Wong, editors, *Proceedings of the Second Annual International Conference on Computing and Combinatorics (COCOON 1996)*, volume 1090 of *LNCS*, pages 219–230. Springer, June 1996.

J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching in Ziv-Lempel compressed text. *Journal of Discrete Algorithms*, 1(3-4):313–338, 2003.

K. Kärkkäinen and P. Sanders. Simple linear work suffix array construction. In J. C. M Baeten, J. K. Lenstra, K. Parrow, and G. J. Woeginger, editors, *Proceedings of the 30th International Colloquium on Automata, Languages, and Programming (ICALP 2003)*, volume 2719 of *LNCS*, pages 943–955. Springer, June 2003.

J. Katajainen, A. Moffat, and A. Turpin. A fast and space-economical algorithm for length-limited coding. In J. Staples, P. Eades, N. Katoh, and A. Moffat, editors, *Proceedings of the International Symposium on Algorithms and Computation*, volume 1004 of *LNCS*, pages 12–21. Springer-Verlag, December 1995.

T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in lzw compressed text. In J. A. Storer and M. Cohn, editors, *Proceedings of the 8th IEEE Data Compression Conference (DCC 1998)*, pages 103–112, Los Alamitos, California, March 1998. IEEE Computer Society Press.

T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Shift-and approach to pattern matching in LZW compressed text. In M. Crochemore and M. Paterson, editors, *Proceedings of the 10th International Symposium on Combinatorial Pattern Matching (CPM 1999)*, volume 1645 of *LNCS*, pages 1–13. Springer, July 1999.

D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In R. Baeza-Yates, E. Chávez, and M. Crochemore, editors, *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, volume 2676 of *LNCS*, pages 186–199. Springer, June 2003.

S. T. Klein. Skeleton trees for the efficient decoding of huffman encoded texts. *Information Retrieval*, 3(1):7–23, January 2000.

S. T. Klein. Efficient recompression techniques for dynamic full-text retrieval. In E. A. Fox, P. Ingwersen, and R. Fidel, editors, *Proceedings of the 18th Annual International Conference on Research and Development in Information Retrieval (SIGIR 1995)*, pages 39–47, New York, July 1995. ACM Press.

S. T. Klein and D. Shapira. Pattern matching in Huffman encoded text. In J. A. Storer and M. Cohn, editors, *Proceedings of the 11th IEEE Data Compression Conference (DCC 2001)*, pages 449–458, Los Alamitos, California, March 2001. IEEE Computer Society Press.

D. E. Knuth. Dynamic Huffman coding. *Journal of Algorithms*, 6:163–180, 1985.

D. E. Knuth. *The art of computer programming: Sorting and searching*, volume 3. Addison-Wesley, Reading, MA, USA, second edition, 1998.

D. E. Knuth, Jr. J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.

P. Ko and S. Aluru. Space-efficient linear-time construction of suffix arrays. In R. Baeza-Yates, E. Chávez, and M. Crochemore, editors, *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching (CPM 2003)*, volume 2676 of *LNCS*, pages 200–210. Springer, June 2003.

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

P. Ko and S. Aluru. Suffix tree applications in computation biology. In S. Aluru, editor, *Handbook of Computational Molecular Biology*, chapter 6, pages 6–1 – 6–27. Chapman & Hall/CRC, 2006.

L. G. Kraft. A device for quantizing, grouping, and coding amplitude modulated pulses. Master's thesis, MIT, 1949.

P. Kumar. Cache oblivious algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*, pages 193–212, Berlin Heidelberg, 2003. Springer-Verlag.

S. Kurtz. Reducing the space requirement of suffix trees. *Software Practice and Experience*, 29 (13):1149–1171, 1999.

J. Larsson and A. Moffat. Offline dictionary-based compression. *Proc. IEEE*, 88(11):1722–1732, November 2000.

V. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

V. E. Levenshtein. On the redundancy and delay of separable codes for the natural numbers. *Problems of Cybernetics*, 20:173–179, 1968.

M. Liddell and A. Moffat. Decoding prefix codes. *Software Practice and Experience*, 35(15): 1687–1710, 2006.

M. Liddell and A. Moffat. Incremental calculation of minimum-redundancy length-restricted codes. *IEEE Transactions on Communications*, 55(3):427–435, March 2007.

M. V. Mahoney. Adaptive weighing of context models for lossless data compression. Technical Report CS-2005-16, Florida Institute of Technology, 2005.

V. Mäkinen. Compressed suffix array. In R. Giancarlo and D. Sankoff, editors, *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM 2000)*, volume 1848 of *LNCS*, pages 305–319. Springer, June 2000.

V. Mäkinen and G. Navarro. Compressed compact suffix arrays. In S. C. Sahinalp, S. Muthukrishnan, and U. Dogrusöz, editors, *Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching (CPM 2004)*, volume 3109 of *LNCS*, pages 420–433. Springer, July 2004.

U. Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Transactions on Information Systems*, 5(2):124–136, April 1997. A preliminary version appeared in CPM 1994.

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

B. B. Mandelbrot. An information theory of the statistical structure of language. In W. Jackson, editor, *Communication Thory*. Academic Press, New York, 1953.

M. E. Maron and J. L. Kuhns. On relevance, probabilistic indexing and information retrieval. *Journal of the ACM*, 7(3):216–244, July 1960.

T. Matsumoto, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Bit-parallel approach to approximate string matching in compressed text. In *Proceedings of the 7th International Symposium on String Processing and Information Retrieval (SPIRE 2000)*, volume 2476 of *LNCS*, pages 221–228. Springer, September 2000.

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

B. McMillan. Two inequalities implied by unique decipherability. *IEEE Transactions on Information Theory*, IT-2(4):115–116, 1956.

R. L. Milidiú, E. S. Laber, and A. A. Pessoa. Bounding the compression loss of the FGK algorithm. *Journal of Algorithms*, 32(2):195–211, 1999.

P. B. Miltersen. Lower bounds on the size of selection and rank indexes. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2005)*, pages 11–12. SIAM, January 2005.

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

A. Moffat. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, 38(11):1917–1921, November 1990.

A. Moffat. An improved data structure for cumulative probability tables. *Software Practice and Experience*, 29(7):647–659, November 1999.

A. Moffat and J. S. Culpepper. Hybrid bitvector index compression. In *Proceedings of the 12th Australasian Document Computing Symposium (ADCS 2007)*, pages 25–31, December 2007.

A. Moffat and Y. Isal. Word-based text compression using the Burrows-Wheeler transform. *Information Processing and Management*, 41:1175–1192, 2005.

A. Moffat and G. S. Port. Splay tree melding: Theme and variations. In *Proceedings of the 13th Australasian Computer Science Conference*, pages 275–284. Monash University, February 1990.

A. Moffat and L. Stuiver. Binary interpolative coding for effective index compression. *Information Retrieval*, 3(1):25–47, July 2000.

A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Kluwer Academic Publisher, Boston, first edition, 2002.

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

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

A. Moffat and J. Zobel. Coding for compression in full-text retrieval systems. In J. A. Storer and M. Cohn, editors, *Proceedings of the 2nd IEEE Data Compression Conference (DCC 1992)*, pages 72–81, Los Alamitos, California, March 1992a. IEEE Computer Society Press.

A. Moffat and J. Zobel. Parameterised compression for sparse bitmaps. In N. J. Belking, P. Ingwersen, and A. M. Pejtersen, editors, *Proceedings of the 15th Annual International ACM Conference on Research and Development in Information Retrieval (SIGIR 1992)*, pages 274–285, New York, June 1992b. ACM Press.

A. Moffat and J. Zobel. Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379, 1996.

A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3):256–294, 1998. A preliminary version appeared in DCC 1995.

J. I. Munro. Tables. In V. Chandru and V. Vinay, editors, *Proceedings of the 16th Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, volume 1180 of *LNCS*, pages 37–42. Springer, December 1996.

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

J. I. Munro and S. S. Rao. Succinct representation of data structures. In D. P. Mehta and S. Sahni, editors, *Handbook of data structures and algorithms*, chapter 37, pages 1–22. CRC Press, 2005.

J. I. Munro, V. Raman, and S. S. Rao. Space efficient suffix trees. *Journal of Algorithms*, 39(2): 205–222, 2001a. A preliminary version appeared in FSTTCS 1998.

J. I. Munro, V. Raman, and A. J. Storm. Representing dynamic binary trees succinctly. In *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2001)*, pages 529–536, January 2001b.

E. W. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM*, 46(3):395–415, 1999.

G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1): 31–88, 2001.

G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1): 2–1 – 2–61, 2007.

G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics*, 5(4):4, 2000. URL `http://www.jea.acm.org`.

G. Navarro and M. Raffinot. *Flexible pattern matching in strings*. Cambridge University Press, Cambridge, United Kingdom, first edition, 2002.

G. Navarro and M. Raffinot. Practical and flexible pattern matching over Ziv-Lempel compressed text. *Journal of Discrete Algorithms*, 2(3):347–371, 2004.

G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In R. Giancarlo and D. Sankoff, editors, *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM 2000)*, volume 1848 of *LNCS*, pages 166–180. Springer, June 2000.

G. Navarro and J. Tarhio. LZgrep: A Boyer-Moore string matching tool for Ziv-Lempel compressed text. *Software Practice and Experience*, 35(12):1107–1130, 2005.

M. Nelson and J. Gailly. *The Data Compression Book.* M&T Books, New York, second edition, 1995.

C. G. Nevill-Manning and I. H. Witten. On-line and off-line heuristics for inferring hierarchies of repetitions in sequences. *Proceedings of IEEE*, 88(11):1745–1755, November 2000.

C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *Computer Journal*, 40(2/3):103–116, 1997.

R. Pagh. Low redundancy in static dictionaries with constant time query. *SIAM Journal on Computing*, 31(2):353–363, 2001.

R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004. A preliminary version appeared in ESA 2001.

R. Pascoe. *Source coding algorithms for fast data compression.* PhD thesis, Stanford University, 1979.

W. W. Peterson. Addressing for random-access storage. *IBM Journal of Research and Development*, 1(2):130–146, 1957.

J. M. Ponte and W. B. Croft. A language modeling approach to information retrieval. In W. B. Croft, A. Moffat, C. J. van Rijsbergen, R. Wilkinson, and J. Zobel, editors, *Proceedings of the 21th Annual International Conference on Research and Development in Information Retrieval (SIGIR 1998)*, pages 275–281. ACM Press, August 1998.

W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

S. J. Puglisi, W. F. Smyth, and A. H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Computing Surveys*, 39(2):4–1 – 4–31, 2007.

S. Rajasekaran. Algorithms for motif search. In S. Aluru, editor, *Handbook of Computational Molecular Biology*, chapter 37, pages 37–1 – 37–21. Chapman & Hall/CRC, 2006.

R. Raman and S. S. Rao. Succinct dynamic dictionaries and trees. In J. C. M. Baeten, J. K. Lenstra, and G. J. Woeginger, editors, *Proceedings of the 30th International Colloquium*

on Automata, Languages and Programming (ICALP 2003), volume 2719 of *LNCS*, pages 357–368. Springer, June 2003.

R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In D. Eppstein, editor, *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002)*, pages 233–242. Society for Industrial and Applied Mathematics, January 2002.

V. Raman and S. S. Rao. Static dictionaries supporting rank. In A. Aggarwal and C. P. Rangan, editors, *Proceedings of the 10th International Symposium on Algorithms and Computation (ISAAC 1999)*, volume 1741 of *LNCS*, pages 18–26. Springer, December 1999.

S. S. Rao. Time-space trade-off for compressed suffix arrays. *Information Processing Letters*, 82(6):307–311, 2002.

J. Rautio, J. Tanninen, and J. Tarhio. String matching with stopper encoding and code splitting. In A. Apostolico and M. Takeda, editors, *Proceedings of the 13th Annual Symposium on Combinatorial Pattern Matching (CPM 2002)*, volume 2373 of *LNCS*, pages 42–51. Springer, July 2002.

R. F. Rice. Some practical universal noiseless coding techniques. Technical Report JPL-PUB-79-22, Jet Propulsion Laboratory, Pasadena, California, March 1979.

J. Rissanen and G. G. Langdon. Arithmetic coding. *IBM Journal Research and Development*, 23(2):149–162, 1979.

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

S. Ristov and E. Laporte. Ziv Lempel compression of huge natural language data tries using suffix arrays. In M. Crochemore and M. Patterson, editors, *Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching (CPM 1999)*, volume 1645 of *LNCS*, pages 196 – 211. Springer, July 1999.

S. E. Robertson and K. Sparck Jones. Relevance weighting of search terms. *Journal of the American Society for Information Science*, 27:129–146, 1976.

F. Rubin. Experiments in text file compression. *Communications of the ACM*, 19(11):617–623, November 1976.

B. Y. Ryabko. Comments on "A locally adaptive data compression scheme". *Communications of the ACM*, 30(9):792–794, September 1987.

W. Rytter. Grammar compression, LZ-encodings, and string algorithms with implicit input. In J. Diaz, J. Karhumäki, A. Lepistö, and D. Sannella, editors, *Proceedings of the 31th International Colloquium on Automata, Languages, and Programming (ICALP 2004)*, volume 3142 of *LNCS*, pages 15–27. Springer, July 2004.

W. Rytter. Algorithms on compresed strings and arrays. In J. Pavelka, G. Tel, and M. Bartoek, editors, *Proceedings of the 26th Conference on Current Trends in Theory and Practice of Informatics (SOFSEM 1999)*, volume 1725 of *LNCS*, pages 48–65, November 1999.

K. Sadakane. Compressed text databases with efficient query algorithms based on the compressed suffix array. In D. T. Lee and S.-H. Teng, editors, *Proceedings of the 11th International Symposium on Algorithms and Computation (ISAAC 2000)*, volume 1969 of *LNCS*, pages 410–421. Springer, December 2000.

K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003. A preliminary version appeared in ISAAC 2000.

K. Sadakane and R. Grossi. Squeezing succinct data structures into entropy bounds. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2006)*, pages 1230–1239. ACM Press, January 2006.

D. Salomon. *Data Compression: The Complete Reference*. Springer, London, fourth edition, 2007.

G. Salton and M. E. Lesk. Computer evaluation of indexing and text processing. *Journal of the ACM*, 15(1):8–36, January 1968.

G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, November 1975.

P. Sanders and F. Transier. Intersection in integer inverted indices. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*, January 2007. To appear.

K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann, San Francisco, third edition, March 2006.

M. Schindler. A fast block-sorting algorithm for lossless data compression. In J. A. Storer and M. Cohn, editors, *Proceedings of the 7th IEEE Data Compression Conference (DCC 1997)*, page 469, Los Alamitos, California, March 1997. IEEE Computer Society Press.

F. Scholer, H. E. Williams, J. Yiannis, and J. Zobel. Compression of inverted indexes for fast query evaluation. In M. Beaulieu, R. Baeza-Yates, S. H. Myaeng, and K. Järvelin, editors, *Proceedings of the 25th Annual International Conference on Research and Development in Information Retrieval (SIGIR 2002)*, pages 222–229, New York, August 2002. ACM Press.

E. J. Schuegraf. Compression of large inverted files with hyperbolic term distribution. *Information Processing & Management*, 12:377–384, 1976.

E. S. Schwartz and B. Kallick. Generating a cononical prefix encoding. *Communications of the ACM*, 7(3):166–169, March 1964.

R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

P. H. Sellers. The theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793, June 1974.

Dr. Seuss. *Fox in socks*. Random House, first edition, 1965. Written by T. Geisel.

J. Seward. On the performance of BWT sorting algorithms. In J. A. Storer and M. Cohn, editors, *Proceedings of the 10th IEEE Data Compression Conference (DCC 2000)*, page 173, Los Alamitos, California, March 2000. IEEE Computer Society Press.

J. Seward. Space-time tradeoffs in the inverse B-W transform. In J. A. Storer and M. Cohn, editors, *Proceedings of the 11th IEEE Data Compression Conference (DCC 2001)*, page 439, Los Alamitos, California, March 2001. IEEE Computer Society Press.

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

D. Shapira and A. Daptardar. Adapting the Knuth-Morris-Pratt algorithm for pattern matching in Huffman encoded texts. *Information Processing and Management*, 42(2):429–439, 2006.

Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa. Boyer-Moore type algorithm for compressed pattern matching. In R. Giancarlo and D. Sankoff, editors, *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM 2000)*, volume 1848 of *LNCS*, pages 181–194. Springer, June 2000.

D. Shkarin. PPM: One step to practicality. In J. A. Storer and M. Cohn, editors, *Proceedings of the 12th IEEE Data Compression Conference (DCC 2002)*, pages 202–211, Los Alamitos, California, March 2002. IEEE Computer Society Press.

A. Siemiński. Fast decoding of the Huffman codes. *Information Processing Letters*, 26(5): 237–241, May 1988.

F. Silvestri, S. Orlando, and R. Perego. Assigning indentifiers to documents to enhance the clustering property of fulltext indexes. In M. Sanderson, K. Järvelin, J. Allan, and P. Bruza, editors, *Proceedings of the 27th Annual International Conference on Research and Development in Information Retrieval (SIGIR 2004)*, pages 305–312. ACM Press, 2004.

A. K. Singhal. *Term weighting revisited.* PhD thesis, Cornell University, Ithica, NY, USA, January 1997.

D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3): 652–686, 1985.

M. Snyderman and B. Hunt. The myriad virtues of text compression. *Datamation*, 16:36–40, 1970.

A. Spink, D. Wolfram, B. J. Jansen, and T. Saracevic. Searching the web: The public and their queries. *Journal of the American Society for Information Science*, 52(3):226–234, 2001.

H. Stone. Parallel querying of large databases: A case study. *Computer*, 20(10):11–21, October 1987.

J. A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, Rockville, Maryland, first edition, 1988.

D. M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8): 132–142, 1990.

W. J. Tehan. *Modelling English Text*. PhD thesis, University of Waikato, New Zealand, May 1998.

J. Teuhola. A compression method for clustered bit-vectors. *Information Processing Letters*, 7 (6):308–311, October 1978.

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

A. Turpin and A. Moffat. Housekeeping for prefix coding. *IEEE Transactions on Communications*, 48(4):622–628, April 2000. Source code available from `http://www.cs.mu.oz.au/~alistair/mr_coder`.

A. Turpin and A. Moffat. Fast file search using text compression. In *Proceedings of the 20th Australasian Computer Science Conference (ACSC 1997)*, pages 1–8, February 1997.

E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms*, 6(1–3):132–137, 1985.

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

P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, June 1977.

J. van Leeuwen. On the construction of Huffman trees. In *Proceedings of the 3rd International Colloquium on Automata, Languages, and Programming (ICALP 1976)*, pages 382–410, Edinburgh University, Scotland, July 1976. Edinburgh University.

J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, June 2001.

J. S. Vitter. Design and analysis of dynamic Huffman codes. *Journal of the ACM*, 34(4):825–845, 1987.

J. S. Vitter and P. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Handbook of theoretical computer science, volume A: Algorithms and complexity (A)*, chapter 9, pages 431–524. Elsevier and MIT Press, 1990.

R. Wan. *Browsing and Searching Compressed Documents*. PhD thesis, University of Melbourne, December 2003.

P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, pages 1–11, Los Alamitos, California, October 1973. IEEE Computer Society Press.

T. A. Welch. A technique for high performance data compression rate coding. *IEEE Computer*, 17(6):8–20, 1984.

D. E. Willard. Examining computational geometry, van Emde Boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal on Computing*, 29(3):1030–1049, 2000.

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

F. Willems, Y. Shtarkov, and T. Tjalkens. Context tree weighting method: Basic properties. *IEEE Transactions on Information Theory*, 32(3):526–532, May 1995.

F. Willems, Y. Shtarkov, and T. Tjalkens. Context weighting for general finite context sources. *IEEE Transactions on Information Theory*, 33(5):1514–1520, September 1996.

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

A. I. Wirth. Symbol-driven compression of burrows wheeler transformed text. Master's thesis, University of Melbourne, September 2000.

A. I. Wirth and A. Moffat. Can we do without ranks in Burrows-Wheeler transform compression? In J. A. Storer and M. Cohn, editors, *Proceedings of the 11th IEEE Data Compression Conference (DCC 2001)*, pages 419–428, Los Alamitos, California, March 2001. IEEE Computer Society Press.

I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–541, June 1986.

I. H. Witten, T. C. Bell, and C. G. Nevill. Models for compression in full-text retrieval systems. In J. A. Storer and J. H. Reif, editors, *Proceedings of the 1st IEEE Data Compression Conference (DCC 1991)*, pages 23–32, Los Alamitos, California, March 1991. IEEE Computer Society Press.

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

K. Wu, E. J. Otoo, and A. Shoshani. A performance comparison of bitmap indexes. In H. Paques, L. Lui, and D. Grossman, editors, *Proceedings of the 10th International Conference on Information and Knowledge Management (CIKM 2001)*, pages 559–561, New York, November 2001. ACM Press.

S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35 (10):83–91, 1992.

C. Zhai and J. D. Lafferty. A study of smoothing methods for language models applied to ad hoc information retrieval. In W. B. Croft, D. J. Harper, D. H. Kraft, and J. Zobel, editors, *Proceedings of the 24th Annual International Conference on Research and Development in Information Retrieval (SIGIR 2001)*, pages 334–342. ACM Press, September 2001.

G. K. Zipf. *Human behavior and the principle of least effort*. Addison-Wesley, Reading, Massachusetts, USA, 1949.

J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, 1977.

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

J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, 38(2): 6–1 – 6–56, 2006.

J. Zobel and A. Moffat. Exploring the similarity space. *ACM SIGIR Forum*, 32(1):18–34, 1998.

J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full-text database systems. In L.Y. Yuan, editor, *Proceedings of the 18th International Conference on Very Large Databases (VLDB 1992)*, pages 352–362. Morgan Kaufmann, August 1992.