Transform Based and Search Aware Text Compression Schemes and Compressed Domain Text Retrieval

by

Nan Zhang
B.S. Beijing Colloge of Economics, 1990
M.S. National University of Singapore, 1998

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the School of Computer Science
in the College of College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Spring Term
2005

Major Professor:
Amar Mukherjee

Abstract

In recent times, we have witnessed an unprecedented growth of textual information via the Internet, digital libraries and archival text in many applications. While a good fraction of this information is of transient interest, useful information of archival value will continue to accumulate. We need ways to manage, organize and transport this data from one point to the other on data communications links with limited bandwidth. We must also have means to speedily find the information we need from this huge mass of data. Sometimes, a single site may also contain large collections of data such as a library database, thereby requiring an efficient search mechanism even to search within the local data. To facilitate the information retrieval, an emerging ad hoc standard for uncompressed text is XML which preprocesses the text by putting additional user defined metadata such as DTD or hyperlinks to enable searching with better efficiency and effectiveness. This increases the file size considerably, underscoring the importance of applying text compression. On account of efficiency (in terms of both space and time), there is a need to keep the data in compressed form for as much as possible.

Text compression is concerned with techniques for representing the digital text data in alternate representations that takes less space. Not only does it help conserve the storage space for archival and online data, it also helps system performance by requiring less number

of secondary storage (disk or CD Rom) accesses and improves the network transmission bandwidth utilization by reducing the transmission time. Unlike static images or video, there is no international standard for text compression, although compressed formats like .zip, .gz, .Z files are increasingly being used. In general, data compression methods are classified as lossless or lossy. Lossless compression allows the original data to be recovered exactly. Although used primarily for text data, lossless compression algorithms are useful in special classes of images such as medical imaging, finger print data, astronomical images and data bases containing mostly vital numerical data, tables and text information. Many lossy algorithms use lossless methods at the final stage of the encoding stage underscoring the importance of lossless methods for both lossy and lossless compression applications.

In order to be able to effectively utilize the full potential of compression techniques for the future retrieval systems, we need efficient information retrieval in the compressed domain. This means that techniques must be developed to search the compressed text without decompression or only with partial decompression independent of whether the search is done on the text or on some inversion table corresponding to a set of key words for the text.

In this dissertation, we make the following contributions:

- Star family compression algorithms: We have proposed an approach to develop a reversible transformation that can be applied to a source text that improves existing algorithm's ability to compress. We use a static dictionary to convert the English words into predefined symbol sequences. These transformed sequences create additional context information that is superior to the original text. Thus we achieve some compression at the preprocessing stage. We have a series of transforms which improve the performance. Star transform requires a static dictionary for a certain size. To avoid the considerable complexity of conversion, we employ the ternary tree data structure that efficiently converts the words in the text to the words in the star dictionary in linear time.

- Exact and approximate pattern matching in Burrows-Wheeler transformed (BWT) files: We proposed a method to extract the useful context information in linear time from the BWT transformed text. The auxiliary arrays obtained from BWT inverse transform brings logarithm search time. Meanwhile, approximate pattern matching can be performed based on the results of exact pattern matching to extract the possible candidate for the approximate pattern matching. Then fast verifying algorithm can be applied to those candidates which could be just small parts of the original text. We present algorithms for both k-mismatch and k-approximate pattern matching in BWT compressed text. A typical compression system based on BWT has Move-to-Front and Huffman coding stages after the transformation. We propose a novel approach to replace the Move-to-Front stage in order to extend compressed domain search capability all the way to the entropy coding stage. A modification to the Move-to-Front makes it possible to randomly access any part of the compressed text without referring to the part before the access point.

- Modified LZW algorithm that allows random access and partial decoding for the compressed text retrieval: Although many compression algorithms provide good compression ratio and/or time complexity, LZW is the first one studied for the compressed pattern matching because of its simplicity and efficiency. Modifications on LZW algorithm provide the extra advantage for fast random access and partial decoding ability that is especially useful for text retrieval systems. Based on this algorithm, we can provide a dynamic hierarchical semantic structure for the text, so that the text search can be performed on the expected level of granularity. For example, user can choose to retrieve a single line, a paragraph, or a file, etc. that contains the keywords. More importantly, we will show that parallel encoding and decoding algorithm is trivial with the modified LZW. Both encoding and decoding can be performed with multiple processors easily and encoding and decoding process are independent with respect to the number of processors.

To my parents and my wife with love

Acknowledgments

I am grateful to all the help given during my doctoral research. First and foremost, I thank Dr. Amar Mukherjee for his guidance and supervision as my academic advisor and committee Chair. His passion to the knowledge deeply impressed me and will sure motivate my pursuing in the rest of my life. I thank my committee member Dr. Mostafa Bassiouni, Dr. Sheau-Dong Lang, and Dr. Huaxin You for their insightful reviews, comments, and other contributions. I also thank for the financial supports provided by NSF fund IIS-9977336, IIS-0207819, and IIS-0312724.

The M5 group provides countless supplies to my research. It is a pleasure and encouraging to discuss with my fellow graduate students, Tao Tao, Ravi Vijaya Satya, Robert Franceschini, Fauzia Awan, Weifeng Sun, Raja Iqabal, and Nitin Motgi. My research and publications also reflect their selfless assistance. I am glad to have the communications with my friends Biao Chen, Yubin Huang, Zhiguang Xu, Rong Wang, Ning Jiang, Jun Li, Ji Liu, Yi Wang, Feng Lu, Guoqiang Wang, Yixiao Yang, Dahai Guo, Jiaying Ni, Chun Huang, Chengya Liang, Dr. Karl Chai, and many other warmhearted schoolmates. They made my life in Orlando a happy one.

Finally, I thank my parents Guoheng Zhang and Zhonghui Huang, my wife Liping Chen for their encouragement and love that never end.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

MOTIVATION AND INTRODUCTION

1.1 Motivation

People have long been exploring the ideal methodology of accurate representation, concise storage, and swift communication of the information in the history. The effort does not stop in the modern age and will continue in the foreseeable future. An earliest example might be transmitting the message of "the enemy comes" by signaling smoke on the top of mountain dating back to around two and half thousand years ago. The representation is simple, precise and easy to be transmitted by relaying the ignition. Language is in fact another example of accurate representation of our minds and is visualized/recorded using written symbols. Text was written on the papyrus, bamboo slices, papers; and now on magnetic tape and disk, compact disk and various media with the development of the modern technology. In the modern digital age, information is mostly processed by the machine automatically. Hence the need request of compact, precise, and efficient representation of the information are also applicable to the computers. With tremendous amount of information accumulated especially in the last few decades, data compression schemes are playing an increasingly significant role in developing compact representation of information. Moreover, finding the useful information from the mass storage emerged as another major problem today.

People are good at producing the data. In recent times, we have witnessed an unprecedented growth of textual information via the Internet, digital libraries and archival text data in many applications. The estimation of the growth rate is reflected by the Parkinson's Law on data that "data expands to fill the space available for storage". The TREC [TRE00] database holds around 800 million static pages having 6 trillion bytes of plain text equal to the size of a million books. The Google system routinely accumulates millions of pages of new text information every week. The Alexa.com is collecting over 1,000GB of information each day from the web and had collected over 35 billion web pages. There have been extensive needs to deal with the overwhelming data.

It is estimated that the memory usage of the computer systems tends to double roughly once every 18 months [LVS03]. Fortunately, the Moore's Law applies well on the computational and memory capacity. For example, the price curves of DRAM generations fits closest to the Moore's law. The capacity of other storage media keeps increasing with respect to its physical size. Two decades ago, a 10MB hard disk in a desktop PC was considered luxury; 40GB hard disk is almost the lowest configuration when you order a Dimension PC at Dell.com today. The 3.5 inch floppy drive for 1.44MB disk is an option and the 48xCDROM is a must for every PC. We may only see the 5 inch floppy drive in the museum or in the corner of our home garage. On the other hand, the price of the storage media has decreased drastically with regard to capacity and physical size. The 10MB hard disk in 1986 and a 160GB one in 2004 are sold with same price.

A recent study by the University of California at Berkeley [LVS03] puts the amount of new information generated in 2002 to be 5 exabytes bytes (an exabyte (EB) is bytes) which is approximately equal to all words spoken by human beings. While a good fraction of this information is of transient interest, useful information of archival value will continue to accumulate. Each year more and more information is created and available to the ordinary users at a dramatic rate through books, film, newspaper, television, radio, and various media. Ninety-two percent of the new information was stored on magnetic media, mostly in hard disks. Table 1.1 shows the comparisons of the units to measure the size of the digital data.

The report also estimates that new stored information grew about 30% a year between 1999 and 2002. For the storage distribution, the report points out that hard disks store most new information. Film represents 7% of the total, paper 0.01%, and optical media 0.002%. Furthermore, the information is not simply sleeping in the storage media. It flows through electronic channels – telephone, radio, TV, and the Internet – contained almost 18 exabytes of new information in 2002, three and a half times more than is recorded in storage media [LVS03].

People are not good at managing the data. We need ways to manage, organize and transport this data from one point to the other on data communications links with limited bandwidth. We must also have means to speedily find the information we need from this

Table 1.1: How Big is an Exabyte? *Source: The table is taken from the Berkeley report and many of these examples were taken from Roy Williams "Data Powers of Ten" web page at Caltech.*

Kilobyte (KB)	$1,000$ bytes OR 10^3 bytes 2 Kilobytes: A Typewritten page. 100 Kilobytes: A low-resolution photograph.
Megabyte (MB)	$1,000,000$ bytes OR 10^6 bytes 1 Megabyte: A small novel OR a 3.5 inch floppy disk. 2 Megabytes: A high-resolution photograph. 5 Megabytes: The complete works of Shakespeare. 10 Megabytes: A minute of high-fidelity sound. 100 Megabytes: 1 meter of shelved books. 500 Megabytes: A CD-ROM.
Gigabyte (GB)	1,000,000,000 bytes OR 10^9 bytes 1 Gigabyte: a pickup truck filled with books. 20 Gigabytes: A good collection of the works of Beethoven. 100 Gigabytes: A library floor of academic journals.
Terabyte (TB)	1,000,000,000,000 bytes OR $10^{1}2$ bytes 1 Terabyte: 50000 trees made into paper and printed. 2 Terabytes: An academic research library. 10 Terabytes: The print collections of the U.S. Library of Congress. 400 Terabytes: National Climactic Data Center (NOAA) database.
Petabyte (PB)	1,000,000,000,000,000 bytes OR $10^{1}5$ bytes 1 Petabyte: 3 years of EOS data (2001). 2 Petabytes: All U.S.academic research libraries. 20 Petabytes: Production of hard-disk drives in 1995. 200 Petabytes: All printed material.
Exabyte (EB)	1,000,000,000,000,000,000 bytes OR $10^{1}8$ bytes 2 Exabytes: Total volume of information generated in 1999. 5 Exabytes: All words ever spoken by human beings.

huge mass of data. Retrieval system, search engine, browsers, and other information management software are powerful tools for hunting relevant documents on the Internet. Although many well known search engines claim to be able to search multimedia information such as image and video using sample images or text description, text documents are still the most frequent targets. According to the Berkeley estimation [LVS03], 70% of the information on the internet is available as text, such as stock exchange data, library information, online books and tutorials, documents, and software, etc. Sometimes, a single site may also contain large collections of data such as a library database, thereby requiring an efficient search mechanism even to search within the local data. For example, Google now provides a new function of local search on your own PC and Microsoft will provide similar application soon. To facilitate the information retrieval, an emerging ad hoc standard for uncompressed text is XML which preprocesses the text by putting additional user defined metadata such as DTD or hyperlinks to enable searching with better efficiency and effectiveness. This increases the file size considerably, underscoring the importance of applying text compression. On account of efficiency (in terms of both space and time), there is a need to keep the data in compressed form for as much as possible.

Text compression provides a transformed representation of the text data that is understandable only by the computer (in this sense, it relates to cryptography to some extent.) The higher the compression ratio, the less disk space is needed to store the data. The advantage of the idea is two fold. First, we use less space to store the information. For example, English text can be compressed to about 30% of the original size, and images may be compressed by a factor of several hundreds times. Normally, lossless compression must be used for text because we expect the full text to be recovered from the compressed form, unlike audio/video and images which have a much higher degree of redundancy and can be compressed with lossy compression algorithms. Second, we require less bandwidth in the internet transmission compared with transmitting raw data. Obviously, it takes less time to download the text in its compressed form. Berkeley's report [LVS03] indicates that the World Wide Web contains about 170 terabytes of information on its surface. In volume this is seventeen times the size of the Library of Congress print collections. Instant messaging generates five billion messages a day (750GB), or 274 terabytes a year. It will be a considerable saving for the network traffic if the data are transmitted with a much smaller size.

Storage is not the purpose of keeping the data because we need to find useful information hidden in the data for different purposes. For example, data mining is a new area catering the need for exploring the knowledge from the sleeping data. The initial step of mining the knowledge is to retrieve the portion of the text by sending a query, typically using keywords. Then algorithms will be performed on the raw or preprocessed text. Pattern matching is the most popularly used method to search the text using keywords. Although there have been comprehensive studies on text information retrieval [BR99b, FB92b, WMB99], not much work has been done on searching directly on compressed text. The compact representation of text is unreadable for human beings. In order to read the data we need to reproduce the original text from the compressed text. Therefore, it is an extra overhead of decompression process rather than mining directly from the original form. Current research on compression shows little consideration for the relationship between the compression algorithm and searching algorithm. We will be focusing on minimizing the overhead by considering the optimal combination of compression and searching schemes.

Pattern matching is a typical starting point for knowledge discovering in large databases. There have been various exact and approximate pattern matching algorithms available in the literature. Boyer-Moore (BM) [BM77] and Knuth-Morris-Pratt (KMP) [KMP77] pattern matching algorithms are among the best of them. However, pattern matching on compressed text has not been thoroughly explored with the known compression methods. Efficient storage, transmission, searching, and mining the knowledge have become critical and difficult problems to deal with the tremendous data flow. In this dissertation, we will address the problems related to the lossless text compression, compressed pattern matching, and compressed text retrieval.

1.2 Some Background

1.2.1 Lossless Text Compression

The amount of digitized data available has heightened the challenge for ways to manage and organize the data (compression, storage, and transmission). Lossless text compression is an old but hard problem. Morse code for telegraphy and Braille code for blind are among the very early methods to build a new and compact representation of language, although compression is not their initial purpose. Modern data compression began in the late 1940s with the development of information theory. The general approach to text compression is to find a representation of the text requiring less number of binary digits. The standard ASCII code uses 8-bits to encode each character in the alphabet. Such a representation is not very efficient because it treats frequent and less frequent characters equally. If we encode frequent characters with a smaller (less than 8) number of bits and less frequent characters with larger number of bits (possibly more than 8 bits), it should reduce the *average* number of *bits per character* (BPC). This observation is the basis of the invention of the Morse code and the famous Huffman code developed in the early 50's. Huffman code typically reduces the size of the text file by about $50-60\%$ or provides compression rate of $4-5$ BPC [WMB99] based on statistics of frequency of characters. In the late 1940's, Claude E. Shannon laid down the foundation of the information theory and modeled the text as the output of a source that generates a sequence of symbols from a finite alphabet A according to certain probabilities. Such a process is known as a *stochastic process* and in the special case when the probability of occurrence of the next symbol in the text depends on the previous symbols or its context it is called a *Markov process*. Furthermore, if the probability distribution of a typical sample represents the distribution of the text it is called an *ergodic process* [Sha48, SW98]. The information content of the text source can then be quantified by the entity called *entropy* H given by

$$H = -\Sigma p_i \log p_i \tag{1.1}$$

where p_i denotes the probability of occurrence of the ith symbol in the text, sum of all symbol probabilities is unity and the logarithm is with respect base 2 and $-\log p_i$ is the amount of *information* in bits for the event (occurrence of the ith symbol). The expression of H is simply the sum of the number of bits required to represent the symbols multiplied by their respective probabilities. Thus the entropy H can be looked upon as defining the average number of BPC required to represent or encode the symbols of the alphabet. Depending on how the probabilities are computed or modeled, the value of entropy may vary. If the probability of a symbol is computed as the ratio of the number of times it appears in the text to the total number of symbols in the text, the so-called static probability, it is called an Order (0) model. Under this model, it is also possible to compute the dynamic probabilities which can be roughly described as follows. At the beginning when no text symbol has emerged out of the source, assume that every symbol is equiprobable [1]. As new symbols of the text emerge out of the source, revise the probability values according to the actual frequency distribution of symbols at that time. In general, an Order(k) model can be defined where the probabilities are computed based on the probability of distribution of the $(k+1)$-grams of symbols or equivalently, by taking into account the context of the preceding k symbols. A value of $k = -1$ is allowed and is reserved for the situation when all symbols are considered equiprobable, that is, $p_i = \frac{1}{|A|}$, where $|A|$ is the size of the alphabet A. When $k = 1$ the probabilities are based on bigram statistics or equivalently on the context of just

[1] This situation gives rise to what is called the zero-frequency problem. One cannot assume the probabilities to be zero because that will imply an infinite number of bits to encode the first few symbols since -log o is infinity. There are many different methods of handling this problem but the equiprobabilty assumption is a fair and practical one.

one preceding symbol and similarly for higher values of k. For each value of k, there are two possibilities, the static and dynamic model as explained above. For practical reasons, a static model is usually built by collecting statistics over a test *corpus*, which is a collection of text samples representing a particular domain of application (viz. English literature, physical sciences, life sciences, etc.). If one is interested in a more precise static model for a given text, a *semi-static* model is developed in a two-pass process; in the first pass the text is read to collect statistics to compute the model and in the second pass an encoding scheme is developed. Another variation of the model is to use a specific text to prime or seed the model at the beginning and then build the model on top of it as new text files come in.

Independent of the model, there is entropy associated with each file under that model. Shannon's fundamental noiseless source coding theorem says that entropy defines a lower limit of the average number of bits needed to encode the source symbols [SW98]. The "worst" model from information theoretic point of view is the order (-1) model, the equiprobable model, giving the maximum value Hm of the entropy. Thus, for the 8-bit ASCII code, the value of this entropy is 8 bits. The redundancy R is defined to be the difference [2] between the maximum entropy H_m and the actual entropy H. As we build better and better models by going to higher order k, lower will be the value of entropy yielding a higher value of redundancy. The crux of lossless compression research boils down to developing compression algorithms that can find an encoding of the source using a model with minimum possible

[2]Shannon's original definition is R/H_m which is the fraction of the structure of the text message determined by the inherent property of the language that governs the generation of specific sequence or words in the text [SW98].

entropy and exploiting maximum amount of redundancy. But incorporating a higher order model is computationally expensive and the designer must be aware of other performance metrics such as decoding or decompression complexity (the process of decoding is the reverse of the encoding process in which the redundancy is restored so that the text is again human readable), speed of execution of compression and decompression algorithms and use of additional memory.

Good compression means less storage space to store or archive the data, and it also means less bandwidth requirement to transmit data from source to destination. This is achieved with the use of a *channel* that may be a simple point-to-point connection or a complex entity like the Internet. For the purpose of discussion, assume that the channel is noiseless, that is, it does not introduce error during transmission and it has a *channel capacity* C that is the maximum number of bits that can be transmitted per second. Since entropy H denotes the average number of bits required to encode a symbol, C/H denotes the average number of symbols that can be transmitted over the channel per second [SW98]. A second fundamental theorem of Shannon says that however clever you may get developing a compression scheme, you will never be able to transmit on average more than C/H symbols per second [SW98]. In other words, to use the available bandwidth effectively, H should be as low as possible, which means employing a compression scheme that yields minimum BPC.

To study the various compression algorithms and schemes available, we can categorize the compression methods with different point of view. Here we briefly describe the compression algorithms according to the history and complexity of algorithm design. We will discuss in more detail in Chapter 2.The lossless text compression algorithm can be categorized into four classes:

- Basic techniques
- Statistical methods
- Dictionary methods
- Transform based methods

Basic techniques are pretty simple and intuitive. Although they are generally inefficient and cannot compete with the more complex methods, they are sometimes incorporated into complex compression schemes as the last stage of compression. Examples are Run Length Coding, Move-to-Front Coding, and Scalar Quantization. The statistical methods explore the probability distribution of the symbols and achieve better compression ratio. They use variable-sized codes based on the statistics of the symbols or group of symbols. The symbol or the group will be assigned shorter length of code if they appeared more often (with a higher frequency of occurrences). Designers aim to minimize the weighted average size to get the maximum compression, and the design of codes should avoid ambiguity so that the decoding process recognizes each code uniquely. The statistics can be obtained online or offline to estimate the source that generates the text. The online methods scan the text only once. The encoding for the current symbol is calculated by the prediction of the local probability based on the history statistics. Prediction is done by using the context (a certain amount of symbols appeared preceding the current symbol). The length of the context is called the order. The most famous online statistical method is PPM (Predict by Pattern Matching)[CW84] and its variances such as: PPMC, PPMD, PPMD+, PPM*, and PPMZ. The offline methods generally scan the whole text to obtain the statistics and design the code. Then the next step is to scan the text again and assign the code to the symbol or a group of symbols. Typical methods are Huffman coding [Huf52], Arithmetic coding [Ris79], etc.

Dictionary methods usually encode a sequence of symbols with another token. The dictionary stores the mapping of the series of symbols and the corresponding tokens, and it can be either static or dynamic. The dictionary can be implemented in an array or trie data structure for speeding up of the location of information. LZ (Ziv and Lempel) compression algorithm family is a typical set of dictionary based methods. Examples are LZ77 [ZL77], LZ78 [ZL78], etc. LZW is used as unix compress command. LZH is the improvement of LZSS and used as the compression tool gzip. Generally we can estimate the entropy of the text in low orders. For example, compute the first order entropy using Shannon's equation [Sha51]. The estimation of the second order entropy can be obtained by the statistics of the bigram. Huffman coding and Arithmetic coding are examples of using the first order entropy. The higher order entropy is difficult to compute due to the exponentially increased symbol combinations. For the more complicated algorithms, the actual order is sometimes implicit. For example, PPMC normally use the order up to five. The order in the LZ family is also a mixed one. So it is difficult to estimate the bound of a stationary ergodic source. Shannon made some experiments to estimate the entropy of English, but they do not prove it. It is hard to compute the entropy of the source up to infinite order. Due to the limitation of computer resources, we have to limit the level of the order when implementing the compression algorithm.

There are some techniques that do not directly compress the text but preprocess the text for the actual compression. One of the basic techniques, the Move-to-Front (MTF) method simply map each symbol into another, but sometimes the first order entropy of the transformed text has a lower entropy, hence it can be compressed better using Huffman or arithmetic coding. Another example is the Burrows-Wheeler block sorting algorithm[BW94a]. The block sorting does not compress the text at all, but actually adds an extra element, the index value. The next step, Move-to-Front, also does not compress anything. But the following Huffman or arithmetic coding can better compress the text due to the rearrangement of the text that somewhat converts the higher order sequences into the lower order ones. Therefore a good compression algorithm would make the overall performance to be optimal instead of just optimizing a certain step in the whole compression process.

1.2.2 Compressed Pattern Matching

The exact pattern matching problem is defined as follows: given a string P and a text T regarded as a longer string, find all the occurrences, if any, of pattern P in text T. The use of pattern matching is obvious for computers processing digitized information. Common applications of the exact pattern matching are in word processors, in text information retrieval, in internet search engines such as Google and Yahoo, in online electronic publications, in biology finding patterns of DNA and RNA sequences stored as strings, etc. Numerous other applications can be listed using pattern matching techniques. The exact pattern matching problem has been well solved for typical word processing applications. Typical exact pattern matching algorithms are: BM [BM77], KMP[KMP77], Aho-Corasick (AC) [AC75], Karp-Rabin (KR) [KR81], etc. They all have the linear bound of searching time and BM could even achieve sublinear running time. There are many string representation and processing problems related to the pattern matching such as suffix tree, suffix array, shortest common ancestor, longest common subsequence, etc. Solving those subproblems help us generate the idea of the kind of intermediate results that may expedite the search [Gus97]. When searching on the compressed text, we will try to build such data structures on the fly by (partially) decomposing the compressed text.

It is very common that user requires some highly similar results to the query words besides those that match exactly. There may be a typo, or actual words or sequences in the database may not be known precisely. Thus it is reasonable to ask for similar patterns in the database and make the decision based on the degree of similarity to the query. This is equivalent to allowing errors in the search results. This is called the approximate pattern matching problem. The approximate matching problem involves the definition of the distance/similarity between patterns and the degree of error allowed in the searching. Efficient techniques such as dynamic programming are available to solve the approximate pattern matching problem between two sequences. In certain applications, we also need to calculate the alignments along with the degree of the errors between the comparing patterns. Searching efficiently and effectively in the large database is still a challenging problem that might involve large amount of local alignment or comparison computation. It was reported that users often suffered from long delays from the on-line catalog of the University of California's library system. Using a local copy of Genbank (the major U.S. DNA database), it currently takes over four hours to find a string that was not there. The University of Central Florida system currently by default provide only totally exact matched result to the keywords that may lead to the miss of useful information to the user. When large databases are compressed, the problem may be even more significant, especially when allowing errors in the pattern. Text information is available in hundreds of millions of web sites on the internet nowadays. It is natural that more and more of the texts are stored in the compressed form to save the cost of local-disk storage and download time. For example, most academic publications are stored in compact form such as gzip and pdf. Current search engines do not have the ability to search the pattern in the compressed text. But there is demand for such tools from both technical and commercial domains. The intuitive method would be to decompress the whole text and search using the traditional pattern matching methods such as BM or KMP. However, it is not economical in both space and time cost. There are some exact pattern matching algorithms searching directly on the compressed text proposed to search the pattern on Run-length encoded text, LZ77, LZ78, LZW, Word Huffman, and Antidictionaries. Approximate pattern matching can be performed on Word Huffman, LZ78, LZW compressed files. BWT compression method has the advantage of having a compression ratio close to PPM and only slower than gzip. The block sorting algorithm makes it easy to get the needed suffixes from inverse BWT transform in linear time. Hence we can use fast algorithms such as binary search on partially decompressed files to locate all the occurrences of the patterns. It is also possible to perform the approximate pattern matching efficiently based on the exact matching on partially decompressed files.

1.2.3 Text Information Retrieval in Compressed Text

Storing and searching on the explosively increasing amount of data is one of the most important problems in the digital library age. A single site may also contain large collections of data such as library database. It has been pointed out that compression is a key for next-generation text retrieval systems [ZMN00]. A good compression method may facilitate efficient retrieval on compressed files. The amount of storage used and the efficiency of indexing and searching are major design considerations for an information retrieval system. The volume of data can be reduced by using compression techniques. However, search and retrieval become much more complicated. Usually, there is a tradeoff between the compression performance and the retrieval efficiency. Therefore, most of the existing search or retrieval schemes on compressed text use compression methods that have relatively lower compression ratio, but simpler indexing and searching schemes such as those using Run-length, Huffman, word based Huffman, or BWT [BW94a]. When the database is compressed, obviously, it is not efficient to decode the whole collection and locate the portions from the uncompressed text. Given a query using a keyword, there are two categories of methods to search a matched pattern in general. One is the compressed pattern matching that searches a pattern directly on the compressed file with or without preprocessing discussed in Chapter 5. Both exact and approximate matching can be performed. This method requires no or some offline preprocessing. The other method is the popular information retrieval approach that requires preprocessing by building index with the keyword and document frequency information []. The query is processed and the search is performed on the index files. Then the documents are ranked using some standard so that the precision and recall are optimal. Relevant feedback may also help to refine the query to have more accurate results. For large collections of text, it is difficult to access a piece of a compressed file for both compressed pattern matching method and popular text retrieval system with index files. One option is to break the whole collections into smaller documents [MSW93a]. However, the compression ratio will be poor for small files. The longer the sequence to be compressed, the better is the estimation of the source entropy. Furthermore, the request for retrieval may change for different purposes. For example, only a small portion of the collection that is relevant to the query is required to deliver to the user. A single record, or a paragraph in stead of a whole document might be enough. It is unnecessary to decompress the whole database and then locate the portion that is retrieved. Using a single level document partitioning system may not be the best answer. We propose to add tags into the document. Different tags indicate different granularity. Decoding will be performed within the bounds. The major concerns of compression method for the retrieval purpose, ranked roughly by their importance are: a) random access and fast (partial) decompression; b) fast and space-efficient indexing. c) good compression ratio. The compression time is not a major concern since the retrieval system usually performs off-line preprocessing to build the index files for the whole corpus. Besides the searching algorithm, random and fast access to the compressed data is critical to the response time to the user query in a text retrieval system. A typical text retrieval system is constructed as follows. First, the keywords are collected from the text database off-line and an inverted index file is built. Each entry points to all the documents that contain the keyword. A popular document ranking scheme is based on the keyword frequency tf and inverted document frequency idf. When a query is given, the search engine will match the words in the inverted index file with the query. Then the document ranking information is computed according to a certain logic and/or frequency rule to obtain the search results that point to the target documents. Finally, only the selected documents are displayed to the user. To find a good compression scheme that meets the given criteria, we first evaluate the performance of text compression algorithms currently in use. Besides the categorization given in Section 1.2.1, the compression schemes can also be categorized as entropy coders and model based coders. Huffman coding and Arithmetic coding are typical entropy coders. LZ family, including LZ77, LZ78, LZW, and their variants [ZL77, ZL78, Wel84] are the most popular compression algorithms because of their speed and good compression ratio. Canonical Huffman uses the language model in which English words are considered as symbols in the alphabet that contains all the words in the text. It has a sound compression performance, but it is language dependant. Dynamic Markov Coding (DMC) uses Markov model to predict the next bit using the history information. It has a good compression ratio. Prediction by Partial Matching (PPM) is currently the algorithm that achieves the best compression ratio. However it has a higher computational complexity. The Burrows-Wheeler Transform (BWT), or block-sorting algorithm has a compression ratio close to PPM and the speed is slightly slower than LZ algorithms. From the compressed searching point of view, there are algorithms that support direct searching and partial decoding. For example, in Huffman coding, given a query keyword, we can obtain the codes from the Huffman tree and then search the codes directly on the compressed file using pattern matching algorithms such as Boyer-Moore or Knuth-Morris-Pratt algorithms. It is possible that some further checking needs to be done. In Canonical Huffman model, a similar method can be used to search the word and decode partially [WMB99]. For the other compression algorithms we have to decode from the beginning of the compressed text and random access is difficult for them. In this thesis, we propose a modified LZW algorithm that supports random access and partial decoding. The original text retrieval system does not need to change on the query evaluation process. The data structure of the indexing file is still the same with the content to be changed into the index for the compressed file in place of the raw text file. A new tag system is incorporated with the indexing system to achieve the different levels of details for the text output. In our algorithm, we can decode any part of the text given the index of the dictionary entry and stop decoding until a certain tag is found or decode a given number of symbols.

1.3 Our Contribution

The problems that our research will focus on are:

- Exact and approximate pattern matching on compressed text used for information retrieval. Development of methods to speedily find the information we need from the huge mass of data. It is important to consider ways to keep the data in the compressed form for as much as possible, even when it is being searched. The algorithms are capable for both exact and inexact search.

- Search aware text compression: compressing text with an acceptable compression ratio while including the features that make the indexing and searching easier. We aim to develop compression schemes based on the identified class of algorithms that will support compressed domain search directly on the compressed data, with minimal or no decompression of the compressed database text. Some preprocessing may be used to help achieve the goal.

The thesis will make the following contributions on the data compression, compressed pattern matching and retrieval problems.

- Development of new lossless text Star family compression algorithms with better compression performance. We have proposed an approach to develop a reversible transformation that can be applied to a source text that improves ability to compress of existing algorithm . The basic idea is to encode every word in the input text file, which is also found in the English text dictionary that we are using, as a word in our transformed static dictionary. These transformed words create additional context information that is superior to the original text. Thus we achieve some compression at the preprocessing stage. We present a series of transforms which improve the comprehensive performance. We also propose the use of ternary tree data structure to improve the efficiency of encoding and decoding time.

- Exact and approximate pattern matching in Burrows-Wheeler transformed (BWT) files: We proposed a method to extract the useful context information in linear time from the BWT transformed text. The auxiliary arrays obtained from BWT inverse transform brings logarithm search time. Meanwhile, approximate pattern matching can be performed based on the results of exact pattern matching to extract the possible candidate for the approximate pattern matching. Then fast verifying algorithm can be applied to those candidates which could be just small parts of the original text. We present algorithms for both k-mismatch and k-approximate pattern matching in BWT compressed text. A typical compression system based on BWT has Move-to-Front and Huffman coding stages after the transformation. We propose a novel approach to replace the Move-to-Front stage in order to extend compressed domain search capability all the way to the entropy coding stage. A modification to the Move-to-Front makes it possible to randomly access any part of the compressed text without referring to the part before the access point.

- Modified LZW algorithm that allows random access and partial decoding for the compressed text retrieval: Although many compression algorithms provide good compression ratio and/or time complexity, LZW is the first one studied for the compressed pattern matching because of its simplicity and efficiency. Modifications on LZW algorithm provide the extra advantage for fast random access and partial decoding ability that is especially useful for the text retrieval system. Based on this algorithm, we can provide a dynamic hierarchical semantic structure for the text, so that the text search can be performed on the expected level of granularity. That is, search result will be provided at the wanted level of details. For example, user can choose to retrieval a single line, a paragraph, or a file, etc. that contains the keywords. More importantly, we will show that parallel encoding and decoding algorithm is trivial with the modified LZW. Both encoding and decoding can be performed with multiple processor easily and encoding and decoding process are independent with respect to the number of processors.

1.4 Contents of the Thesis

The rest of the thesis is organized as follows. Chapter 2 introduces the background and the review of the current schemes, algorithms, and/or systems for lossless text compression, (compressed) pattern matching, text retrieval in the literature. Traditional compression algorithms and their time/space performance will be discussed. String matching and useful data structures such as suffix tree will be presented. We will also review the text retrieval methods and typical systems on compressed text such as the one in Managing Gigabyte [WMB99].

By studying those algorithms, we present our new star transformation algorithms to make the text to be compressed at better compression ratio with almost all the existing models in Chapter 3. Method for efficient implementation is also presented. Chapter 4 discusses the pattern matching algorithm for compressed text and approximate pattern matching algorithms. We illustrate the new data structure from the current BWT compression scheme and propose our pattern matching algorithm in BWT transformed text. Chapter 5 describes the modified LZW algorithm that supports efficient text retrieval with the properties of random access and partial decoding. Chapter 6 concludes the works and discusses the possible future works related to this area.

CHAPTER 2

REVIEW OF RELATED WORKS

Data compression and searching have been embedded into numerous computer and internet applications including data storage and transmission. In this chapter, we will provide a more detail overview of text compression, compressed pattern matching and text retrieval on compressed file.

2.1 Classification of Lossless Compression Algorithms

No compression algorithm has yet been discovered that consistently attain the predictions of lower bound of data compression [Sha51] over wide classes of text files. Our goal in the lossless text compression area is to find better algorithms to explore the redundancy of the context and achieve a better compression ratio with a good time complexity. Besides the basic techniques such as Run Length Coding (RLC) and Move-to-Front (MTF), etc. the lossless algorithms can be classified into three broad categories: *statistical methods* , *dictionary methods* and *transform based methods*. We will give a review of these methods in this section.

2.1.1 Statistical Methods

The classic method of statistical coding is Huffman coding [Huf52]. It formalizes the intuitive notion of assigning shorter codes to more frequent symbols and longer codes to infrequent symbols. It is built bottom-up as a binary tree as follows: given the model or the probability distribution of the list of symbols, the probability values are sorted in ascending order. The symbols are then assigned to the leaf nodes of the tree. Two symbols having the two lowest probability values are then combined to form a parent node representing a composite symbol that replaces the two child symbols in the list and whose probability equals the sum of the probabilities of the child symbols. The parent node is then connected to the child nodes by two edges with labels '0' and '1'in any arbitrary order. The process is now repeated with the new list (in which the composite node has replaced the child nodes) until the composite node is the only node remaining in the list. This node is called the root of the tree. The unique sequence of 0's and 1's in the path from the root to the leaf node is the Huffman code for the symbol represented by the leaf node. At the decoding end the same binary tree has to be used to decode the symbols from the compressed code. In effect, the tree behaves like a dictionary that has to be transmitted once from the sender to receiver and this constitute an initial overhead of the algorithm. *This overhead is usually ignored in publishing the BPC* *results for Huffman code in literature.* The Huffman codes for all the symbols have what is called the *prefix property* which is that no code of a symbol is the prefix of the code for another symbol, which makes the code *uniquely decipherable (UD)*. This allows forming a code for a sequence of symbols by just concatenating the codes of the individual symbols and the decoding process can retrieve the original sequence of symbols without ambiguity. Note that a prefix code is not necessarily a Huffman code nor may obey the Morse's principle and a uniquely decipherable code does not have to be a prefix code, but the beauty of Huffman code is that it is UD, prefix and is also optimum within one bit of the entropy H. Huffman code is indeed optimum if the probabilities are $1/2^k$ where k is a positive integer. There are also Huffman codes called canonical Huffman codes which uses a look up table or dictionary rather than a binary tree for fast encoding and decoding [Sal00, WMB99].

Note in the construction of the Huffman code, we started with a model. Efficiency of the code will depend on how good this model is. If we use higher order models, the entropy will be smaller resulting in shorter average code length. As an example, a word-based Huffman code is constructed by collecting the statistics of words in the text and building a Huffman tree based on the distribution of probabilities of words rather than the letters of the alphabet. It gives very good results but the overhead to store and transmit the tree is considerable. Since the leaf nodes contain all the distinct words in the text, the storage overhead is equal to having an English words dictionary shared between the sender and the receiver. We will return to this point later when we discuss our transforms. Adaptive Huffman codes takes longer time for both encoding and decoding because the Huffman tree has to be modified at each step of the process. Finally, Huffman code is sometimes referred to as a variable length code (VLC) because a message of a fixed length may have variable length representations depending on what letters of the alphabet are in the message.

In contrast, the *arithmetic code* encodes a variable size message into a fixed length binary sequence [RL79]. Arithmetic code is inherently adaptive, does not use any lookup table or dictionary and in theory can be optimal for a machine with unlimited precision of arithmetic computation. The basic idea can be explained as follows: at the beginning the semi-closed interval $[0, 1)$ is partitioned into $|A|$ equal sized semi-closed intervals under the equiprobabilty assumption and each symbol is assigned one of these intervals. The first symbol, say a_1 of the message can be represented by a point in the real number interval assigned to it. To encode the next symbol a_2 in the message, the new probabilities of all symbols are calculated recognizing that the first symbol has occurred one extra time and then the interval assigned to a_1 is partitioned (as if it were the entire interval) into $|A|$ sub-intervals in accordance with the new probability distribution. The sequence a_1a_2 can now be represented without ambiguity by any real number in the new sub-interval for a_2. The process can be continued for succeeding symbols in the message as long as the intervals are within the specified arithmetic precision of the computer. The number generated at the final iteration is then a code for the message received so far. The machine returns to its initial state and the process is repeated for the next block of symbol. A simpler version of this algorithm could use the same static distribution of probability at each iteration avoiding re-computation of probabilities. The literature on arithmetic coding is vast and the reader is referred to the texts cited above [Sal00, Say00, WMB99] for further study.

The Huffman and arithmetic coders are sometimes referred to as the *entropy coders*. These methods normally use an order (0) model. If a good model with low entropy can be built external to the algorithms, these algorithms can generate the binary codes very efficiently. One of the most well known modeler is "*prediction by partial match*" (PPM) [CW84, Mof90b]. PPM uses a finite context Order (k) model where k is the maximum context that is specified ahead of execution of the algorithm. The program maintains all the previous occurrences of context at each level of k in a trie-like data structure with associated probability values for each context. If a context at a lower level is a suffix of a context at a higher level, this context is excluded at the lower level. At each level (except the level with $k = -1$), an *escape character* is defined whose frequency of occurrence is assumed to be equal to the number of distinct context encountered at that context level for the purpose of calculating its probability. During the encoding process, the algorithm estimates the probability of the occurrence of the *next character* in the text stream as follows: the algorithm tries to find the current context of maximum length k in the context table or trie. If the context is not found, it passes the probability of the escape character at this level and goes down one level to $k-1$ context table to find the current context of length $k-1$. If it continues to fail to find the context, it may go down ultimately to $k = -1$ level corresponding to equiprobable level for which the probability of any next character is $1/|A|$. If a context of length q, $0 \leq q \leq k$, is found, then the probability of the next character is estimated to be the product of probabilities of escape characters at levels k, $k-1$, ..., $q+1$ multiplied by the probability of the context found at the qth level. This probability value is then passed to the backend entropy coder (arithmetic coder) to obtain the encoding. Note, at the beginning there is no context available so the algorithm assumes a model with $k = -1$. The context lengths are shorter at the early stage of the encoding when only a few contexts have been seen. As the encoding proceeds, longer and longer contexts become available. In one version of PPM, called PPM*, an arbitrary length context is allowed which should give the optimal minimum entropy. In practice a model with $k = 5$ behaves as good as PPM* [CT97]. Although the algorithm performs very well in terms of high compression ratio or low BPC, it is very computation intensive and slow due to the enormous amount of computation that is needed as each character is processed for maintaining the context information and updating their probabilities.

Dynamic Markov Compression (DMC) [CH93] is another modeling scheme that is equivalent to finite context model but uses finite state machine to estimate the probabilities of the input symbols which are bits rather than bytes as in PPM . The model starts with a single state machine with only one count of '0' and '1' transitions into itself (the zero frequency state) and then the machine adopts to future inputs by accumulating the transitions with 0's and 1's with revised estimates of probabilities. If a state is used heavily for input transitions (caused either by 1 or 0 input), it is cloned into two states by introducing a new state in which some of the transitions are directed and duplicating the output transitions from the original states for the cloned state in the same ratio of 0 and 1 transitions as the original state. The bit-wise encoding takes longer time and therefore DMC is very slow but the implementation is much simpler than PPM and it has been shown that the PPM and DMC models are equivalent [BM89].

2.1.2 Dictionary Methods

The dictionary methods, as the name implies, maintain a *dictionary or codebook* of words or text strings previously encountered in the text input and data compression is achieved by replacing strings in the text by a reference to the string in the dictionary. The dictionary is *dynamic or adaptive* in the sense that it is constructed by adding new strings being read and it allows deletion of less frequently used strings if the size of the dictionary exceeds some limit. It is also possible to use a *static* dictionary like the word dictionary to compress the text. The most widely used compression algorithms (Gzip and Gif) are based on Ziv-Lempel or LZ77 coding [ZL77] in which the text prior to the current symbol constitute the dictionary and a greedy search is initiated to determine whether the characters following the current character have already been encountered in the text before, and if yes, they are replaced by a reference giving its relative starting position in the text. Because of the pattern matching operation the encoding takes longer time but the process has been fine tuned with the use of hashing techniques and special data structures. The decoding process is straightforward and fast because it involves a random access of an array to retrieve the character string. A variation of the LZ77 theme, called the LZ78 coding, includes one extra character to a previously coded string in the encoding scheme. A more popular variant of LZ78 family is the so-called LZW algorithm which lead to widely used *Compress* utility. This method uses a suffix tree to store the strings previously encountered and the text is encoded as a sequence of node numbers in this tree. To encode a string the algorithm will traverse the existing tree as far as possible and a new node is created when the last character in the string fails to traverse a path any more. At this point the last encountered node number is used to compress the string up to that node and a new node is created appending the character that did not lead to a valid path to traverse. In other words, at every step of the process the length of the recognizable strings in the dictionary gets incrementally stretched and is made available to future steps. Many other variants of LZ77 and LZ78 compression family have been reported in the literature (See [Sal00] and [Say00] for further references).

2.1.3 Transform Based Methods: The Burrows-Wheeler Transform (BWT)

The word 'transform' has been used to describe this method because the text undergoes a transformation, which performs a permutation of the characters in the text so that characters having similar lexical context will cluster together in the output. Given the text input, the forward Burrows-Wheeler transform [BW94a] forms all cyclic rotations of the characters in the text in the form of a matrix M whose rows are lexicographically sorted (with a specified ordering of the symbols in the alphabet). The last column L of this sorted matrix and an index r of the row where the original text appears in this matrix is the output of the transform. The text could be divided into blocks or the entire text could be considered as one block. The transformation is applied to individual blocks separately, and for this reason the method is referred to as *block sorting* transform [Fen96b]. The repetition of the same character in the block might slow down the sorting process; to avoid this, a run-length encoding (RLE) step could be preceded before the transform step. The Bzip2 compression algorithm based on BWT transform uses this step and other steps as follows: the output of the BWT transform stage then undergoes a final transformation using either move-to-front (MTF) [BST86a] encoding or distance coding (DC) [Arn00] which exploits the clustering of characters in the BWT output to generate a sequence of numbers dominated by small values (viz. $0, 1$ or 2) out of possible maximum value of $|A|$. This sequence of numbers is then sent to an entropy coder (Huffman or Arithmetic) to obtain the final compressed form. The inverse operation of recovering the original text from the compressed output proceeds by decoding the inverse of the entropy decoder, then inverse of MTF or DC and then an inverse of BWT. The inverse of BWT obtains the original text given (L, r). This is done easily by noting that the first column of M, denoted as F, is simply a sorted version of L. Define an index vector Tr of size $|L|$ such that $Tr[j] = i$ if and only if both $L[j]$ and $F[i]$ denote the kth occurrence of a symbol from A. Since the rows of M are cyclic rotations of the text, the elements of L precede the respective elements of F in the text. Thus $F[Tr[j]]$ cyclically precedes $L[j]$ in the text which leads to a simple algorithm to reconstruct the original text. More details will be discussed in chapter 4

2.1.4 Comparison of Performance of Compression Algorithms

An excellent discussion of performance comparison of the important compression algorithms can be found in [WMB99]. In general, the performance of compression methods depends on the type of data being compressed and there is a tradeoff between compression performance, speed and the use of additional memory resources. The authors report the following results with respect to the Canterbury corpus: In order of increasing compression performance (decreasing BPC), the algorithms can be listed as order zero arithmetic, order zero Huffman giving over 4 BPC; the LZ family of algorithms come next whose performance range from 4 BPC to around 2.5 BPC (gzip) depending on whether the algorithm is tuned for compression or speed. Order zero word based Huffman (2.95 BPC) is a good contender for this group in terms of compression performance but it is two to three times slower in speed and needs a word dictionary to be shared between the compressor and decompressor. The best performing compression algorithms are: Bzip2 (based on BWT), DMC, and PPM all giving BPC ranging from 2.1 to 2.4. PPM is theoretically the best but is extremely slow as is DMC, bzip2 strikes a middle ground, it gives better compression than Gzip but is not an on-line algorithm because it needs the entire text or blocks of text in memory to perform the BWT transform. LZ77 methods (Gzip) are fastest for decompression, then LZ78 technique, then Huffman coders, and the methods using arithmetic coding are the slowest. Huffman coding is better for static applications whereas arithmetic coding is preferable in adaptive and online coding. Bzip2 decodes faster than most of other methods and it achieves good compression as well. A lot of new research on Bzip2 (see Chapter 4) has been carried on recently to push the performance envelope of Bzip2 both in terms of compression ratio and and speed and as a result Bzip2 has become a strong contender to replace the popular Gzip and Compress.

New research is going on to improve the compression performance of many of the algorithms. However, these efforts seem to have come to a point of saturation with regard to lowering the compression ratio. To get a significant further improvement in compression, other means like transforming the text before actual compression and use of grammatical and semantic information to improve prediction models should be looked into. Shannon made some experiments with native speakers of English language and estimated that the English language has entropy of around 1.3BPC [Sha51]. Thus, it seems that lossless text compression research is now confronted with the challenge of bridging a gap of about 0.8 BPC in terms of compression ratio. Of course, combining compression performance with other performance metric like speed, memory overhead and on-line capabilities seem to pose even a bigger challenge.

2.1.5 Transform Based Methods: Star (*) transforms

We present our research on new transformation techniques that can be used as preprocessing steps for the compression algorithms described in the previous section. The basic idea is to transform the text into some intermediate form, which can be compressed with better efficiency. The transformation is designed to exploit the natural redundancy of the language with a special static dictionary. We have developed a class of such transformations, each giving better compression performance over the previous ones and most of them giving better compression over current and classical compression algorithms discussed in the previous section. We will present a transform called Star Transform (also denoted by *-encoding) to preprocess the original text. We then present four new transforms called LPT, SCLPT, RLPT and LIPT that improves the compression. The algorithms use a fixed amount of storage overhead in the form of a word dictionary for the particular corpus of interest and must be shared by the encoder/sender and decoder/receiver of the compressed files. Word based Huffman method also make use of a static word dictionary but there are important differences as we will explain in chapter 3. Because of this similarity, we specifically compare the performance of our preprocessing techniques with that of the word-based Huffman. Typical size of dictionary for the English language is about 0.5 MB and can be downloaded along with application programs. If the compression algorithms are going to be used over and over again, which is true in all practical applications, the amortized storage overhead for the dictionary is negligibly small. We will present experimental results measuring the performance (compression ratio, compression times, and decompression times) of our proposed preprocessing techniques using three corpuses: Calgary [Cor00a], Canterbury [Cor00b] and Gutenberg corpus [Cor].

2.2 Compressed Pattern Matching

Since the digitized data is usually stored using some compression technique, and because of the problem of efficiency (in terms of both storage space and computational time), the trend now is to keep the compressed data in its compressed form for as much time as possible. That is, operations such as search and analysis on the data (text or images) is performed directly on the compressed representation, without decompression, or at least, with minimal decompression. Intuitively, compared to working on the original uncompressed data, operating directly on the compressed data will require the manipulation of a less amount of data as, and hence should be more efficient. This also avoids the often time consuming process of decompression, and the problem of storage space as may be required to keep the decompressed data. The need to search data directly in its compressed form is even being recognized by new international compression standards such as MPEG-7 [Sik97] where part of the requirements is the ability to search for objects directly in the compressed video.

Searching for patterns is an important function in many applications, for both humans and machine. The *pattern searching problem* can be stated as follows: given a query string (the *pattern*), and a database string (the *text*), find one or all of the occurrences of the query in the database. The problem then is to search the entire text for the requested pattern, producing a list of the positions in the text where a match starts (or ends). In this section, we describe the pattern matching problem, and the general methods that the have been used to reduce the time required.

2.2.1 The pattern matching problem and its variants

Solution to the pattern searching problem depends on a variant of the problem - the *string pattern matching problem*: given two strings, determine whether they are matches or not. Matches between strings are determined based on the distance between them.

The distance is traditionally calculated using the *string edit distance* (also called the Leveistien distance). Given two strings $A : a_1 ... a_n$, and $B : b_1 ... b_m$, over an alphabet Σ, and a set of allowed edit operations, the edit distance indicates the minimum number of edit operations required to transform one string into the other. Three basic types of edit operations are used - insertion of a symbol, $(\epsilon \rightarrow a)$; deletion of a symbol, $(a \rightarrow \epsilon)$; and substitution of one symbol with another $(a \rightarrow b)$; (ϵ represents the zero-length empty symbol, and $x \rightarrow y$ indicates that x is transformed into y). The edit operations could be assigned different costs, using suitable weighting functions. The edit distance is a generalization of the Hamming distance, which considers only strings of the same length, and allows only substitution operations. Computing the edit distance usually involves dynamic programming, and requires an $O(mn)$ computational time.

Given a text string A, and a pattern string B, the *exact string matching problem* is to check for the existence of a substring of the text that is an exact replica of the pattern string. That is, the edit distance between the substring of A and the pattern should be zero. Exact pattern matching is an old problem, and various algorithms have been proposed [Sel80, WF74, BM77, KMP77].

A variant of the pattern matching problem is the *k-difference problem* also called *approximate string matching*. The problem is to check if there exists a substring A_s of a string A, such that the edit distance between A_s and a second string B is less than k. Another form of approximate matching, the *k-mismatch problem*, checks for a substring of A having only a maximum of k mismatches with B. That is, only the substitution operation is allowed. The parameter k thus acts as a form of threshold to determine the correctness of a match. As with the exact matching problem, different algorithms have also been proposed for the case of approximate matching [Ukk85, GP90, CL92, Mye94].

Other variants of the pattern matching problem have also been identified, usually for specific applications. Examples include

- pattern matching with swaps [LKP97]: a transposition of two symbols (or symbol blocks) in one of the strings is treated specially by using a different weights;

- pattern matching with fusion [TY85, ALK99]: consecutive symbols the same character can be merged into one symbol, and one symbol can be split into different symbols of the same character;

- pattern matching with don't cares [Aku94] - a more general form of approximate pattern matching in which wild characters can be allowed in both the text and the pattern;

- multiple pattern matching (a generalization of the pattern matching problem, in which various patterns can be searched for in parallel.

- super-pattern matching: finding a pattern of patterns [KM99].

- multidimensional pattern matching [GG97, LV94]: matching when the text and pattern are multidimensional - typically used for images (2D pattern matching) or video (2D/3D pattern matching).

2.2.1.1 Compressed pattern matching

In general, compressed pattern matching involves one or more of the above variants, with the constraint that, either the text, the pattern, or both are in compressed form [AB92, KPR95]. The *fully compressed pattern matching* is when both the text and the pattern are both compressed, and matching involves no form of decompression.

The general problems are that the format used to represent the compressed data is usually different from that of uncompressed data. More seriously, applying the same compression algorithm on two identical patterns that have different contexts could lead to completely different representations. That is, the same pattern located in two different text regions could result in different representations. Matching in such an environment will then have to consider the specific compression scheme used, and how the context could affect the compression. For lossy compression, the effect of the introduced error would also have to be considered, and once again, the error introduced could depend on the context.

2.2.1.2 Applications of pattern matching

Although pattern matching is sometimes pursued for its algorithmic significance, it also has applications in various real life problems. Traditional areas where pattern matching has been used include simple spell checkers, comparing files and text segments, protein and DNA sequence alignments [Wat89, RC94, CL94], automatic speech recognition [SC78, NO00], character recognition [BS90], shape analysis [TY85], and general computer vision [TY85, BS90].

Recently, new applications of string pattern matching have been reported. Examples are in image and video compression [ASG00], audio compression [ASG00], video sequence analysis [ALK99] music sequence comparison [MS90], and music retrieval .

2.2.2 Search strategies for text

The naive pattern-matching algorithm runs in $O(nm)$ time. It generally ignores context information that could be obtained from the pattern, or from the text segment already matched. Most algorithms that provide significant improvement in the matching make use of such information, by finding some relationship between the symbols in the pattern and/or text.

Fast methods for string pattern matching is an area that has long been investigated, especially for exact pattern matching [BM77, Ca94, KMP77]. The methods can be broadly grouped as either *pre-indexing*, *pre-filtering*, or their combination. Pre-indexing (or preprocessing) usually involves the description of the database strings using a pre-defined index. The indices are typically generated by use of some hashing function or a scoring scheme [Mye94, RC94, CL94]. Pre-filtering methods generally divide the matching problem into two stages: the filtering stage and the verification stage [CL94, OM88, WM92b, PW93, ST96]. In the first stage, an initial filtering is performed to select candidate regions of the database sequence that are likely to be matches to the query sequence. In the second stage, a detailed analysis is made on only the selected regions to verify if they are actually matches.

The performance (in both efficiency and reliability of results) depends critically on the pre-filtering stage: if the filter is not effective in selecting only the text regions that are potentially similar to the pattern, the verification stage will end up comparing all parts of the text. Conversely, any region missed during the filtering stage can not be considered in the verification, and hence any false misses incurred at the first stage will be carried over to the final results. Pattern matching algorithms with sub-linear complexity have recently been reported [Mye94, CL94]. They generally combine both pre-indexing and pre-filtering methods. For [CL94], sub-linearity was defined in the sense of Boyer-Moore [BM77]: on average, less than n symbols are compared for a text of length n. That is, matching time is in $O(n^p)$ for some $0 < p < 1$

Fast algorithms have also been proposed for approximate string matching. Ukkonen [Ukk85] suggested the use of a cut off, which avoids calculating portions of a column if the entries can be inferred to be more than the required k-distance. Galil and Park [GP90] proposed some methods based on the observation that the diagonal of the edit distance matrix is non decreasing, and that adjacent entries along the rows or columns differ by at most one (when equal weights of unity are used for each edit operation). Chang and Lawler [CL94] proposed the column partitioning of the matrix based on the matching statistics - i.e. the longest local exact match. A general comparison of approximate pattern matching algorithms is presented in [CL92].

In this section we discuss methods for pattern matching in uncompressed text. The methods used for approximate pattern matching generally make use of techniques for exact pattern matching. Further, the various proposed fast algorithms for exact pattern matching can be traced to one or more of the three basic fast algorithms - KR, KMP, and BM algorithms. All the three algorithms used some form of pre- processing. BM and KR also used pre-indexing and verification.

2.2.2.1 Linear search

Although a simple linear search is often regarded as the least efficient method for searching, it has some interesting variants that can perform surprisingly well. In particular, if the access pattern to the text is known then more frequently accessed records can be placed nearer the front, and if the probability distribution of access is skewed this can result in very efficient searching. "Self-adjusting" lists [ST85] exploit this by using various heuristics to move items towards the front when they are used.

This idea can be extended to compressed-domain searching by observing that the order of the data in the compressed file might be permuted to put frequently accessed items towards the front. For example, in an image, areas that have a lot of detail might be more likely to be chosen. Savings can also be made by putting smaller items towards the front, if they are likely to cost less to make a comparison.

2.2.2.2 The Karp-Rabin Algorithm

The Karp-Rabin (KR) algorithm [KR81] is based on the concept of hashing, by considering the equivalence of two numbers modulo another number. Given a pattern P, the m consecutive symbols of P are viewed as a length-m d-ary number, say P_d. Typically, d is the size of the alphabet, $d = |\Sigma|$. Similarly, m-length segments of the text T are also converted into the same d-ary number representation. Suppose the numeric representation of the $i-$th such segment is $T_d(i)$. Then we can conclude that the pattern occurs in the text if $P_d = T_d(i)$, for some i - i.e. if the numeric representation for the pattern is the same as that of some segment of the text.

The KR algorithm provides fast matching by pre-computing the representations for the pattern and the text segments. For the m-length pattern, this is done in $O(m)$ time. Interestingly, the representation for each of the $(n - m)$ possible m-length segments of the n- length text can also be computed in $O(n)$ total time, by using a recursive relationship between the representations for consecutive segments of the text. Hence, the algorithm takes $O(n+m)$ time to compute the representations, and another $O(n)$ time to find all occurrences of the pattern in the text.

A problem arises when the pattern is very long, whereby the corresponding representations could be very large numbers. The solution is to represent the numbers to a suitable modulus, usually chosen as a prime number. This may however lead to the possibility of two different numbers producing the same representation, leading to spurious matches. Hence, a verification stage is usually required for the KR algorithm. The chance of a spurious match can be made arbitrary small by choosing large values for the modulus. The time required for verification will usually be very small when compared to that of matching, and hence can be ignored. On average, the running time is $O(n+m)$, while the worst case is $O((n-m+1)m)$.

2.2.2.3 The Knuth-Morris-Pratt Algorithm

The KMP algorithm [KMP77] simulates a pattern-matching automaton. It uses certain information gained by considering how the pattern matches against shifts of itself to determine which subsequent positions in the text can be skipped without missing out possible matches.

The information is pre-computed by use of a prefix function. In general, when the pattern is matched against a text segment, it is possible that a prefix of the pattern will match a corresponding suffix of the text. Suppose we denote such prefix of the pattern as P_p. The prefix function determines which prefix of the pattern P is a suffix of the matching prefix P_p. The prefix function is pre-computed from the m-length pattern in $O(m)$ time using an iterative enumeration of all the prefixes of $p_1p_2\ldots p_m$ that are also suffixes of $p_1p_2\ldots p_q$, for any $q, q = 1, 2, \ldots, m$.

By observing that a certain prefix of the pattern has already matched a segment of the text, the algorithm uses the prefix function to determine which further symbol comparisons will not result in a potential exact match for the pattern, and hence skips them. The overall matching time is bounded by $O(n + m)$.

The KMP algorithm is one of the more frequently cited pattern- matching algorithms. It has also been used for multidimensional pattern match [Bak78] and for compressed domain matching. See Table 2.1.

2.2.2.4 The Boyer-Moore Algorithm

Like the KMP, the BM algorithm matches the pattern and the text by skipping characters that are not likely to result in exact matching with the pattern. Like the KR algorithm, it also performs a pre-filtering of the text, and thus requires an $O(m)$ verification stage. Unlike the other methods, it compares the strings from right to left of the pattern.

At the heart of the algorithm are two matching heuristics - the *good- suffix heuristic* and the *bad-character heuristic*, based on which it can skip a large portion of the text. When a mismatch occurs, each heuristic proposes a number of characters that should be skipped at the next matching step, such that a possibly matching segment of the text will not be missed.

The match is performed by sliding the pattern over the text, and by comparing the characters right to left, starting with the last character in the pattern. When a mismatch is found, the mismatching character in the text is called the "bad character". The part of the text that has so far matched some suffix of the pattern is called the "good suffix".

The bad-character heuristic proposes to move the pattern to the right, by the amount that guarantees that the bad character in the text will match the rightmost occurrence of the bad character in the pattern. Therefore, if the bad character does not occur in the pattern, the pattern may be moved completely past the bad character in the text. The good-suffix heuristic proposes to move the pattern to the right, by the minimum amount that guarantees that some pattern characters will match the good suffix characters previously found in the text. The BM algorithm then takes the larger of the two proposals.

It is possible that the bad-character heuristic might propose a negative shift (i.e. moving back to the already matched text area). However, the good-suffix heuristic always proposes a positive number, and hence guaranteeing progress in the matching.

The bad-character heuristic requires $O(m + |\Sigma|)$ while the good- suffix heuristic requires $O(m)$ time units. The BM algorithm has a worst case running time of $O((n-m+1)m+|\Sigma|)$. The average running time is typically $\leq O(n + m)$. Overall, the BM algorithm generally produces better performance than the KMP and the KR algorithms for long patterns (large m), and relatively large alphabet sizes. See [Ca94, HS91] for new improvements on the BM algorithm.

2.2.2.5 Bit-parallel Algorithms

The Shift-Or [BG92] and Shift-And [KTS99] algorithms are another family of algorithms that have been proposed to improve the efficiency of string pattern matching. These produce speed-ups by exploiting the parallelism in the bit level representation of the characters in the symbol alphabet. The bit-parallel algorithms have also been used in compressed pattern matching [MNB00, NR99a, KTS99].

Various other algorithms have also been proposed, most of them being some modification or combination of the above methods. A recent survey by Hume and Sunday [HS91] describes a more efficient variant of the Boyer-Moore method. [CL96] gives a brief overview of pattern-matching methods, including the BM and KMP algorithms. The paper also discussed text compression, but the relationship between the compression and pattern matching was not discussed. The basic pattern matching algorithms have been extended to two dimensional pattern matching [Bir77], which was improved by [ZT89]. Baker[Bak78] applies string matching algorithms to character arrays. The algorithms also represent the primary building blocks for compressed domain pattern matching, Table 2.1.

There are also methods based on automata theory [CLR90]. Navarro and Raffinot [NR00] proposed methods that combine surfix automata and bit-parallel algorithms. Various other methods for approximate pattern matching have also been proposed in the literature [LV88, WM92b, MW92, WM92a, MW94, Man97, Tak94, Tak96, ALV92].

2.2.3 Relationship between searching and compression

It might seem that compression and searching work against each other, since a simple system would have to decompress a file before searching it, thus slowing down the pattern matching process. However, there is a strong relationship between compression and pattern matching, and this can be exploited to enable both tasks to be done efficiently at the same time.

In fact, pattern matching can be regarded as the basis of compression. For example, a *dictionary* compression system might identify English words in a text, and replace these with a reference to the word in a lexicon. The main task of the compression system is to identify patterns (in this example, words), which are then represented using some sort of compact code. If the type of pattern used for compression is the same as the type being used during a later search of the text, then the compression system can be exploited to perform a fast search. In the example of the dictionary system, if a user wishes to search the compressed text for words, then they could look up the word in the lexicon, which would immediately establish whether a search will be successful. If the word is found, then its code could be determined, and the compressed text searched for the code. This will considerably reduce the amount of data to be searched, and the search will be matching whole words rather than a character at a time. In one sense, much of the searching has already been performed off-line at the time of compression.

The potential savings are large. Text can be compressed to less than a half of its original size, and images are routinely compressed to a tenth or even a hundredth of the size of the raw data. These factors indicate that there is considerable potential to speed up searching, and indeed, systems exist that are able to achieve much of this potential saving. For instance, compressed domain indexing and retrieval is the preferred approach to multimedia information management [AL96, MIP99], where orders of magnitude speedup has been recorded over operations on uncompressed data [AL97, YL96].

Some authors have considered compression as basically a pattern matching problem [ASG00, AGS99, LS97]. More generally, most compression methods require some sort of searching:

- The Ziv-Lempel methods search the previously coded text for matches;
- PPM methods search for previous occurrences of a context using a trie data structure to predict what will happen in the current one;
- DMC uses a finite state machine to establish a context that turns out to have a similar meaning to the PPM context [BM89]. This is akin to algorithms such as Boyer-Moore constructing a machine to accelerate a search; and
- Vector Quantization (VQ) must search the codebook for the nearest match to the pattern being coded.
- MPEG requires searching as part of its motion estimation and motion compensation - the key aspects of the MPEG standard, as they affect both the compression ratio and compression time. Motion estimation requires a fast method to determine the motion vectors, and always involves searching for the matching blocks withing a spatio-temporal neighbourhood. While the quality of the compression improves with more search area, the compression time increases.

In [KF93], data compression was viewed as a pattern recognition problem. Explicit considerations on the data structures used in searching as a way of improving the compression performance have been considered in [BK93, Szp93, CS94]

The relationship between pattern matching and compression for images have been studied in [AGS99, ASG00]. More theoretical studies on optimal and suboptimal data compression with respect to pattern matching can be found in [LS97, YK96, SG93, Szp93]. A comparative study of pattern-matching image compression algorithms is presented in [YK95]

In general, for both lossy and lossless compression, more extensive searching often results in more compression, but with a correspondingly more compression time. For lossy compression, more search usually leads to less error in the compression (i.e. better quality in the reconstructed image). There is thus a trade-off between the extent of the search and the compression time.

More importantly, the different searching activities may be exploited later for compressed-domain pattern matching. In principle we need only code the pattern to be located, and then search for the compressed pattern in the compressed data. However, because coding can depend on the context of the item being coded, this näive approach will not work. Furthermore, we may be looking for an approximate match, and two patterns that are similar may not appear to be similar in the compressed domain.

A solution could be to constrain the compression, such that overlapping between contexts is suitable for matching. An example here is the tagged Huffman coding used in [MNB00].

In general, for a compression scheme to be suitable for compressed pattern matching, the scheme may need to provide random access to different points in the compressed data (this may require splitting the data into blocks and coding blocks of data at a time), a dictionary or vacubulary of the codewords, and a fixed code assignment for the encoded data stream.

2.2.4 Searching compressed data: lossless compression

Manber has described a compression system that allows for search [Man97]. This system is tolerant of errors, that is, it allows for approximate pattern matching, although the compression is lossless. [ABF96a] describe a method for searching LZW coded files. [BCA98] extend the work of [ABF96a] to an LZ compression method that uses the so-called "identity" or ID heuristic [MW85]. The ID heuristic is also known as LZMW. This heuristic grows the phrases in the dictionary by concatenating pairs of adjacent parsed phrases, rather than just adding one character to an existing phrase. The search method is able to exploit these large components to keep track of whether or not they contain the target pattern. Their algorithm requires $O(m + t)$ space, where m is the pattern size and t is the maximum target length. This is essentially optimal. However, the search time is $O(n(m + t))$, where n is the size of the compressed file. This is not as good as the "optimal" time established by Amir and Benson [AB92], which is $O(n + m)$. [KPR95] consider *fully compressed pattern matching* for LZ coded data, where the search pattern is also compressed, and neither are decompressed during searching.

Mukherjee et. al. [MA94] describe techniques for searching Huffman compressed files. Related VLSI algorithms have also been published [MA95]. Their method raises the possibility of searching for part of a variable-length compressed string, even if the compressed file is only searched on byte boundaries. This is achieved by searching for all variations of the search string generated by starting at different points in the string. Only eight starting points need be considered to cover every possible way the coded string could cross a byte boundary. This idea almost sounds like what could be achieved with the Burrows-Wheeler transform [BW94a] (BWT), as part of the BWT involves sorting the data according to context. [MA94] also claim that their method can search data that has been compressed with an *adaptive* Huffman code, and they claim to have proved that it is not possible to search data that has been compressed with arithmetic coding.

Amir et. al. [AB92] describe a method for searching two-dimensional data that has been compressed by run-length coding. An "optimal" version is described in [ABF97]. The general case of pattern matching for a class of "highly compressed" two-dimensional texts is explored by [BKL97, BKL96a, BKL96b]. They distinguish between compressed pattern matching, where the text is compressed, and full compressed pattern matching, where both the search pattern and the text are compressed.

Maa [Maa93] considers a special case where the pattern to be located is a bar-code. Maa observes that for the CCITT fax standard, which uses both vertical and horizontal run-length coding, bar codes create distinctive coding patterns, and can be detected reliably. It may be possible to extend this idea to other types of images; for example, half tone images will compress very poorly using run-length coding; text will have many short runs; and line drawings will have many long runs of white.

Table 2.1 shows the theoretical performance for various proposed algorithms for compressed pattern matching, using lossless compression schemes. Table 2.2 gives the reference of the algorithms given in table 2.1.

Table 2.1: Methods for compressed pattern matching for text. See Table 2.2 for the corresponding references. Note that Table 2.2 may not be needed if we change the reference format

.

s.n	Compression method	Search strategy	Exact match	Approx. match	Time complexity	Space complexity
0	näive		$\checkmark$	$\checkmark$	$O(u)$	$O(n+m)$
1	RLE					
2	LZ77		$\checkmark$		$O(n\log^2(\frac{u}{n})+m)$	
3	LZ78, LZW	KMP	$\checkmark$		$O(n+m^2)$ or $O(n\log m+m)$	$O(n+m^2)$ or $O(n+m)$
4	word Huffman	BM, Shift-OR	$\checkmark$	$\checkmark$	$O(n+m)$ or $O(n+m\sqrt{u})$	$O(\sqrt{u})$
5	LZ78, LZW	d.p.	$\checkmark$	$\checkmark$	$O(mkn+r)$ $O(k^2n+\min mkn, m^2(m\Sigma)^k+r)$	$O(mkn+n\log n)$ or
6	LZW	suffix trees	$\checkmark$		$O(n+m\sqrt{m}\log m)$ or $O(nk+m^{1+\frac{1}{\alpha}}\log m), \alpha\geq 1$	
7	LZ77		$\checkmark$		$O(n+m)^6$	$O(n\log^2 u+m_c\log\log m_c$ $+n^2\log u)$
8	LZ77, LZ78	Shift-OR	$\checkmark$		$O(\min u, n\log m+r)$ or $O(\min u, mn+r)$ w.c.	$O(n+r)$
9	LZW	Shift-AND	$\checkmark$	$\checkmark$	$O(n+r)$	$O(n+m)$
10	LZ78, LZW	BM	$\checkmark$		$\Omega(n), O(mu)$ w.c.	$O(n+r)$
11	antidictionaries	KMP	$\checkmark$		$O(m^2+a+n+r)$	$O(m^2+a)$
12	gen. dictionary	BM	$\checkmark$		$O(f(d).(d+n)+n.m+m^2+r)$	$O(d+m^2)$

Table 2.2: References for Table 2.1

.

s.n	Reference
0	
1	[EV88]
2	[FT95b]
3	[ABF96a]
4	[ZMN00, MNB00]
5	[KNU00a]
6	[Kos95]
7	[GKP96a, KPR95]
8	[NR99a]
9	[KTS99]
10	[NT00]
11	[STS99]
12	[SMT00]

2.3 Indexed Search on Compressed Text

Storing and searching large volumes of data has become an important problem in the digital library age. With more and more text data being stored on the Internet on a daily basis, search engines have become a powerful tool for finding relevant documents on net. A single site (such as a library database) may also contain large collections of data and thus requires efficient search methods, even to search within the local data. The most popular query and target are still the text on the major sites that provide searching function such as Google, Yahoo, MSN, and AOL. Although the Information Retrieval (IR) scheme has be studied comprehensively, there is still large room for better efficiency and effectiveness. An important observation is that a major part of text data is stored in a compressed form. This is primarily due to the reduced storage requirement when the data is compressed. However, more recently, compression has been used as an important tool in efficient search and retrieval for large text repositories [Man97]. Here, the internal data structures used by the compression scheme are re-used at the time of search. This often results in a dramatic reduction in search time, when compared with the traditional decompress-then-search approach.

Retrieval directly on the compressed text is not the only solution to obtain the information. Usually, the compression does nothing to organize the data particularly for the purpose of information retrieval. As we stated in the previous sections, the most popular methods for text searching is to perform a keyword pattern matching in the compressed or uncompressed text. Typical pattern matching algorithms including BM and KMP etc. Although their complexity can reach as linear or even sublinear, they have to start from the beginning of the text. The preprocessing is usually executed on the patterns. Obviously, a more efficient methodology is to organize the text by extracting the structure information of the text we are searching. The most commonly used method is to build the index and search based on the index or indices. Index of a book, table of content, list of figures are the successful examples for users to find a specific keywords in a paper book. It is said that if you can not find a section in the chapter, find it before the chapter one. That is, the table of content, where you can locate the section from. The keyword index at the end session of the book help to avoid the full scan from the beginning of a page.

A typical text retrieval system is constructed as follows. First, the keywords are collected from the text database off-line and an inverted index file is built. Each entry in the index points to all the documents that contain the keyword. A popular document ranking scheme is based on the keyword frequency tf and inverted document frequency idf. When a query is given, the search engine will match the words in the inverted index file with the query. Then the document ranking information is computed according to some logic and/or frequency rules, (for example by a similarity measurement) to obtain the search results that point to the target documents. Finally, only the selected documents are retrieved. The right half of Figure 2.1 (see the area in the dotted block) shows the structure of a traditional text retrieval system.

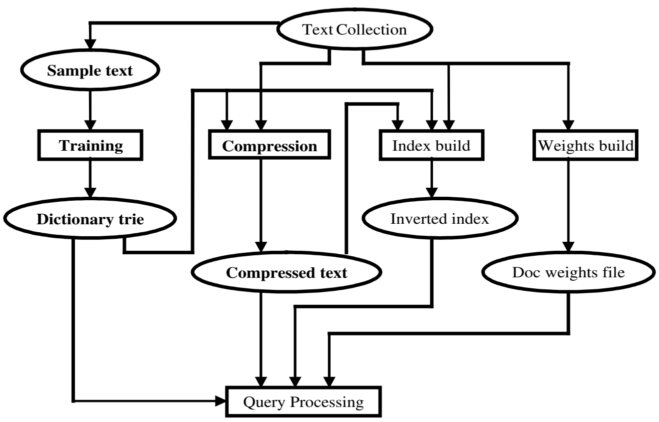


Figure 2.1: (Compressed) Text retrieval system.

There are various models, methods, data structures, or algorithms in each structure. Typical information retrieval modes include Boolean, Probabilistic, Vector models, Fuzzy models and their alternatives [BR99b]. Various data structures beside flat file are used to facilitate searching such as inverted file. signature file, graphs, hashing, and different kinds of trees. Query may also be processed to extract more useful information such as parsing, clustering, or using feedback [FB92a]. Terms in the text will be processed during indexing stage to organize the information in a more efficient way such as parsing, clustering, ranking, sorting, etc.

To facilitate information retrieval, the text is also preprocessed besides the traditional operations listed above, for instance, to incorporate auxiliary meta-data (or other indices) in the raw text. A popular example is XML, which is used for multimedia data. The auxiliary information, however, will typically increase the file size, thus compounding the problems of storage and transmission. Thus, text compression and retrieval would seem to have conflicting objectives. Furthermore, at first glance, it would appear that compression is no longer important, especially given the decreasing cost of huge-capacity storage devices. However, a good compression scheme could not only reduce the amount of data needed to be stored, but could also facilitate efficient search and retrieval directly on the compressed data. The data structures used by the compression algorithm could be exploited by a search algorithm for more effective information retrieval. In fact, it has been pointed out that compression is a key for next-generation text retrieval systems [ZMN00]. We will describe how our compression scheme incorporated with a IR system in Chapter 5

CHAPTER 3

STAR TRANSFORM FAMILY

In this chapter we present our research on new transformation techniques that can be used as preprocessing steps for the compression algorithms. A series of reversible transforms are designed to transform the text so that the existing compressors can perform with a better compression ratio.

3.1 Transform Based Methods: Star (*) transform

The basic idea is to transform the text into some intermediate form, which can be compressed with better efficiency. The transformation is designed to exploit the natural redundancy of the language. We have developed a class of such transformations, each giving better compression performance over the previous ones and most of them giving better compression over current and classical compression algorithms discussed in the previous section. We first present a brief description of the first transform called Star Transform (also denoted by *-encoding) [FM96]. We then present four new transforms called LPT, SCLPT, RLPT and

LIPT. The algorithms use a fixed amount of storage overhead in the form of a word dictionary for the particular corpus of interest and must be shared by the sender and receiver of the compressed files. The typical size of dictionary for the English language is about 0.5 MB and can be downloaded along with application programs. If the compression algorithms are going to be used over and over again, which is true in all practical applications, the amortized storage overhead for the dictionary is negligibly small.

3.1.1 Star (*) Transform

The basic idea underlying the star transformations is to define a unique signature of a word by replacing letters in a word by a special placeholder character (*) and keeping a minimum number of characters to identify the word uniquely [FM96]. For an English language dictionary D of size $60,000$ words, we observed that we needed at most two characters of the original words to keep their identity intact. In fact, it is not necessary to keep any letters of the original word as long as a unique representation can be defined. The dictionary is divided into sub-dictionaries D_s containing words of length, $1 \leq s \leq 22$, because the maximum length of a word in English dictionary is 22 and there are two words of length 1 viz 'a' and 'I'.

The following encoding scheme is used for the words in D_s : The first word is represented as sequence of s stars. The next 52 words are represented by a sequence of $s-1$ stars followed by a single letter from the alphabet $\Sigma = (a, b \dots z, A, B \dots Z)$. The next 52 words have a similar encoding except that the single letter appears in the last but one position. This will continue until all the letters occupy the first position in the sequence. The following group of words have $s-2$ *'s and the remaining two positions are taken by unique pairs of letters from the alphabet. This process can be continued to obtain a total of 53 unique encodings which is more than sufficient for English words. A large fraction of these combinations are never used; for example for $s = 2$, there are only 17 words and for $s = 8$, there are about 9000 words in English dictionary. Given such an encoding, the original word can be retrieved from the dictionary that contains a one-to-one mapping between encoded words and original words. The encoding produces an abundance of * characters in the transformed text making it the most frequently occurring character. If the word in the input text is not in the English dictionary (viz. a new word in the lexicon) it will be passed to the transformed text unaltered. The transformed text must also be able to handle special characters, punctuation marks and capitalization. The space character is used as word separator. The character ‘~’ at the end of an encoded word denotes that the first letter of the input text word is capitalized. The character ‘‘’ denotes that all the characters in the input word are capitalized. A capitalization mask, preceded by the character ‘^’, is placed at the end of encoded word to denote capitalization of characters other than the first letter and all capital letters. The character ‘\’ is used as escape character for encoding the occurrences of ‘*’,‘~’,‘‘’,‘^’, and ‘\’ in the input text. The transformed text can now be the input to any available lossless text compression algorithm, including Bzip2 where the text undergoes two transformation, first the *-transform and then a BWT transform.

3.1.2 Class of Length Preserving Transforms (LPT and RLPT)

The Length-Preserving Transform (LPT) [KM98] was invented to handle the problem that arises due to the use of run-length encoding after BWT transform is applied to the *-transformed output. It is defined as follows: words of length more than four are encoded starting with '*', this allows Bzip2 to strongly predict the space character preceding a '*' character. The last three characters form an encoding of dictionary offset of the corresponding word in this manner: entry $D_i[0]$ is encoded as "zaA". For entries $D_i[j]$ with $j > 0$, the last character cycles through $[A - Z]$, the second-to-last character cycles through $[a - z]$, the third-to-last character cycles through $[z - a]$. For words of more than four characters, the characters between the initial '*' and the final three-character-sequence in the word encoding is filled up with a suffix of the string "…nopqrstuvw". The string may look arbitrary but note that its order is the same as that of the orders of the letters in the alphabet and the suffix length is exactly 4 minus the length of the word. For instance, the first word of length 10 would be encoded as "*rstuvwzaA". This method provides a strong local context within each word encoding and its delimiters. In this scheme each character sequence contains a marker ('*') at the beginning, an index at the end, and a fixed sequence of characters in the middle. The fixed character sequence provides BWT with a strong prediction for each character in the string.

In the further study of BWT and PPM, we found that a skewed distribution of context should have better result because PPM and its alternatives should have fewer entries in the frequency table. This leads to higher probabilities and we have a shorter code length. The Reverse Length-Preserving Transform (RLPT), a modification of LPT, exploits this information. The padding part is simply reversed for RLPT. For example, the first word of length 10 would be encoded as "*wvustrzaA". The test results show that the RLPT plus PPMD, outperforms *-transform and LPT. RLPT combined with compression algorithms of Huffman, Arithmetic compress, Gzip, Bzip2 also performs better if RLPT is not used in combination with these algorithms.

3.1.3 Class of Index Preserving Transforms SCLPT and LIPT

We observed that it is not necessary to keep the word length in encoding and decoding as long as the one-to-one mappings are held between word pairs in original English dictionary and transform dictionary. The major objectives of compression algorithms such as PPM are to be able to predict the next character in the text sequence efficiently by using the deterministic context information. We noted that in LPT, The padding sequence to maintain length information can be uniquely determined by its first character. For example, a padding sequence "rstuvw" is determined by 'r' and it is possible to replace the entire sequence used in LPT by the sequence "*rzAa" and vice versa. We call this transform SCLPT (Shrotened Context LPT).We now have to use a shortened-word dictionary. If we apply LPT along with the PPM algorithm, there should be context entries of the forms "*rstu" 'v', 'stu' 'v', 'tu' 'v', 'u' 'v' in the context table and the algorithm will be able to predict 'v' at length order

5 deterministically. Normally PPMD goes up to order 5 context, so the long sequence of "*rstuvw" may be broken into shorter contexts in the context trie. In SCLPT, such entries will all be removed and the context trie will be used to reveal the context information for the shortened sequence such as "*rzAa". The result shows that this method competes with the RLPT plus PPMD combination.It beats RLPT using PPMD in 50% of the files and has a lower average BPC over the test bed. The SCLPT dictionary is only 60% of the size of the other transform dictionary, thus there is about 60% less memory use in conversion and less CPU time consumed. In general, it outperforms the other schemes in the star-encoded family. LPT has an average improvement of 4.4% on Bzip2 and 1.5% over PPMD; RLPT has an average improvement of 4.9% on Bzip2 and 3.4% over PPMD+ [TC96] using file *paper6* in Calgary corpus [Cor00a] as training set. We have similar improvement with PPMD in which no training set is used. The SCLPT has an average improvement of 7.1% on Bzip2 and 3.8% over PPMD+. For Bzip2, SCLPT has the best compression ratio in all the test files. Our results show that SCLPT has the best compression ratio in half of the test files and ranked second in the rest of the files.

A different twist to our transformation comes from the observation that the frequency of occurrence of words in the corpus as well as the predominance of certain lengths of words in English language might play an important role in revealing additional redundancy to be exploited by the backend algorithm. The frequency of occurrence of symbols, k-grams and words in the form of probability models, of course, forms the corner stone of all compression algorithms but none of these algorithms considered the distribution of the length of words directly in the models. We were motivated to consider length of words as an important factor in English text as we gathered word frequency data according to lengths for the Calgary, Canterbury [Cor00a, Cor00b], and Gutenberg Corpus [Cor]. A plot showing the total word frequency versus the word length results for all the text files in our test corpus (combined) is shown in Figure 3.1.

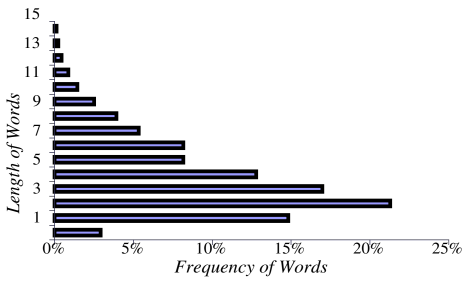


Figure 3.1: Frequency of English words versus length of words in the test corpus

It can be seen that most words lie in the range of length 1 to 10. The maximum number words have length 2 to 4. The word length and word frequency results provided a basis to build context in the transformed text. We call this Length Index Preserving Transform (LIPT). LIPT can be used as an additional component in the Bzip2 before run length encoding or simply replace it. Compared to the *-transform, we also made a couple of modifications to improve the timing performance of LIPT. For *-transform, searching for a transformed word for a given word in the dictionary during compression and doing the reverse during decompression takes time which degrades the execution times. The situation can be improved by pre-sorting the words lexicographically and doing a binary search on the sorted dictionary both during compression and decompression stages. The other new idea that we introduce is to be able to access the words during decompression phase in a random access manner so as to obtain fast decoding. This is achieved by generating the addresses of the words in the dictionary by using, not numbers, but the letters of the alphabet. We need a maximum of three letters to denote an address and these letters introduce artificial but useful context for the backend algorithms to further exploit the redundancy in the intermediate transformed form of the text. LIPT encoding scheme makes use of recurrence of same length of words in the English language to create context in the transformed text that the entropy coders can exploit.

LIPT uses a static English language dictionary of 59951 words having a size of around 0.5 MB. LIPT uses transform dictionary of around 0.3 MB. The transformation process requires two files namely English dictionary, which consist of most frequently used words, and a transform dictionary, which contains corresponding transforms for the words in English dictionary. There is one-to-one mapping of word from English to transform dictionary. The words not found in the dictionary are passed as they are. To generate the LIPT dictionary (which is done offline), we need the source English dictionary to be sorted on blocks of lengths and words in each block should be sorted according to frequency of their use.

A dictionary D of words in the corpus is partitioned into disjoint dictionaries D_i, each containing words of length i, where $i = 1, 2 \ldots n$. Each dictionary D_i is partially sorted according to the frequency of words in the corpus. Then a mapping is used to generate the encoding for all words in each dictionary D_i. $D_i[j]$ denotes the jth word in the dictionary D_i. In LIPT, the word $D_i[j]$, in the dictionary D is transformed as $*c_{len}[c][c][c]$ (the square brackets denote the optional occurrence of a letter of the alphabet enclosed and are not part of the transformed representation) in the transform dictionary D_{LIPT} where c_{len} stands for a letter in the alphabet $[a-z, A-Z]$ each denoting a corresponding length $[1-26, 27-52]$ and each c is in $[a-z, A-Z]$. If $j=0$ then the encoding is $*c_{len}$. For $j>0$, the encoding is $*c_{len}[c][c]$. Thus, for $1 \leq j \leq 52$ the encoding is $*c_{len}c$; for $53 \leq j \leq 2756$ it is $*c_{len}cc$, and for $2757 \leq j \leq 140608$ it is $*c_{len}ccc$. Thus, the 0th word of length 10 in the dictionary D will be encoded as "*j" in D_{LIPT}, $D_{10}[1]$ as "$*ja$", $D_{10}[27]$ as "$*jA$", $D_{10}[53]$ as "$*jaa$", $D_{10}[79]$ as "$*jaA$", $D_{10}[105]$ as "$*jba$" , $D_{10}[2757]$ as "$*jaaa$", $D_{10}[2809]$ as "$*jaba$", and so on.

3.1.4 StarNT

There are three considerations that lead us to this transform algorithm.

First, we gathered data of word frequency and length of words information from our collected corpora (All these corpora are publicly available), as depicted in Figure 3.1. It is clear that almost more than 82% of the words in English text have the lengths greater than three. If we can recode each English word with a representation of no more than three symbols, then we can achieve a certain kind of "*pre-compression*". This consideration can be implemented with a fine-tuned transform encoding algorithm, as is described later.

The second consideration is that the transformed output should be compressible to the backend compression algorithm. In other words, the transformed immediate output should maintain some of the original context information as well as provide some kind of "artificial" but strong context. The reason behind this is that we choose BWT and PPM algorithms as our backend compression tools. Both of them predict symbols based on context information.

Finally, the transformed codewords can be treated as the offset of words in the transform dictionary. Thus, in the transform decoding phase we can use a hash function to achieve $O(1)$ time complexity for searching a word in the dictionary. Based on this consideration, we use a continuously addressed dictionary in our algorithm. In contrast, the dictionary is split into 22 sub-blocks in LIPT [MA02]. Results show that the new transform is better than LIPT not only in time complexity but also in compression performance.

The performance of search operation in the dictionary is the key for fast transform encoding. We have used a special data structure, *ternary search tree* to achieve this objective. we will discuss the ternary tree technique and its applications in section 3.2 as well as the possible parallel processing using ternary suffix tree in section 3.1.4.

3.1.4.1 Dictionary Mapping

The dictionary used in this experiment is prepared in advance, and shared by both the transform encoding module and the transform decoding module. In view of the three considerations mentioned in section 2.1.2, words in the dictionary D are sorted using the following rules:

- Most frequently used words are listed at the beginning of the dictionary. There are 312 words in this group.
- The remaining words are stored in D according to their lengths. Words with longer lengths are stored after words with shorter lengths. Words with same length are sorted according to their frequency of occurrence.
- To achieve better compression performance for the backend data compression algorithm, only letters $[a..zA..Z]$ are used to represent the codeword.

With the ordering specified above, each word in D is assigned a corresponding codeword. The first 26 words in D are assigned 'a', 'b', ..., 'z' as their codewords. The next 26 words are assigned 'A', 'B', ..., 'Z'. The 53rd word is assigned "aa", 54th "ab". Following this order, "ZZ" is assigned to the 2756th word in D. The 2757th word in D is assigned "aaa", the following 2758th word is assigned "aab", and so on. Hence, the most frequently occurred words are assigned codewords form 'a' to "eZ". Using this mapping mechanism, totally $52 + 52 * 52 + 52 * 52 * 52 = 143,364$ words can be included in D.

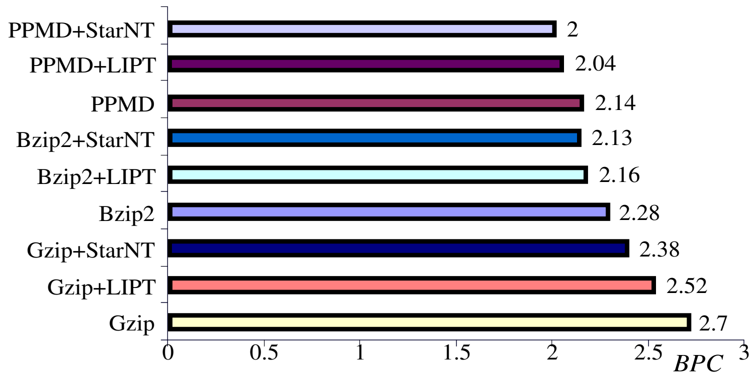


Figure 3.2: Compression ratio with/without transform

3.1.4.2 Performance Evaluation

Our experiments were carried out on a 360MHz Ultra Sparc-IIi Sun Microsystems machine housing SunOS 5.7 Generic_106541 − 04. We choose Bzip2 (−9), PPMD (order 5) and Gzip (−9) as the backend compression tool. Facilitated with our proposed transform algorithm, Bzip2 −9, Gzip −9 and PPMD all achieve a better compression performance in comparison to most of the recent efforts based on PPM and BWT. Figure 3.2 shows that, for Calgary corpus, Canterbury corpus and Gutenberg corpus, StarNT achieves an average improvement in compression ratio of 11.2% over Bzip2 −9, 16.4% over Gzip −9, and 10.2% over PPMD. This algorithm utilizes *Ternary Search Tree* in the encoding module. With a finely tuned dictionary mapping mechanism, we can find a word in the dictionary at time complexity $O(1)$ in the transform decoding module. Results shows that for all corpora, the average compression time using the transform algorithm with Bzip2 −9, Gzip −9 and PPMD is 28.1% slower, 50.4% slower and 21.2% faster compared to the original Bzip2 −9, Gzip −9 and PPMD re-

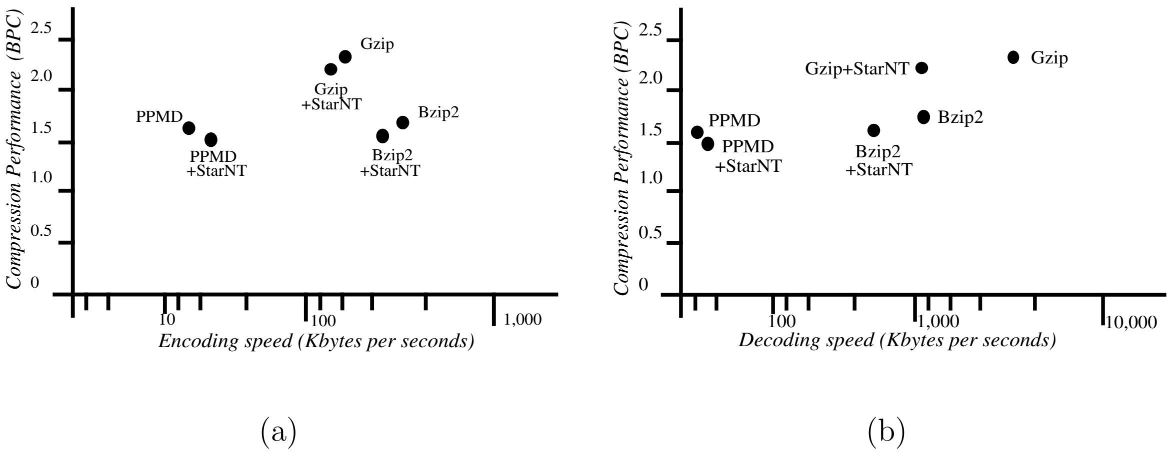


Figure 3.3: Compression effectiveness versus (a) Compression (b) Decompression speed

spectively. The average decompression time using the new transform algorithm with Bzip2 −9, Gzip −9 and PPMD is 100% slower, 600% slower and 18.6% faster compared to the original Bzip2 −9, Gzip −9 and PPMD respectively. Figure 3.3 illustrates the compression ratio vs. compression/decompression speed for the different algorithms. However, since the decoding process is fairly fast, this increase is negligible.We draw a significant conclusion that Bzip2 in conjunction with StarNT is better than both Gzip and PPMD both in time complexity and compression performance.

Based on this transform, we developed StarZip, a domain-specific lossless text compression utility for archival storage and retrieval. StarZip uses specific dictionaries for specific domains. In our experiment, we created five corpora from publicly available website, and derived five domain-specific dictionaries. Results show that the average BPC improved 13% over bzip2 -9, 19% over Gzip −9, and 10% over PPMD for these five corpora.

3.2 Search Techniques For Text Retrieval

We have described the text compression using star transform family in section 3.1. The performance of search operation in the dictionary mapping is the key for fast transform encoding. Our initial encoding and decoding speed is slower than traditional compression algorithms without transformation in an order of magnitude. The remove the bottleneck for the real world application, we have used a special data structure, *ternary search tree* to achieve this objective.

3.2.1 Ternary Search Tree for Dictionary Search

Ternary search trees are similar to digital search tries in that strings are split in the trees with each character stored in a single node as *split char*. Besides, three pointers are included in one node: left, middle and right. All elements less than the split character are stored in the left child, those greater than the split character are stored in the right child, while the middle child contains all elements with the same character.

Search operations in ternary search trees are quite straightforward: current character in the search string is compared with the split char at the node. If the search character is less than the split char, then go to the left child; if the search character is greater than the split char, go to the right child; otherwise, if the search character is equal to the split char, just go to the middle child, and proceed to the next character in the search string. Searching for a string of length k in a ternary search tree with n strings will require at most $O(\log n + k)$ comparisons. The construction time for the ternary tree takes $O(n \log n)$ time [BS97].

Furthermore, ternary search trees are quite space-efficient. In Figure 3.4, seven strings are stored in this ternary search tree. Only nine nodes are needed. If multiple strings have same prefix, then the corresponding nodes to these prefixes can be reused, thus memory requirements is reduced in scenarios with large amounts of data.

In the transform encoding module, words in the dictionary are stored in the ternary search trees with the address of corresponding codewords. The ternary search tree is split into 26 distinct ternary search trees. An array is used to store the addresses of these ternary search trees corresponding to the letters $[a..z]$ of the alphabet in the main root node. Words having the same starting character are stored in same sub-tree, viz. all words starting with ‘a’ in the dictionary exist in the first sub-tree, while all words start with ‘b’ in second sub-tree, and so on.

In each leaf node of the ternary search tree, there is a pointer which points to the corresponding codeword. All codewords are stored in a global memory that is prepared in

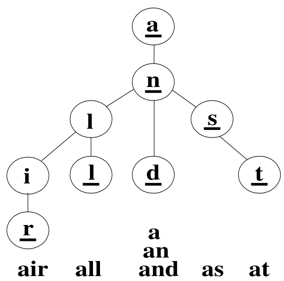


Figure 3.4: A Ternary Search Tree

advance. Using this technique we can avoid storing the codeword in the node, which enables a lot of flexibility as well as space-efficiency. To expedite the tree-build operation, we allocate a big pool of nodes to avoid overhead time for allocating storage for nodes in sequence.

Ternary search tree is sensitive to insertion order: if we insert nodes in a good order (middle element first), we end up with a balanced tree for which the construction time is small; if we insert nodes in the order of the frequency of words in the dictionary, then the result would be a skinny tree that is very costly to build but efficient to search. In our experiment, we confirmed that insertion order has a lot of performance impact in the transform encoding phase. Our approach is just to follow the *natural* order of words in the dictionary. Result shows that this approach works very well (see section 3.1.4.2).

3.2.2 Ternary Suffix Tree

There are many applications where a large, static text database is searched again and again. A library database is one such application. The user can search for books based on keywords, author names, title or subject. A library database handles thousands of requests per day. Modifications to the data are relatively rare. In such applications, an efficient way of searching the database is necessary. Linear pattern matching techniques, though very efficient, are not applicable in such cases, as they take time proportional to the size of the database. The database can be very big, and handling thousands of requests becomes a huge problem.

For such applications, we need to store the data in some sort of pre-processed form, in order to be able to handle the search requests efficiently. Building index files and inverted indices is one solution. But this might need a lot of 'manual' effort for maintenance. Some one has to decide what words to include in the index. Besides, words that are not in the index can not be searched.

There are some data structures to handle these situations appropriately. Suffix trees and binary search trees are the most popular ones. Suffix trees have very efficient search performance - search takes $O(n)$ time, where n is the length of the pattern being searched. But suffix trees require huge amounts of storage space - they typically require around $24m$ to $28m$ space, where m is the size of the text(the dictionary or the database, in this case). Another alternative is to use the suffix array. The suffix array takes much lesser space than the suffix tree. The search procedure involves a binary search for the pattern within the suffix array. The search performance is much slower than that of a suffix tree. Therefore, we need a data structure that requires lesser space than the suffix tree, but gives better search performance than the binary tree. *Ternary suffix tree* is such a data structure.

A ternary suffix tree is nothing but ternary search tree built for all the suffixes of an input string/file. In the ternary suffix tree, no branch is extended beyond what is necessary to distinguish between two suffixes. Therefore, if the ternary tree is constructed for a dictionary, as each term in the dictionary is unique, the ternary suffix tree will effectively be a ternary tree for the words in the dictionary.

3.2.3 Structure of Ternary Suffix Trees

Each node in a ternary tree has three children - the lesser child, the equal child, and the greater child. The search path takes the lesser, equal or greater child depending on whether the current search key is lesser, equal to or greater than the *split char* at the current node. The node stores pointers to all the three children.

In order to be able to retrieve all the occurrences of a key word, we need two more pointers at each node - *begin* and *end*. *Begin* and *end* correspond to the beginning and ending indices of the range of suffixes that correspond to the node. Therefore, begin is equal to end for all leaf nodes, as the leaf node corresponds to a unique suffix.

The begin and end pointers serve two purposes - firstly, they eliminate the necessity to store the rest of the suffix at every leaf in the tree. These pointers can be used to go to the exact location in the text corresponding to the current suffix and compare directly with the text if the search for a string reaches a leaf node. Therefore, we need to store the text in memory only once.

Secondly, the *begin* and *end* pointers help in finding multiple occurrences of a string. If the search for a string ends at an intermediate node, all the suffixes in the suffix array from *begin* to *end* match the string. There fore the string occurs $(end - begin + 1)$ number of times in the text, and the locations of the occurrences are the same as the starting positions of the corresponding suffixes.

Table 3.1: Suffixes and Suffix array for the text T = "abrab$"

	Suffixes	Suffix array Hrs
1	$	6
2	ab$	4
3	abrab$	1
4	b$	5
5	brab$	2
6	rab$	3

3.2.4 Construction of Ternary Suffix Trees

The construction of the ternary suffix tree requires the sorted suffixes, or the suffix array. The sorted suffixes can be obtained in many ways, depending on the form of the input text. If the input is uncompressed text, we can do a quick sort on all the suffixes, which takes $O(mlogm)$ time. If the input is BWT-compressed text, we can use the suffix array (Hrs) data structure constructed during the decompression process in BWT [AMB02]. Table 1 gives an example of the suffixes and corresponding suffix array entry of the string *"abrab$"*.

Once we have the sorted suffixes, the ternary tree can be constructed using different approaches. Here, we consider the median approach and the mid-point approach. In the median approach, the median of the list of characters at the current depth in all the suffixes represented by the node is selected to be the split char. i.e, there will be the same number of nodes on either side that are at the same depth as that of the current node. In the midpoint approach, we select the suffix that is located exactly in the middle of the list of suffixes corresponding to the current node. Calculation of the median requires a scan through all the suffixes corresponding to the current node, where as calculating the midpoint is a single operation. Therefore, construction of a ternary suffix tree based on the median approach is significantly slower than the construction based on midpoint approach. However, the median approach results in a tree that is alphabetically more balanced - at every node, the lesser child and the greater child correspond to approximately the same number of symbols of the alphabet. The midpoint approach results in a tree that is more balanced population wise - the lesser sub tree and the greater sub tree have nearly the same number of nodes.

We provide a comparison between the two approaches. The tree construction is around 300% faster for the midpoint approach as compared to the median approach. The search is around 5% faster for the midpoint approach compared to the median approach.

3.2.4.1 Searching for strings

The search procedure involves the traversal of the ternary tree according to the given pattern. To begin, we search for the first character of pattern, starting at the root. We compare the first character of the pattern with the split char at the root node. If the character is smaller than the split char, then we go the lesser(left) child, and compare the same character with the split char at the left child. If the search character is greater than the splitchar, we go to the greater(right) child. If the search character matches the splitchar, we take the equal(middle) child, and advance the search to the next character in the pattern. At any node, if the desired greater child or lesser child is null, it implies that the pattern is not found in the text. If the search advances to the last character in the given pattern, and if the search path reaches a node at which that character matches the splitchar, then we found the pattern. The pattern occurs in the text at locations $Hrs[begin]$, $Hrs[begin+1]$, and $Hrs[end]$.

3.2.5 Implementation

3.2.5.1 Elimination of leaves

The leaves of the ternary tree do not resolve any conflicts. Therefore, all the leaves can be eliminated. This, however would require slight modifications in the search procedure. If the desired child is a lesser child or greater child, and that child was null, we have to go to the location in the text which would correspond to that leaf if that leaf existed, and linearly compare the text and the pattern. This location is easy to calculate: The leaf, either lesser or greater, corresponds to only one location in the text. Therefore, for the lesser child, this location has to be $Hrs[begin]$, and for the greater child this has to be $Hrs[end]$, where begin, end belong to the current node.

On a test run, for a file of size 586,000 characters, this optimization saved 385,000 nodes, which is a saving of almost 14 bytes per character.

Figure 3.5(a) shows the ternary suffix tree for the string "abrab$". Figure 3.5-(a) shows the ternary suffix tree after eliminating the leaves. The edges to equal children are repre-

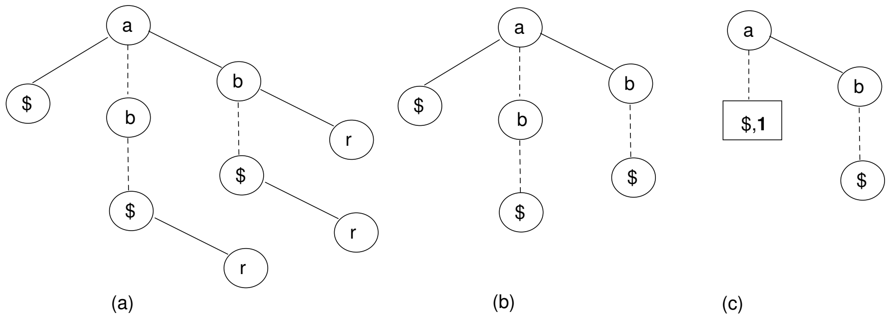


Figure 3.5: Ternary tree example

(a)Ternary suffix tree for "abrab$" (b) The tree after eliminating leaves (c) The after eliminating leaves and applying path compression.

sented by a broken line.

3.2.5.2 Path compression

For a node, if both the lesser and greater leaves are null, and if the equal child is not null, then it is not necessary to have that node in the tree. The node can be replaced by its equal child if the equal child can some how store the number of nodes eliminated in this fashion between its immediate parent and itself. We call this as *path compression.*

Therefore, the nodes at which path compression has been implemented can be treated as special nodes. These nodes store an extra integer (denoted by pclength), which stores the path compression length. When the search path reaches any of these special nodes, The

Table 3.2: Comparison of search performance

File	size	Binary Search		Ternary suffix tree			
		Time(ms)	Comparisons	Total Time	Search Time	Construction Time	Search Comparisons
Alice29.txt	152089	16000	17312160	12800	10440	2360	17233990
Anne11.txt	588960	56330	24285300	20260	11920	8340	19592500
Bib.txt	111261	15460	17184600	13570	11020	2550	17384840
Book2	610856	25890	24557380	22420	11770	10650	19235470
Lcet10.txt	426754	23930	23282720	19070	11150	7920	18340670
Plrabn12.txt	481861	25470	24497200	18500	12230	6270	20095030
News	377109	22630	22441110	31110	12370	18740	19589940
World95.txt	2988578	38970	31760050	120010	13230	106780	23196930
1musk10.txt	1344739	33340	28236560	32300	12780	19520	21010900

comparison has to be done in the text for a length equal to pclength before continuing the search from the current node. The comparison in the text can be done at any of the locations corresponding to $Hrs[begin]$, $Hrs[end]$ of the special node.

On a test run, for a file of size 586,000 characters, this optimization saved 355,000 nodes, which is a saving of almost 10 bytes per character of the original text.

Figure 3.5-(c) shows the ternary suffix tree after applying path compression to the tree in figure 3.5-(b). The special nodes are represented by rectangular boxes. The path compression length is indicated in bold type.

3.2.6 Results

For the test file mentioned above, the total number of nodes in the tree were around 549,000. Out of these special nodes were around 128,000, requiring 24 bytes per node. Therefore, the total memory used was around 19.6 bytes per character. Other files produced similar results, using around 20 bytes per character. We compared the search performance of the ternary suffix tree with the that of binary search in the suffix array. The results are shown in 3.2. Ternary suffix trees were built for different files in the calgary corpus. The construction time shown in the table is the construction time required to build the ternary suffix tree from the suffix array of each file. The results are based on searching each word from a dictionary of 59,951 English words, searching each word 10 times. It can be seen form the results that the search performance of the ternary suffix tree is much better than that of binary search in the suffix array. It can also be seen that the extra construction time required for building the ternary is more than compensated for over a large number (approximately 600,000) searches.

CHAPTER 4

COMPRESSED PATTERN MATCHING ON BURROW-WHEELER TRANSFORMED TEXT

4.1 Problem Description

With the increasing amount of text data available, most of these data are now typically stored in a compressed format. Thus, efforts have been made to address the *compressed pattern matching problem*. Given a text string T, a search pattern P, and Z the compressed representation of T, the problem is to locate the occurrences of P in T with minimal (or no) decompression of Z. Different methods have equally been proposed [ABF96b, Man97, FT98, FT95a, GKP96b, NR99b]. The motivation includes the potential reduction of the delay in response time introduced by the initial decompression of the data, and the possible elimination of the time required for later compression. Another is the fact that, with the compact representation of data in compressed form, manipulating such smaller amounts of data directly will inherently lead to some speedup in certain types of processing on the data.

Initial attempts at compressed pattern matching were directed towards compression schemes based on the Lempel-Ziv (LZ, for short) family of algorithms [ZL77, ZL78] where algorithms have been proposed that can search for a pattern in an LZ77-compressed text string in $O(n \log^2(\frac{u}{n}) + m)$ time, where $m = |P|, u = |T|$, and $n = |Z|$ [FT98]. The focus on LZ might be attributed to the wide availability of LZ-based compression schemes on major computing platforms. For example, GZIP and COMPRESS (UNIX), PKZIP (MSDOS) and WINZIP (MS WINDOWS) are all based on the LZ algorithm. In general, the LZ-family are relatively fast, but they do not produce the best results in terms of data compression. On average, the PPM (*prediction by partial matching*)-family of algorithms [CW84, Mof90a, CT97] provide the best performance in terms of compaction. They are, however, generally slow. Methods for pattern matching directly on data compressed with non-LZ methods have also been proposed as we discussed in Chapter 2.

Burrows and Wheeler [BW94b] presented a transformation mechanism (usually called the Burrows-Wheeler Transform (BWT), or block-sorting) as a basis for a compression algorithm that is close to the PPM-family in terms of compression performance, but close the LZ family in complexity. Their method is based on a simple permutation of the input text, and subsequent encoding of the permuted output.

With this middle-ground performance, the BWT becomes an important approach to data compression, especially where there is need for significant compression ratios with fast compression or decompression. However, not much work has been done on compressed pattern matching for the BWT. So far, only *off-line* exact-pattern matching algorithms [FM00, FM01, Sad00] have been proposed for searching directly on BWT-compressed data.

Although there has been substantial work in compressed pattern matching [ZMN00, MNZ00, KNU00b, KTS99], not much work has been done on searching directly on BWT-compressed text [1]. So far, mainly index-based algorithms [FM00, FM01, Sad00] have been proposed for searching directly on BWT-compressed data. These methods, as with most compressed pattern matching algorithms, have focused mainly on exact pattern matching. One exception is the work reported in [MNU01], where they considered approximate pattern matching, when the text is coded using run-length encoding (RLE).

Different data structures, (such as sorted link list, binary search tree, suffix tree, suffix arrays, etc.) can be used to perform fast searching on text. The motivation for our approach is the partial sorted nature of the output string from the Burrows Wheeler Transform, and the potential for constructing some important search data structures from the BWT compressed sequence. In our work, we developed methods for efficient exact pattern matching on BWT-transformed text [AMB02, BPM02]. We make the following contributions:

Given a text string $T = t_1 t_2 \ldots t_u$, a pattern $P = p_1 p_2 \ldots p_m$, over a symbol alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$. Let Z be the BWT-transformed output for T:

[1] In our work, we use the terms "BWT-transformed text" and "BWT-compressed text" interchangeably. More precisely, the BWT-compressed text corresponds to the final encoded output of the BWT compression pipeline. The BWT-transformed text corresponds to the direct output of BWT, before subsequent encoding stages, see Section 4.2.2. Given the BWT-compressed text, we can obtain the BWT-transformed text by partial decompression in linear time.

1. We propose algorithms for exact pattern matching on a text sequence, after the text has been transformed using the BWT. The final algorithm (QGREP) performs exact pattern matching in $O(m+\eta_{occ}+\log\frac{u}{|\Sigma|})$ time on average, and $O(m+\eta_{occ}+\log u)$ worst case, where η_{occ} is the number of occurrences of the pattern. Each algorithm requires an $O(u)$ auxiliary arrays, which are constructed in $O(u)$ time. Experimental results show orders of magnitude improvement in search time when compared with standard text search algorithms.

2. We extend the exact pattern matching algorithm to an algorithm that can locate all the k-mismatches of P in T, using Z, in $O(uk\log\frac{u}{|\Sigma|})$ time.

3. Based on the exact pattern matching algorithm, we develop an algorithm to locate the k-approximate matches of P in T, using Z, in $O(|\Sigma|\log|\Sigma| + \frac{m^2}{k} + m\log\frac{u}{|\Sigma|} + k\alpha)$ time on average, $(\alpha \leq \eta_h(m+2k), \alpha \leq u)$, after an $O(u)$ preprocessing on Z, where η_h is the number of hypothesized potential matches.

4. Compression with the BWT is usually accomplished in four phases, viz:

   input $\rightarrow$ BWT $\rightarrow$ MTF $\rightarrow$ RLE $\rightarrow$ VLC $\rightarrow$ output,

   where, we have BWT — the forward BWT transform producing an output string L of length u ; MTF — move-to-front encoding [BST86b] to further transform L for better compression (this usually produces runs of the same symbol); RLE — run length encoding of the runs produced by the MTF; and VLC — variable length coding of the RLE output using entropy encoding methods, such as Huffman or arithmetic coding.

We propose an alternative approach to the move-to-front algorithm which enables pattern matching directly using the output of the MTF stage. To our knowledge, this is the first attempt to perform pattern matching at a stage beyond the direct output from the BWT stage.

4.2 Compressed Pattern Matching on BWT Text

In 1994, Burrows and Wheeler [BW94b] presented a transformation mechanism (usually called the Burrows-Wheeler Transform (BWT), or block-sorting) as a basis for a compression algorithm. Their method is based on a permutation of the input text, and subsequent encoding of the selected output. In terms of data compression, empirical evidence [BW94b, Fen96c, BKS99] shows that the BWT is significantly superior to the more popular LZ-based methods (such as GZIP and COMPRESS), and is only second to the PPM* algorithm [CT97]. In terms of running time, the BWT is much faster than the PPM*, but comparable with the algorithms from the LZ-family. (See the web site for the standard copus[2] for results on empirical comparison of various text compression algorithms). Since the initial proposal, there have been a number of modifications and improvements on the original algorithm [KB00, Lar99, Fen96a, Fen96c, BK00, Sew01, Yam02, Yok97]. Although the empirical performance of the BWT has been one of its major strengths, its theoretical performance has also been analyzed [Man99, BK00, EV02]. The BWT thus forms a middle

[2]The Canterbury Corpus: http://corpus.canterbury.ac.nz

ground between the superior compression ability of the PPM*, and the fast compression time of the LZ-family. The makes the BWT an important approach to data compression, especially where there is need for significant compression ratios with fast compression or decompression.

4.2.1 Related Works

The BWT can be computed using suffix trees, and/or context trees, and such data structures could be made available to the decoder at a minimal cost [Lar99, BK00]. This is based on the observation that when the forward BWT transformation matrix M is sorted to produce the output L, it implicitly induces a lexicographic order on the suffixes of the input string. (The BWT is discussed in more detail in Section 4.2.2). Cleary and Teahan [CT97] used this observation to relate the BWT to the unique contexts used in PPM. In [Eff00], similar ideas were used to propose a PPM*-based compression algorithm, with the good compression performance of PPM, but requiring only the linear time complexity of the BWT.

This important relationship may be exploited to search for a pattern in the compressed text, especially if the special data structures are available to the decoder, or if the text is preprocessed off-line before searching can begin. This line of work has been explored by Sadakane [SI99, Sad00] and Ferragina and Manzini [FM00, FM01] who used suffix trees and suffix arrays to incorporate search structures as part of the compressed text. They achieved sub-linear search complexity, but at the cost of reduced performance in data compression.

Both methods used the suffix array introduced by Manber and Meyer [MM93] as the basic data structure for representing the transformed text. A suffix array is simply a sorted index of all the suffixes of a given string. This is exactly the same order that we will get if we consider the suffixes in the sorted rows of the M matrix of cyclic shifted strings in the BWT. With the suffix array, Manber and Meyer were able to locate a pattern in a text in of $O(m+\log u+\eta_{occ})$ time [MM93], where η_{occ} is the number of the occurrences of the pattern. Compressed suffix arrays have been studied by Grossi and Vitter [GV00] who used it to perform text searches in $O(m+\log^{1+\epsilon} u+\eta_{occ}\log^{\epsilon} u)$, $0 \leq \epsilon \leq 1$.

In [Sad00], the compressed suffix array proposed by Grossi and Vitter [GV00] was used to build index structures for searching BWT-compressed text. Using the compressed suffix-array search index, searching for the η_{occ} occurrences of a pattern P in the text T was accomplished in $O(m\log u+\eta_{occ}\log^{\epsilon} n)$ time, while partial decompression for an l-length substring was done in $O(l+\log^{\epsilon} n)$. Ferragina and Manzini proposed the use of a precomputed search index (called the *FM-Index*) for searching the text. The FM-Index is basically a compressed suffix array and some auxiliary data structures. It also contains the L array — the transformed output from the BWT. Using a special partitioning scheme and a marking strategy, they proposed an $O(m+\eta_{occ}\log^2 u)$ algorithm to retrieve all the occurrences of the pattern in the text. They also suggested a modification of the above scheme that leads to an $O(m+\eta_{occ}\log^{\epsilon} u)$ time algorithm.

Our work is more closely related to those of Ferragina & Manzini and of Sadakane & Imai. An important difference is that, the previous methods used special data structures and search indices (based on suffix trees [McC76] and suffix arrays [MM93, GV00]), **which are pre-computed and stored along with the compressed data**. Although our methods make use of some auxiliary transformation vectors, we compute these at the time of search, rather than at the time of text compression. Thus, we make no assumptions about the compressed data, except that it needs to be compressed with the BWT.

We also note that the above methods considered only the exact pattern matching problem. And they basically operate on the BWT output. That is, they still have to perform the inverse VLC and MTF. In this thesis, we also consider solutions to the problem of pattern matching with errors (for both k-mismatch and k-approximate matches) using the BWT output. We also describe how this can be performed at a later stage in the BWT compression pipeline — i.e. using the output from the MTF phase (rather than the direct BWT output.

4.2.2 The Burrows-Wheeler Transform

BWT-compression is one in the family of block-sorting compression algorithms, where a block of data is coded at a time. The block size could be as large as the original file, or smaller. Basically, the BWT performs a permutation of the characters in the text, such that characters in lexically similar contexts will be near to each other. This re-arrangement is then exploited for compression by using a coder that assigns short codes to recently seen symbols. Important procedures in BWT-based compression/decompression are the forward and inverse BWT, and the subsequent encoding of the permuted text.

4.2.2.1 The Forward Transform

Given an input text $T = t_1 t_2 \ldots t_u$, the forward BWT is composed of three steps:

1. Form u permutations of T by cyclic rotations of the characters in T. The permutations form a $u \times u$ matrix M', with each row in M' representing one permutation of T.
2. Sort the rows of the permutation matrix M' lexicographically to form another matrix M. M includes T as one of its rows.
3. Record L, the last column of the sorted permutation matrix M, and id, the row number for the row in M that corresponds to the original text string T.

```
     M'                            M
                          (F)            (L)
  mississippi              i  mississip   p
  ississippim              i  ppimissis   s
  ssissippimi              i  ssippimis   s
  sissippimis              i  ssissippi   m
  issippimiss              m  ississipp   i    <<
  ssippimissi              p  imississi   p
  sippimissis              p  pimississ   i
  ippimississ              s  ippimissi   s
  ppimississi              s  issippimi   s
  pimississip              s  sippimiss   i
  imississipp              s  sissippim   i
```

The output of the BWT is the pair, (L, id). Generally, the effect is that the contexts that are similar in T are made to be closer together in L. This similarity in nearby contexts can be exploited to achieve compression. An example is given below for the input string *mississippi*. F and L denote the array of *first* and *last* characters respectively, and the "<<" marker shows the row in M that corresponds to the original text, T . For the above example, the output of the transformation will be the pair: (*pssmipissii*, 5).

4.2.2.2 The Inverse Transform

The BWT is reversible. It is quite striking that given only the (L, id) pair, the original text can be recovered exactly. This reversibility is based on three facts: the BWT output L is just a permutation of T; the first character in each row of M can be obtained easily by sorting L; and since M contains cyclic shifts of the same string (T), for any given row, j, $L[j]$ cyclically precedes $F[j]$ in T. With these observations, the inverse transformation can be performed using the following steps [BW94b]:

1. Sort L to produce F, the array of first characters.

2. Compute V, the *transformation vector* that provides a one-to-one mapping between the elements of L and F, such that $F[V[j]] = L[j]$. That is, for a given symbol $\sigma \in \Sigma$, if $L[j]$ is the c-th occurrence of σ in L, then $V[j] = i$, where $F[i]$ is the c-th occurrence of σ in F.

3. Generate the original text T, using the third observation: Since the rows in M are cyclic rotations of each other, the symbol $L[i]$ cyclically precedes the symbol $F[i]$ in T.

Combine this with the previous step, we have that, $L[V[j]]$ cyclically precedes $L[j]$ in T.

For the example with *mississippi*, we will have $V = [6\ 8\ 9\ 5\ 1\ 7\ 2\ 10\ 11\ 3\ 4]$. Given V and L, we can generate the original text by iterating with V. This is captured by a simple algorithm:

$$T[u + 1 - i] = L[V^i[id]], \forall i = 1, 2, \dots u$$

where, $V^1[s] = s$; and $V^{i+1}[s] = V[V^i[s]]], 1 \leq s \leq u$.

In practical implementations, the transformation vector V is computed by use of two arrays of character counts $C = c_1, c_2, \dots c_{|\Sigma|}$, and $R = r_1, r_2, \dots r_u$:

$$V[i] = R[i] + C[L[i]], \forall i = 1, 2, \dots u,$$

where, for a given index, c, $C[c]$ stores the number of occurrences in L of all the characters preceding σ_c, the c-th symbol in Σ. $R[j]$ keeps count of the number of occurrences of character $L[j]$ in the prefix $L[1, 2, \dots, j]$ of L. If we have V, we can use the relation between L, F, C, and V to avoid the sorting required to obtain F. Thus, we can compute F in $O(u)$ time.

4.2.3 Auxiliary Arrays

We now introduce data structure and other information derived from the inverse BWT to facilitate pattern matching. The motivation for our approach is the observation that the

BWT provides a lexicographic ordering of the input text as part of its inverse transformation process. The decoder only has limited information about the sorted context, but it is possible to exploit this via the auxiliary arrays establishing the relation between the F array and the text T. We will discuss how to compute these arrays at the output of the BWT and MTF stages respectively.

Given F and L, the characters in L followed by the corresponding characters in F constitute the set of *bi-grams* for the original text sequence T. Let $\mathcal{Q}_2^T$ and $\mathcal{Q}_2^P$ be the set of bi-grams for the text string T and the pattern P, respectively. We can use these bi-grams for at least two purposes:

Pre-filtering. To search for potential matches, we consider only the bi-grams that are in the set $\mathcal{Q}_2^T \cap \mathcal{Q}_2^P$. If the intersection is empty, it means that the pattern does not occur in the text, and we do not need to do any further decompression.

Approximate pattern matching. We can obtain the q-grams, $2 \leq q \leq m$ and perform q-gram intersection on $\mathcal{Q}_q^T$ and $\mathcal{Q}_q^P$ — the set of q-grams from T and P. At a second stage we verify if the q-grams in the intersection are part of a true k-approximate match to the pattern.

Example. Suppose $T = abraca$, and $P = rac$. We will have $L = caraab$, and $F = aaabcr$. Using F and L, we can obtain the bi-grams: $\mathcal{Q}_2^T = \{ac, ab, br, ca, ra\}$. For P, $\mathcal{Q}_2^P = \{ra, ac\}$. Intersecting the two, we see that only $\{ra, ac\}$ are in the intersection. For exact pattern matching, ac will be eliminated, and thus we will only need to check in the area in T that contains ra, since any match must contain ra. Suppose we had $P = abr$ as the pattern, the intersection will produce $\{ab, br\}$, eliminating the other potential starting points in F that also started with a. $\diamondsuit$

In traditional pattern matching, complete decompression to get back the original input text would be required before pattern matching can begin. We avoid the need for complete decompression and obtain the q-grams by use of some auxiliary transformation arrays.

4.2.3.1 Auxiliary Arrays from BWT Output

The inverse BWT transformation is defined as: $\forall_{i=1,2,\ldots,u}, T[u+1-i] = L[V^i[id]]$, where $V^i[s] = V[V[\ldots V[s]]]$ (i times) and $V^1[s] = s$. Since $V^i[z]$ is just one more indirection on $V^{i-1}[z]$, we can reduce the time required by storing the intermediate results, to be used at the next iteration of the loop. We can use an auxiliary array $G[i]$ to hold the intermediate steps of the indexing using V. That is, $G[i] = V^i[id]$. Then, $T[i] = L[G[u+1-i]]$. For more straight forward indexing, we can reverse G from Gr, and then have: $T[i] = L[Gr[i]]$. The array Gr forms a one-to-one mapping between T and L.

We will need to search on L or F to determine where pattern matching can begin or end. Since F is already sorted, and $F[z] = L[V[z]]$, we can use a mapping between T and F, (rather than L), so that we can use binary search on F. We use an equivalent auxiliary array H (and its reverse, Hr):

$\forall i, i = 1, 2, \ldots, u$

$H[i] = V[G[i]]$, and $T[i] = F[H[u+1-i]]$, or

$Hr[i] = V[Gr[i]]$, and $T[i] = F[Hr[i]]$

Hr (also H) represents a one-to-one mapping between F and T. By simply using F and Hr, we can access any character in the text, without using T itself — which is not available without complete decompression. Notice also that we do not need G to compute H.

Let array Hrs denote the sorted index of Hr. Equivalently, Hrs can be viewed as the inverse of Hr, viz: $F[i] = T[Hrs[i]]$. The procedure below generates the Hr and Hrs mapping vectors from V.

Algorithm 4.2.1 Compute the Hr and Hrs arrays

Compute-Auxiliary-Arrays(V, id)

$x \leftarrow id$
for $i \leftarrow 1$**to** u **d**o
    $x \leftarrow V[x]$
    $Hr[u-i+1] \leftarrow x$
    $Hrs[x] \leftarrow u-i+1$
end for

Example. The mapping vectors are shown below for $T = abraca$, $u = |T| = 6$, $id = 2$.

idx	T	L	F	V	Gr	Hr	Hrs
1	a	c	a	5	4	2	6
2	b	a	a	1	6	4	1
3	r	r	a	6	3	6	4
4	a	a	b	2	5	3	2
5	c	a	c	3	1	5	5
6	a	b	r	4	2	1	3

4.2.3.2 Auxiliary Arrays from MTF Output

Existing compressed pattern matching algorithms based on the BWT need some partial decompression at the VLC and MTF stages to derive the string L. The BWT output has a good locality property and it is easy to compute the auxiliary arrays through which we can access any part of the text. The outputs of the VLC or MTF stages do not have such a property unless we add some markers or extra information in the encoded stream. This could significantly degrade the compression performance. In this section, we propose a modification of the MTF stage, so as to construct the auxiliary arrays directly from the MTF output. The existence of algorthms for matching at the VLC outputs (such as on Huffaman codes [ZMN00, MNZ00]) imply that, with such a modification of the MTF algorithm, we can now perform pattern matching at any stage of the BWT-compression pipeline, including directly at the output of the VLC stage.

The MTF algorithm is used to transform the BWT output (L, id) so that the resulting output will contain mainly a sequence with small numbers. The sequence can then be compressed using a variable length coding scheme, such as Huffman or arithmetic coding. We cannot, however, compute the auxiliary arrays directly from the MTF output. We have to completely decode the MTF results before we can compute the arrays F, V, Hr and Hrs, etc. Below, we modifiy the MTF algorithm, in order to compute the auxiliary arrays faster.

We modify the forward MTF to produce two output sequences (M_1, M_2) for string L, viz.

1. Given the BWT output sequence, if a given symbol is different from the preceeding symbol, we output a ' **1**' to the sequence M_1, and output its position δ_i in Σ to M_2.
2. Otherwise, output a ' **0**' to M_1 only.

M_1 is a binary sequence, where a **1** represents a change in symbol, while a **0** indicates a run of the same symbol. M_2 maintains the ordering information for the appearance of distinct symbols. For example, with $\Sigma = \{a, b, c\}$ and the position codes representing the alphabet set to $\{0, 1, 2\}$, the modified move-to-front coding for string "*bbaaaaccccbbbaaabbb*" will be the strings ("1010001000100100100", "102101"). There is no actual move-to-front operation for the most recently used symbol. The decoding is straightforward. Based on string M_1, if a $'1'$ is encountered, look for the next symbol in M_2 and output the corresponding symbol in Σ. If a $'0'$ is encountered, simply repeat the previous symbol. At first glance, M_1 is of length u and M_2 is the extra sequence that makes no compression at all. But M_1 is a binary sequence which can be further compressed by a high efficiency algorithm such as run-length encoding (RLE) or Dynamic Markov Coding (DMC). In our experiments, we tested the use of GZIP, BZIP2, ARITHMETIC, and DMC in compressing the modified MTF output, M_1 and M_2. The best compression ratio was obtained by using DMC on M_1, and compressing M_2 with MTF again, and then with VLC. Our implementation of the new MTF lead to a compression performance of 2.23 $b$pc. This can be compared with the original 2.30 $b$pc on the test corpus, using the BSMP program.

As stated in section 4.2.3.1, we need array V to compute Hr and Hrs. And we need the C array to compute V, where $C = (c_0, c_1, ..., c_{|\Sigma|-1})$ is the character count array . $C[c]$ is the number of occurrence in L of all the characters preceding σ_c, the c-th symbol in Σ. Therefore, the major problem in searching directly on BWT-compressed text is to compute these arrays from the compressed outputs. Given $m_1 = |M_1| = u$, $m_2 = |M_2|$, Algorithms 4.2.2 and 4.2.3 below compute the C array and the V array respectively, using the output from the modified MTF algorithm.

Algorithm 4.2.2 Compute C array from output of modified MTF

Compute-C-Array(M_1, M_2, Σ)
$j \leftarrow 0$
$CC[i] \leftarrow 0, \forall i$ # initialize cummulative counter
for $i \leftarrow 1, \ldots, u$ **do**
    $j \leftarrow j + M_1[i]$
    $\sigma = M_2[j]$
    $CC[\sigma] = CC[\sigma] + 1$
end for
$C[0] \leftarrow 0$
for $i \leftarrow 1, \ldots, |\Sigma| - 1$ **do**
    $C[i] \leftarrow C[i-1] + CC[i]$
end for

Both algorithms are of complexity $O(u)$. Notice that we do not need to know $L(i)$, for each i here. The normal MTF requires $O(|\Sigma|u)$ time to compute $L(i)$. For ASCII text, $|\Sigma|$ is generally taken as 256 which is quite a large number in practical implementations. Our method has a small constant for the $O(u)$ computation, since we avoid the alphabet adjustment for each symbol.

Besides the advantage of not computing L explicitly, we have a good locality property if we need to access some arbitrary character $L(i)$ for some application. It requires $O(u)$ computation to access $L(i)$ using the traditional MTF. With the modified MTF, we can locate $L(i)$ in $O(\log m_2)$ time. There is an algorithm to obtain a specific element $L(i)$ without computing the whole L array using M_1 and M_2. If M_1 is represented by the number of repetitions of the corresponding symbols in M_2, M_1 in the above example becomes "2,4,4,3,3,3". Using M_1 we can obtain the accumulation count "$2, 6, 10, 13, 16, 19$" for the distinct symbol list in M_2 in $O(m_2)$ time. Then the symbol corresponding to the i-th position in string L can be found by binary search in the accumulation array and the corresponding symbol code in M_2 can be directly obtained. Thus the complexity becomes $O(u \log m_2)$. For the 133 text files in our test courpus (see the section on results), $m_2 \approx u/3$ on average. Thus, when compared with the $O(|\Sigma|u)$ time for traditional MTF, the $O(u \log m_2)$ complexity of the modified MTF represents a significant improvement.

Algorithm 4.2.3 Compute V array from output of modified MTF

Compute-V-Array(M_1, M_2)
$j \leftarrow 0$
$CC[i] \leftarrow 0, \forall i$ # initialize cummulative counter
for $i \leftarrow 1, \ldots, u$ **do**
    $j \leftarrow j + M_1[i]$
    $\sigma \leftarrow M_2[j]$
    $CC[\sigma] = CC[\sigma] + 1$
    $V[i] \leftarrow CC[\sigma] + C[\sigma]$
end for

4.3 Exact Matching on BWT Text

With pre-computed auxiliary arrays available, we can perform fast q-gram generation. Then exact pattern matching can performed by using binary search, based on these q-grams. Below, we describe an algorithm for exact pattern matching on BWT text. Our methods for pattern matching with errors are based on an extension of the exact matching algorithm[AMB02].

4.3.1 Generating q-grams from BWT output

We need the q-grams to produce an initial index to where a potential match can start in T (the original string). However, to generate the q-grams in the text, we will need to access the individual characters in T, without the full BWT inverse transformation.

From the array of last characters, L — the available output of the BWT, we generate the array of first characters, F (usually required for the inverse transform anyway). With only F and L, we can easily generate the bi-grams, and all the other q-grams in T, ($q \leq u$). We describe the general procedure below with an example, using the string used in the previous example. Let the string be $T = abraca$. The first (F) and the last (L) columns of the BWT matrix are

F....L		V	M
a	c	5	aabrac
a	a	1	abraca
a	r	6	acaabr
b	a	2	bracaa
c	a	3	caabra
r	b	4	racaab

The whole matrix M is shown for convenience. The index column V denotes the positions of elements of L in F. Since each row of the matrix is a cyclic rotation, we know that the set of bi-grams in the string are $\mathcal{Q}_2^T = \{ca, aa, ra, ab, ac, br\}$ which is obtained by concatenating corresponding elements F after L. If we now sort $\mathcal{Q}_2^T$ lexicographically, we get the first (F) and the second (S) columns of the matrix

FS....L		V
aa	c	5
ab	a	1
ac	r	6
br	a	2
ca	a	3
ra	b	4

But we do not need to sort lexicographically again. We can just read the bi-grams corresponding to the index set $V = \{5, 1, 6, 2, 3, 4\}$ from the set $\mathcal{Q}_2^T$, that is, the second element in $\mathcal{Q}_2^T$ occupies the 1^{st} position in FS column, the fourth occupies the 2^{nd} position,

$\ldots$, the third occupies the 6^{th} position. Now to find all the 3-gram context, all we need to do is follow the V column and list LFS giving

```
FST....L   V

aab    c   5
abr    a   1
aca    r   6
bra    a   2
caa    a   3
rac    b   4
```

The third column (T) of the sorted matrix is produced as a by-product. We can proceed similarly to generate 4-grams, 5-grams, $\ldots u$-grams, and use these to do pattern matching.

The complexity of the algorithm is determined by only a one time sorting initially $O(u \log u)$, and then all steps take $O(u)$ operations. If we are looking for a pattern of length m, we could do a binary search on the sorted m-grams taking only $O(m \log u)$ time but needs $O(mu)$ storage.

A simple procedure that generates sorted q-grams using the description above is given below. We denote the sorted m-grams as $F(m\text{-}gram)$ which is a vector of length $u = |T|$ of m-tuples of characters. Obviously, $F = F(1\text{-}gram)$ and the lexicographically sorted matrix of all cyclic rotations of T is $F(u\text{-}gram)$. We assume $m \leq u$. The symbol '*' denotes concatenation of character strings.

If $q = 2$, it will be the sorted bi-grams. The procedure is $O(u)$ if q is small, but $O(u^2)$ in the worst case.

Algorithm 4.3.1 Generate qgrams from BWT arrays

```
Qgrams(F, L, V, q)
u ← |F|;
F(1 − gram) ← F;
for m ← 2 to q do
    for i ← 1 to u do
        F(m − gram)[V(i)] ← L[i] ∗ F((m − 1) − gram)[i];
    end for
end for
```

4.3.1.1 Matching with q-grams

To find if the pattern occurred in the text, we perform the q-gram intersection using the q-grams. The intersection of the q-grams in $\mathcal{Q}_q^T$ and $\mathcal{Q}_q^P$ involves matching all the q-grams in one set with all the q-grams in the second set. The time required will depend on the size of the sets. In the worst case, we could be matching the u^2 q-grams in $\mathcal{Q}_q^T$ with the m^2 q-grams in $\mathcal{Q}_q^P$, leading to a potential $O(q(mu)^2)$ cost. A simple way to reduce this is to match the x-length q-grams in one set against only the x-length q-grams in the other set, rather than matching against all q-grams.

By using the special nature of the BWT, we can reduce the time and space requirements for the above procedures. In subsequent sections, we show that we can indeed generate and match the q-grams that we need much faster than what is described above, and using a smaller space.

4.3.2 Fast q-gram generation

4.3.2.1 Permissible q-grams

Given q, the function **qgrams**(F, L, T, q) produces **all** x-grams, where $x = 1, 2, \ldots q$, (an $O(n^2)$ number of x-grams for small q). However, for a given pattern, we do not need every one of the $O(n^2)$ possible q-grams that is generated. Further, since F is sorted, and we shall be doing a lot of searching on the arrays, it might be better to use F rather than the permuted sequence in L.

Recall that Hr provides a one-to-one mapping between F and T. The key to a faster approach is to generate *only* the q-grams that are necessary, using the F and Hr arrays. We call these q-grams that are necessary the ***permissible q-grams*** — they are the only q-grams that are permissible given u, m, and the fact that matching can not progress beyond the last characters in T and P. That is, we do not need to match against the possible rotations in the strings.

This means that the q-grams involving rotations with the substring $t_u t_1$ (the last and first characters in T), are not needed. Thus, the bi-gram aa and the 3-gram aab produced in the previous example, are not permissible.

Further, if we wish to perform exact-pattern matching for a pattern P, where $|P| = m$, all we need will be the m-length q-grams (i.e. the m-grams) in the text T. The m-length q-grams (and excluding the q-gram from the rotations of the text) are the *permissible q-*

grams. In general, the number of permissible q-grams should decrease with increasing q. From the foregoing, we have a total of $u - q + 1$ permissible q-grams for a u-length text.

Given that $m = |P|$, we can generate all the permissible m-grams in the text T. We already have the *single* permissible m-gram from P — the pattern itself. Thus, for exact pattern matching, the required search becomes equivalent to computing the m-gram intersection using the permissible m-grams in $\mathcal{Q}_m^T$ and $\mathcal{Q}_m^P$. The major problems then are to find cheap ways to generate the q-grams from T, and then how to perform the intersection quickly.

4.3.2.2 Fast q-gram generation

We can improve the time for generating the q-grams by making use of the auxiliary arrays. With F and Hr, we can obtain any area of T as needed, by simple array indirection. Then, there is a simple algorithm to generate the q-grams, for any given q:

$\forall_{x=1,2,\ldots u-q+1}$, q- **gram**$[x] = T[x] \ldots T[x+q-1]$; Or equivalently,

$\forall_{x=1,2,\ldots u-q+1}$, $\mathcal{Q}_q^T[x] = F[Hr[x]] \ldots F[Hr[x+q-1]]$;

This will take $u - q + 1$ time units, and will produce $(u - q + 1)$ q-grams. These q-grams are not sorted. We can sort them in $q(u - q + 1)\log(u - q + 1) \approx q(u - q)\log(u - q)$ time units. For most values of q, this will take about $u \log u$ time units, with a worst case of $\frac{u^2}{4}\log(\frac{u}{2})$, at $q = \frac{u-1}{2}$.

However, we do not need to sort the q-grams separately. We can obtain the sorted q-grams directly by picking out the x's according to their *order* in Hr, and then use F to locate them in T. Thus, with the auxiliary array Hrs, the sorted index for Hr, we need to consider only the first $u - q + 1$ entries. (See the procedure for generating the auxiliary arrays in Section 4.2.3.1).

From the foregoing, we have the following lemma:

Lemma 1:

Given a text string $T = t_1 t_2 \ldots t_u$ that has been transformed with the BWT, the array of first characters, F, and the auxiliary arrays Hr and Hrs, then for any $q, 1 \leq q \leq u$, the set of permissible q-grams, $\mathcal{Q}_q^T$, can be generated in O(u-q) time, and can be stored using $O(u - q)q$ space. $\diamondsuit$

The following lemma shows that we do not really need to generate the q-grams explicitly. Neither do we need any extra storage for them. We can access them directly in T via F, Hr and Hrs any time we need to use them.

Lemma 2:

Given a text string $T = t_1 t_2, \ldots t_u$ transformed with the BWT, the array of first characters, F, and the auxiliary arrays Hr and Hrs, for any $q, 1 \leq q \leq u$, $\mathcal{Q}_q^T$, the set of **sorted** permissible q-grams can be obtained in constant time and constant space.

Proof.

The availability of F and Hr implies constant-time access to any area in the text string T. We notice that the x used in the previous description is simply an index on the elements of T. Thus, for any given x, $(0 \leq x \leq u-q)$, we already know the starting and ending points of $\mathcal{Q}_q^T[x]$, the x-th q-gram in T. Therefore, with the index x, we can obtain the corresponding q-gram in T by simply using array indexing on F and Hr. This will be an O(1) operation. If we choose the indices x based on the values in Hrs, the result will be a sorted set of permissible q-grams. Since $F[i] = T[Hrs[i]] = F[Hr[Hrs[i]]]$.

The following algorithm produces the sorted q-grams:

$\forall_{x=1,2,\ldots u-q+1}$,

$v = Hrs[x]$;

$\mathcal{Q}_q^T[x] = F[Hr[v]], F[Hr[v+1]], \ldots, F[Hr[v+q-1]]$;

Thus, with x, q and just one array lookup, $(v = Hrs[x])$, we already know the location of the x-th sorted q-gram in T, via the Hr and F arrays. We neither need to compute, nor to sort the q-grams separately. All we need is to perform array indirection at the time we need the q-grams.

Conversely, in principle, we won't need any extra space to store the q-grams. We can always pick them up on the fly from T (via F and Hr), whenever we need them. In terms of implementation, to compare two q-grams, it might be easier to use $2q$ space to hold the two q-grams as they are being compared, but this is not necessary. $\diamondsuit$

4.3.3 Fast q-gram intersection

First, we consider the general q-gram intersection problem. (The general problem will be important in our approach to inexact pattern matching on BWT-compressed text). We modify the general algorithm for the specific case with fixed $q = m$, which corresponds to the case of exact pattern matching. Below, we present different fast q-gram intersection algorithms as a series of refinements on a basic algorithm. The refinements are based on the nature of the different arrays used in the BWT process, and the new transformation vectors previously described. In some cases, the improvement in running time is obtained at the cost of some extra space.

4.3.3.1 Naive algorithm

Let $\mathcal{MQ}_q = \mathcal{Q}_q^P \cap \mathcal{Q}_q^T$. We call $\mathcal{MQ}_q$, the *set of matching q-grams.* For each q-gram, we use indexing on F and Hr to pick up the required areas in T, and then match the patterns.

To compute $\mathcal{MQ}_q$, we need to search for the occurrence of each member of $\mathcal{Q}_q^P$ in $\mathcal{Q}_q^T$. This will require a running time proportional to $q(u-q+1)(m-q+1))$. This will be $O(mu)$ on average, with a possible worst case in $O(u^3)$.

We can improve the search time by using the fact that F is already sorted. Hence, we can use binary search on F to determine the location (if any) of each q-gram from $\mathcal{Q}_q^P$. This will reduce the time to search for each q-gram to $q \log(u - q + 1)$ time units, giving an

$O(q(m-q)\log(u-q))$ time for the intersection. Average time will be in $O(m \log u)$, while the worst case will be in $O(\frac{u^2}{4}\log\frac{u}{2}) = O(u^2 \log u)$.

4.3.3.2

Improved algorithm

With the sorted characters in F, we can view the F array as being divided into $|\Sigma|$ disjoint partitions, some of which may be empty:

$$F = \bigcup_{i=1,2,\ldots,|\Sigma|} \mathcal{PF}_i,$$

where the $\cup$ operation maintains the ordering in F. The size of $\mathcal{PF}_i$, the i-th partition, is simply the number of occurrences of the i-th symbol in the text string, T. This number can be pre-computed by using C, the count array used in constructing V from L (see Section 4.2.2.2). Let $\mathcal{Z}_i^{\mathcal{F}} = |\mathcal{PF}_i|$. Then,

$$\mathcal{Z}_i^{\mathcal{F}} = C[i+1] - C[i], \forall_{i=1,2,\ldots|\Sigma|-1}, \text{ and } \mathcal{Z}_i^{\mathcal{F}} = u - C[i] \quad \textbf{if } i = |\Sigma|.$$

Similarly, since for a pattern P, the members of $\mathcal{Q}_q^P$, are sorted, these q-grams also form an equivalent $|\Sigma|$ disjoint partitions, some of which may be empty. Let $\mathcal{PQ}_i^P$ be the i-th partition. $\mathcal{PQ}_i^P$, contains the q-grams in $\mathcal{Q}_q^P$ that start with the i-th symbol. Let $\mathcal{Z}_i^{\mathcal{P}} = |\mathcal{PQ}_i^P|$. Let $\mathcal{Z}_{d_T}$ be the number of q-grams in $\mathcal{Q}_q^T$ that started with *distinct* characters — simply, the number of non-empty partitions in $\mathcal{Q}_q^T$. Also, let $\mathcal{Z}_{d_P}$ be the number of q-

grams in $\mathcal{Q}_q^P$ that started with *distinct* characters. Thus, $\mathcal{Z}_{d_T} \leq u - q + 1$ and $\mathcal{Z}_{d_T} \leq |\Sigma|$. Similarly, $\mathcal{Z}_{d_P} \leq m - q + 1$ and $\mathcal{Z}_{d_P} \leq |\Sigma|$.

We can reduce the search time by noting that a q-gram in one partition in $\mathcal{Q}_q^P$, say $(\mathcal{P}\mathcal{Q}_i^P)$, can only match a q-gram in the corresponding partition in F, (i.e. a q-gram in $\mathcal{P}\mathcal{F}_i$). Thus, we can limit the search to within only the relevant partition in F. Also, we only need to do the search for the first character just once for each distict symbol in $\mathcal{Q}_q^P$. Thus, to do the matching, we first perform binary search on F, using just the *first character* in each q-gram partition in $\mathcal{Q}_q^P$. This will locate the starting position of the corresponding partition in F. Then, we search for the given q-gram from P, starting from the first untill the last q-gram in the corresponding partition in F. The running time will be in: $O(\mathcal{Z}_{d_P} \log(u) + q \sum_{i \in \Sigma} \mathcal{Z}_i^{\mathcal{F}} \mathcal{Z}_i^{\mathcal{P}})$, where $\mathcal{Z}_{d_P} \leq m - q + 1$, $\sum_{i \in \Sigma} \mathcal{Z}_i^{\mathcal{P}} = m - q + 1$, and$\sum_{i \in \Sigma} \mathcal{Z}_i^{\mathcal{F}} = u$.

We can make a simple modification to the above, based on the fact that we already know the size of each partition in F. That is, instead of doing a sequential search until the end of the current partition in F, we do a binary search.

For a q-gram in the i-th partition, the bounds of the binary search are determined by $\mathcal{Z}_i^{\mathcal{F}}$, the size of the current partition in F. This can be obtained using the starting and end pointers of each partition in F, which in turn can be determined using C. With this, the time for q-gram intersection can be reduced to: $O(\mathcal{Z}_{d_P} \log(u) + q \sum_{i \in \Sigma} \mathcal{Z}_i^{\mathcal{P}} \log \mathcal{Z}_i^{\mathcal{F}})$.

4.3.3.3

The QGRAM algorithm

A further modification is based on the observation that we can obtain not only $\mathcal{Z}_i^{\mathcal{F}}$, the size of the i-th partition in F, but also the starting position of each distinct character in F using the count array, C. Since both C and F are sorted in the same order, we can determine the start positions (sp_c) and end position (ep_c) of each character partition in F by using C. We could compute sp_c and ep_c as needed, or we can pre- compute them and store in $2|\Sigma|$ space.

Then, we do the initial search for sp_c, the starting points for each partition by using C rather than F. That is, binary search on C to determine the position in F of the first character for the members of each q-gram partition in $\mathcal{Q}_q^P$. Then, binary search on T (via F and Hr), using the start and end pointers for each partition — obtained from C. This reduces the first term in the complexity figure for the previous algorithm, leading to a running time of $O(\mathcal{Z}_{d_P} \log |\Sigma| + q \sum_{i \in \Sigma} \mathcal{Z}_i^{\mathcal{P}} \log \mathcal{Z}_i^{\mathcal{F}})$.

We sumarize the foregoing with the following lemma:

Lemma 3:

Given a text string $T = t_1 t_2 \ldots t_u$, transformed with the BWT, a pattern $P = p_1 p_2 \ldots p_m$, a symbol alphabet with equi-probable symbols $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$ and the arrays F, Hr, Hrs, C.

The **QGRAM** algorithm performs q-gram intersection in $O(|\Sigma| \log |\Sigma| + q(m-q) \log \frac{u}{|\Sigma|})$ time on average, and $O(|\Sigma| \log |\Sigma| + m^2 \log \frac{u}{|\Sigma|})$ worst case.

Proof.

With equi-probable symbols, we have $\mathcal{Z}_{d_P} = |\Sigma|$; $\forall i, \mathcal{Z}_i^{\mathcal{P}} = \frac{m-q+1}{|\Sigma|}$; and $\mathcal{Z}_i^{\mathcal{F}} = \frac{u}{|\Sigma|}$. Then, the average case claim follows easily. The worst case occurs at $q = \frac{m-1}{2}$, when we have $\frac{m^2}{4} \log \frac{u}{|\Sigma|} = O(m^2 \log \frac{u}{|\Sigma|})$ for the second component of the cost. $\diamondsuit$

Comapared to the previous algorithm, the improvement in speed produced by the **QGRAM** algorithm can be quite significant, since typically $|\Sigma| \ll u$. This is especially the case for small alphabets, such as DNA sequences, or binary strings with **1's** and **0's**. We relate the above to the specific case of exact pattern matching (i.e. $q = m$) to obtain the following theorem:

Theorem 1:

Given a text string $T = t_1 t_2 \dots t_u$, a pattern $P = p_1 p_2, \dots, p_m$, and a symbol alphabet with equi-probable symbols $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_{|\Sigma|}\}$, let T be transformed by the BWT to produce an output Z. There is an algorithm that can locate **all** the η_{occ} occurrences of P in T, using only the BWT output Z (i.e. without full decompression) in $O(m \log \frac{u}{|\Sigma|}) + \eta_{occ}$ time on average, and in $O(\log |\Sigma| + m \log u) + \eta_{occ}$ worst case, after an $O(u)$ preprocessing on Z.

Proof

For exact pattern matching, $q = m$ and $\mathcal{Z}_{d_P} = 1$. Further, we will need to consider **just one** partition in F, since we have a single non-empty partition in $\mathcal{Q}_q^P$. That is, the partition in F that starts with symbol $P[1]$. Apart from this single non-empty partition in $\mathcal{Q}_q^P$ (whose size is 1), for all other partitions, say $\mathcal{P}\mathcal{Q}_i^P$, we have $\mathcal{Z}_i^{\mathcal{P}} = |\mathcal{P}\mathcal{Q}_i^P| = 0$. Using these in the QGRAM algorithm, we obtain a running time of $O(\log|\Sigma| + m\sum_{i\in\Sigma}\log\mathcal{Z}_i^{\mathcal{F}})$.

Since we need to consider only one partition in F, the summation term will be evaluated over just a single partition. Call it the σ-th partition. Then the cost will be in $O(\log|\Sigma| + m\log\mathcal{Z}_\sigma^{\mathcal{F}})$. The worst case will be when T and P are repetitions of the same symbol, example $T = a^u$, $P = a^m$. Here, we have only one partition of size u in F, and the characters will always be matching those in P. Since we will need only a single one-character comparison to determine the partition $P[1]$ in F, the $\log|\Sigma|$ cost will disappear. Thus we have a running time of $O(\log|\Sigma| + m\log u)$. With an equi-probable symbol alphabet, we will have: $\forall i, \mathcal{Z}_i^{\mathcal{F}} = \frac{u}{|\Sigma|}$. Since we need to consider only the σ-th partition in F, we will have an average running time of $O(\log|\Sigma| + m\log\frac{u}{|\Sigma|}) \approx O(m\log\frac{u}{|\Sigma|})$. Since all the matching patterns will be lined up in the same area of the text, we need an $O(\eta_{occ})$ number of array look-ups, via F and Hr arrays to report the actual positions of the occurrences in T. The $O(u)$ time in the theorem is needed to compute the V, F, Hr and Hrs arrays from the BWT output, Z. $\diamondsuit$

Below, we present a further refinement of the QGRAM algorithm to provide an improved time complexity.

4.3.4 The QGREP algorithm

We can improve the running time above by observing that during q-gram intersection with a q-length substring of the pattern, a mismatch for any p- gram prefix of the substring ($p \leq q$) implies that completing the symbol comparison for the q-gram cannot lead to a successful match. Thus, we can terminate the match more intelligently by stopping the comparison whenever a mismatch has occurred. This means that, during the binary search used in the q-gram intersection, we do not really need to perform all the q symbol-by- symbol comparisons, as was done by the QGRAM algorithm.

A further improvement can be obtained by observing that during the binary search step used by the algorithms, when a mismatch occurs, we can record information about the positions of the mismatch. Since the q-grams are sorted, at the next step of the binary match, we do not need to start matching from the beginning of the q-grams again. We now start from the previously recorded mismatch position. At a given iteration of the binary search, let u_p be the position of a mismatch for the upper boundary in the binary search, and let l_p be the corresponding position of a mismatch for the lower boundary. Let $c_p = \min\{l_p, u_p\}$. Let S be the sorted suffix at the next jump in the binary search. Then, for the next match iteration (i.e. next jump in the binary search), instead of the usual case of starting the match at position 1, (i.e. match $S[1, 2, \ldots]$ versus $P[1, 2, \ldots]$, etc.), we now start at position c_p, (i.e. match $S[c_p, c_p + 1, \ldots, c_p + m]$ versus $P[c_p, c_p + 1, ..., m]$). For exact pattern matching, after the $O(u)$ construction for the auxiliary arrays, this will lead to an average case of

$O(m + \log \frac{u}{|\Sigma|})$ time, to find the first occurrence of the pattern, or to determine that the pattern does not occur in the text.

When we find the first match, we can determine the total number of occurrences in at most $O(\log \frac{u}{|\Sigma|})$ time steps. This means an $O(m + \log \frac{u}{|\Sigma|})$ time to **count** all the η_{occ} occurrences, independent of η_{occ}. This independence on the number of occurrences will be important when the ratio $\frac{u}{m}$ is large. This is typical of certain applications in biological sequence analysis, where the size of the text string T could be in billions, while P could be a short pattern, with less than hundred characters. Moreover, for some applications, such as Internet search engines, only a count of the number of occurrences η_{occ} (rather than the actual positions of the match) is required. However, we will still require an $O(\eta_{occ})$ number of array look-ups, via F and Hr arrays to report the actual positions of the occurrence in T.

The worst case search time for the above is still in $O(m \log \frac{u}{|\Sigma|})$ time. This occurs when we have a text sequences $T = a^u$, (i.e. $T = aaaaa...$, u times), and the pattern $P = a^m$. It is, however, possible to reduce the worst case time to $O(m + \log u)$ by storing further information about the longest common prefixes ($l$cp) between the suffixes. For our purposes, only two arrays with the lcp values for $u - 1$ specially selected pair of sorted suffixes are needed. The two lcp arrays can be constructed in $O(u)$ time, and can be stored in $2u$ additional space. An example of such a modification can be found in [MM93]. With our auxiliary arrays from the BWT, it is a simple matter to construct the two lcp arrays using the BWT output.

We summarize the above results in the following theorem:

Theorem 2: Exact pattern matching on BWT-compressed text.

Given a text string $T = t_1 t_2 \ldots t_u$***, a pattern*** $P = p_1 p_2, \ldots, p_m$***, and a symbol alphabet*** $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$***. Let*** T ***be compressed by the BWT to produce an output*** Z***. There is an algorithm to locate all the*** η_{occ} ***occurrences of*** P ***in*** T***, using only the BWT output*** Z (***i.e. without full decompression***) ***in*** $O(m + \log\frac{u}{|\Sigma|} + \eta_{occ})$ ***time on average, and in*** $O(m + \log u + \eta_{occ})$ ***worst case, after an*** $O(u)$ ***preprocessing on*** Z***.***

4.3.5 Space considerations

In general, BWT decompression requires space for L, R, C and V, or $(3u + |\Sigma|)$ units of storage. But after obtaining V from R and C, we do not need R anymore. In our descriptions, we use both Hr and Hrs arrays. With F we don't need L, and after generating Hr and Hr, we can release the space used by T. Thus, the described algorithms require only $(4u+|\Sigma|+m)$ units of storage space - an extra space of $u + m$

If the starting and end points for the partitions are precomputed, then we will need $2\mathcal{Z}_{d_T}$ units of space to store them, requiring $2|\Sigma|$ units in the worst case — when every symbol in the alphabet appeared in T.

As described above, the extra space needed for the two *lcp* arrays used by the QGREP algorithm is $2u$. So, the worst case extra space for each of the algorithms described will be in $O(u + m + |\Sigma|)$.

4.4 Experimental Results for Exact Pattern Matching

4.4.1 Experimental Setup

To evaluate the performance of the BWT-based approach to text pattern matching, we performed two experiments on a database of text sequences. The database is made up of a total of 133 text files selected from three different text corpora (text files from the Canterbury Corpus, *html* and *C* program files from the Calgary Corpus, and the AP, DOE and FR files from Disk 1 of the TIPSTER corpus). The file sizes ranged from 11,150 characters (`fields.c`) to 4,161,115 characters (FR89011 in TIPSTER). The average file size was 935,719 characters.

The first experiment was performed using 10 sets of sample patterns (words). Each set has 100 words with the same length, $m = |P| = 2, 3, \ldots 11$. The words were chosen from the most frequently used words in the English dictionary. The second test was based on a set of 14 patterns, some of which do not necessarily occur in the text. The tests were carried on a Sun UltraSparc-IIi workstation (360 MHz, 256MB RAM) running solaris 2.5 operating system. The programs were written in C.

For ease of implementation, we used a simple BWT compression program (BSMP reported in [BPM02]). This is decidedly slower than standard BWT compression schemes, such as BZIP2, but produces a comparable compression. Under this implementation, the overhead for computing the auxiliary arrays was 0.94sec/MB on average. The average decompression time was 1.28sec/MB. To ensure that the reported times were not affected by system load, or a one-time event, we ran each test 500 times, and recorded the average time used. The reported search times are in seconds.

4.4.2 Number of occurrences

The number of occurrences (η_{occ}) has a direct effect on the overall search time. We use AGREP, the standard pattern matching algorithm [WM92b], and its recent variant NR-GREP [Nav01] as the standard algorithms for comparative performance with the proposed methods. AGREP and NRGREP report only one occurrence for each line in the input text. That is, when the text has no end of line markers, they will report only one occurrence (if the pattern occurs), independent of the actual number of occurrences. For a fair comparison with the proposed algorithms, we introduced a new line after each distinct word in the text. Thus, each pattern is now wholly contained in one line of text. For a given pattern length m, we recorded the number of occurrence of each test pattern in the entire database of text. Tables 4.1 shows the trend that the longer the pattern, the less the occurrences of the patterns with the exception of the patterns of length nine.

Table 4.1: Number of occurrences and number of comparisons for BWT-based pattern matching

Pattern length	Number of Occurrence	Number of Comparisons		
		QGREP	BWT-BINARY	QGRAM
2	52,535,860	27	28	48
3	5,908,036	34	41	81
4	2,184,440	44	55	106
5	783,420	46	57	118
6	201,855	33	46	109
7	101,860	31	45	112
8	8,061	26	42	111
9	48,874	32	50	125
10	5,704	28	46	122
11	8,449	31	47	123

4.4.3 Number of comparisons

Table 4.1 also shows the number of comparisons for the BWT-based pattern matching algorithms. The number of comparison provides an independent measure of performance for pattern matching algorithms. The table does not include results for AGREP and NRGREP since these are based on hardware shift operations, and not on direct symbol comparison.

As expected, the final algorithm (QGREP) requires less number of comparisons than BWT-BINARY, which in turn requires less number of comparisons than QGRAM. On average, the QGREP algorithm uses about 30% less number of comparisons than the binary search algorithm, with the difference increasing with increasing pattern length.

4.4.4 Search Time

To compare the proposed methods with existing pattern matching algorithms, we use the search time. Here, the search time is the time needed to perform a search for a given pattern (including time for array lookup for BWT based methods). It does not include the time needed for one-off operations, such as for construction of the auxiliary arrays (for BWT-based methods), or for decompression and later re-compression when using the non-BWT based methods (AGREP and NRGREP).

Table 4.2 shows comparative results for the search time for AGREP and NRGREP, and for the proposed methods, for varying pattern length, m. For all values of m in the range tested, the proposed algorithms (QGREP and BWT-binary and QGRAM) required less search time than the standard algorithms, AGREP and NRGREP. At $m = 2$, the QGRAM algorithm reported the least search time. In general, however, the time for QGAM was higher than that needed by BWT-binary or QGREP, but significantly lower than the time required by AGREP or NRGREP.

4.4.5 Search time for non-occurrence.

For a better understanding of the behavior of the pattern matching algorithms, we tested the algorithms using known patterns, rather than the set of frequently occurring words used above. The test set included both patterns that are known to occur in the text and those

Table 4.2: Number of occurrences and number of comparisons for BWT-based pattern matching

Pattern length	Number of Occurrence	Search Time (seconds)				
				BWT		
		AGREP	NRGREP	QGREP	BINARY	QGRAM
2	52,535,860	2.020	3.022	0.0690	0.0682	0.0415
3	5,908,036	1.490	1.708	0.0138	0.0129	0.0182
4	2,184,440	1.259	1.430	0.0108	0.0097	0.0179
5	783,420	1.163	1.308	0.0083	0.0084	0.0177
6	201,855	1.031	1.155	0.0063	0.0059	0.0158
7	101,860	0.991	1.078	0.0056	0.0056	0.0161
8	8,061	0.911	1.032	0.0050	0.0052	0.0155
9	48,874	0.934	1.009	0.0056	0.0058	0.0170
10	5,704	0.862	0.949	0.0050	0.0054	0.0164
11	8,449	0.860	0.926	0.0050	0.0053	0.0164
	Average : all	1.152	1.362	0.0134	0.0132	0.0192
	Average : m > 2	1.056	1.177	0.0073	0.0071	0.0168

that do not occur. Table 4.3 shows the results, including η_{occ} and the number of comparisons η_{cmp}. In each case, the best result is produced with one of the proposed BWT-based methods (QGREP, BWT-binary or QGRAM). For cases where $m > 2$, (QGREP and BWT-binary are faster than AGREP and NRGREP by more than two orders of magnitude. The performance improvement increases with decreasing number of occurrence or increasing pattern length.

Between QGREP and BWT-binary, the result was mixed. The QGREP algorithm always required less number of comparisons. However, at times, BWT-binary reported less search time. In the last test (with known patterns), the QGREP was best (in terms of search time) on 5 occasions, the BWT-binary was best on 4 occasions, with 4 ties. This could mean that the search time does not depend only on the number of comparisons, but on other factors, such as the size of the auxiliary arrays. All the same, QGREP provided an overall best performance, in terms of both search time and number of comparisons. Again, the performance difference increased with decreasing η_{occ}, or increasing m.

Table 4.3: COMPARATIVE SEARCH TIME (CONTROLLED SET OF PATTERNS, WITH POSSIBLY NO MATCHES) P17: patternmatchingin, P30: bwtcompressedtexttobeornottobe, P22: thisishishatitishishat, P26: universityofcentralflorida, P44: instituteofelectricalandelectromicsengineers

			Search Time (seconds)					Number of comparisons		
	m	η_{cmp}	AGREP	NRGREP	QGREP	BWT BINARY	QGRAM	QGREP	BWT BINARY	QGRAM
p	1	2,287,850	2.694	6.632	0.2921	0.2643	0.3824	0	0	0
pa	2	273,497	2.076	2.416	0.0339	0.0343	0.0224	26	26	36
pat	3	20,920	1.556	1.752	0.0068	0.0068	0.0118	36	42	57
patt	4	1,977	1.416	1.304	0.0043	0.0042	0.0117	36	44	68
patte	5	1,796	1.300	1.370	0.0046	0.0044	0.0124	41	49	72
patter	6	1,761	1.190	1.168	0.0050	0.0050	0.0129	45	53	76
pattern	7	1,656	1.014	1.056	0.0053	0.0051	0.0134	50	58	79
patternm	8	0	0.948	1.000	0.0037	0.0042	0.0135	28	49	81
P17	34	0	0.534	0.596	0.0041	0.0045	0.0140	28	49	81
P30	13	0	0.796	0.878	0.0035	0.0036	0.0126	25	39	61
P22	22	0	0.616	0.540	0.0041	0.0041	0.0157	30	44	107
P26	26	0	0.712	0.566	0.0045	0.0045	0.0178	41	53	114
P44	44	0	0.470	0.526	0.0047	0.0061	0.0201	33	77	113
P99	99	0	0.420	0.918	0.0041	0.0043	0.0127	21	33	61
Average: all	20	184961	1.124	1.480	0.0272	0.0254	0.0410	31	44	72
Average: $\eta_{occ} > 0$	4	369922	1.607	2.243	0.0503	0.0463	0.0667	33	39	56
Average: $\eta_{occ} = 0$	36	0	0.642	0.718	0.0041	0.0045	0.0152	30	49	89

One might also observe the unusually large search time when $m = 1$ or 2. This is mainly due to the time required to report all the matches, *after* the matches have been found. This is most evident when $m = 1$, where the proposed methods require no comparison at all. Yet the search time was the highest, due to the large number of occurrences.

4.5 Locating k-mismatches

We present a k-mismatch algorithm based on the fast q-gram generation algorithm, which is an extension of extension of the QGRAM algorithm [AMB02], and of the QGREP algorithm described. The pattern matching operation is performed with all possible alignments of the pattern with the text. This is done in an incremental and indirect manner, via the matrix of sorted suffixes, S, as specified by the vector Hrs. The suffix matrix S is part of the sorted matrix of rotations, M (see Section 4.2.2). The characters of the pattern P are compared with the characters in successive columns of the suffix array matrix S. If there is a mismatch between the characters at corresponding locations of P and T for a given row in S, the number of mismatches is incremented by 1. Since the S matrix is lexicographically sorted, the match or mismatch takes place within the entire group of consecutively located rows in S. We record the number of mismatches (*count*) for the group as well as the start(*st*) and end (*ed*) positions for the group in the form of a triplet $(st, ed, count)$. We place the triplet in an output list called **Candidates**. If *count* is still less than k, we will continue to search in the group. If in a given row, the suffix length becomes less than the pattern length, it means that for this alignment of the pattern, the text has ran out of characters. So, for each additional operation the mismatch count in its row triplet has to be incremented by one as long as $count \leq k$. The operation proceeds until the last character of P is processed yielding a final partition of the suffixes in S having maximum of k mismatches with P. The triplets remaining in **Candidates** at this point will be those that survive with $count \leq k$, and thus correspond to positions with a maximum of k-mismatches to the pattern.

	P[1]=s	P[2]=s	P[3]=i	P[4]=s
	F	FS	FST	FSTF
1	i	i X		
2	i (1,4,1)	ip (2,3,2)	ipp X	
3	i	is (3,4,1)	iss (3,4,2)	issi X
4	i	is	iss	issi
5	m (5,5,1)	mi (5,5,2)	mis X	
6	p (6,7,1)	pi (6,6,2)	pi X	
7	p	pp (7,8,2)	ppi (7,8,2)	ppi X
8	s	si (8,9,1)	sip (8,8,2)	sipp X
9	s (8,11,0)	si	sis (9,9,2)	siss (9,9,2)
10	s	ss (10,11,0)	ssi (10,11,0)	ssip (10,11,1)
11	s	ss	ssi	ssis (10,11,0)

We demonstrate the algorithm with an example: P="$s$sis", T= "$m$ississippi", $k = 2$. When the first character of the pattern 's' is checked, the first column of q-gram is partitioned into four groups. These are represented by the four triplets recording the start and end position in the sorted matrix as well as the number of errors that occurred. For example, row 8 to row 11 has the character 's' matching $P[1]$. This has triplet $(8, 11, 0)$. Row 1 to row 4 has the character 'i' that is a mismatch to $P[1]$, hence the triplet $(1, 4, 1)$. Then continue matching with the next character in the pattern by checking the second character of the bigrams in the text. The first row corresponds to the suffix starting from the last character of the text, thus the bigram beginning at this position is not a valid (or *permissible*) bigram. The match operation, therefore has to be terminated. Terminated matches are indicated with 'X' in the above example. Then suffixes that belong to rows 2 to 4 are split into two groups, one with a mismatch giving the triplet (2,3,2) and the other a match giving triplet (3,4,1). Continuing with the third character of the pattern, the suffix in row two has a mismatch so that the error count is greater than k. The corresponding triplet is removed from the candidate set, denoted by 'X'. This also happens for the suffix in row 5. Finally, suffixes in rows 9,10 and 11 lead to pattern match with the error count 2,1 and 0, respectively.

A formal description of the procedure is given in Algorithm 4.5.1. The algorithm makes use of an array of counts, $C = [c_1, c_2, \ldots c_{|\Sigma|}]$. For a given index, c, $C[c]$ stores the number of occurrences in L of all the characters preceding σ_c, the c-th symbol in Σ, $1 \leq c \leq |\Sigma|$. The array C is typically computed as part of the partial decompression of the BWT-compressed text.

Algorithm 4.5.1 The k-mismatch Algorithm

K-mismatch(P, F, C, Hr, Hrs, k)
Initialize *Candidates*

for each symbol $\sigma_c \in \Sigma$, (the c -th symbol in Σ), and $\sigma_c \in F$ **do**
  create a triplet with
  $st \leftarrow C[c] + 1$
  if $c \leq |\Sigma|$ **then** $ed \leftarrow C[c+1]$
  else
    $ed \leftarrow u$
  end if
  if $\sigma_c \leftarrow P[1]$ **then** $count \leftarrow 0$
  else
    $count \leftarrow 1$
  end if
  if $count \leq k$ **then** append triplet to *Candidates*
  end if
end for

for $j \leftarrow 2$ to m **do**
  for each element in *Candidates* that survive the $(j-1)-$ th iteration **do**
    Remove the triplet $(st, ed, count)$ from *Candidates*
    for each distinct symbol $\sigma_c \in F$ **do**
      locate the start and end position st' and ed' in F between st and ed
      using binary search on the j-th column of the suffix matrix S

      (Note that we do not need to generate the j-gram for each row
      in the S matrix during binary search. Instead, given the row
      index pos of a $(j-1)$-gram in S, the j-th symbol s can
      be accessed in constant time as: $s = F[Hr[Hrs[pos] + j]]$)
      if $\sigma_c = P[j]$ **then**
        add triplet $(st', ed', count)$ to *Candidates*
        (Since it is a match to $P[j]$, there is no change to $count$)
      else
        if $count + 1 \leq k$ **then**
          add triplet $(st', ed', count + 1)$ to *Candidates*
          increment $count$ by 1

        end if
      end if
    end for
  end for
end for
Report the m-length patterns between st and ed for each element of *Candidates* as the k-mismatches. The row positions in F can be converted to the corresponding positions in T using Hrs, as explained earliers

4.5.1 Complexity analysis

The preprocessing cost for preparing the auxiliary arrays is $O(u)$. For each iteration of the innermost loop, binary search is used to locate all the groups with the same j-gram. At most u groups will be generated. Thus each loop takes $O(u \log \frac{u}{|\Sigma|})$ time in the worst case. The maximum number of triplets that can be generated will be in $O(|\Sigma|^k)$. But this cannot be more than u, the text size. In practice, many groups or triplets will be dropped because the error count becomes greater than k. In fact, at each match step $i, i \leq k$, a maximum of u triplets can be generated. After step $i = k$, no new triplet will be generated, while the number of remaining triplets will be decreasing. The worst case time to search the whole text will be $O(uk \log \frac{u}{|\Sigma|})$. Fig. 4.1 shows the variation of the number of triplets with the number of iterations for different values of k. Typically, the number of triplets increased with the iterations, approaching a peak at the k-th iteration, and then dropped off quickly as more characters from P are checked (see Fig. 4.1). Fig. 4.2 shows the empirical behavior of the peak and average number of triples with pattern lengths. We observe that the maximum number of triplets remains relatively constant for different pattern lengths, m. The average number of triplets decreases with increasing, m. Both the peak and the average number of triplets are, however, relatively small compared to the average file size (935,719 characters) over the test corpus.

The above results are summarized in the following theorem:

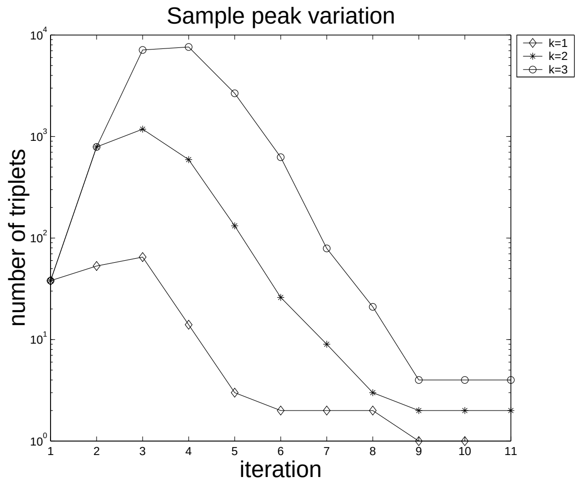
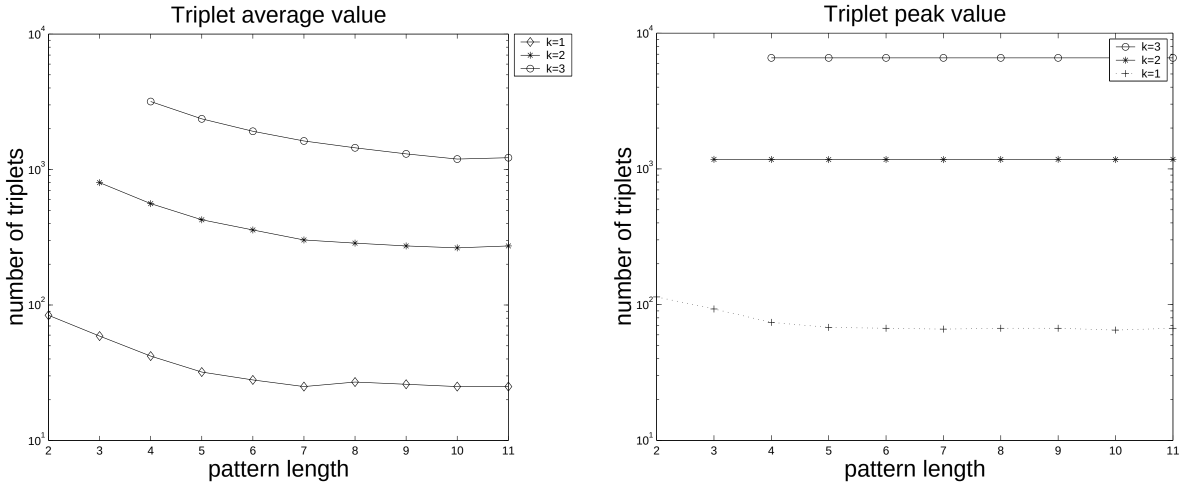


Figure 4.1: A typical variation of triplets with matching step ($m = 11$)

Theorem 2: Given a text string $T = t_1 t_2 \ldots t_u$, a pattern $P = p_1 p_2 \ldots p_m$, and an equiprobable symbol alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$. Let k be given, and let T be compressed by the BWT to produce a compressed output Z. There is an algorithm to locate the k-mismatches of P in T, using only the compressed output Z in $O(uk \log \frac{u}{|\Sigma|})$ time on average, and in $O(uk \log \frac{u}{|\Sigma|})$ worst case, after an $O(u)$ processing on Z. $\diamond$

4.6 Locating k-approximate matches

We perform $k-$approximate matching in two phases. In the first phase, we locate areas in the text that contain potential matches by performing some filtering operations using *appropriately sized* q-grams. In the second phase, we verify the results that are hypothesized by the filtering operations. The verification stage is generally slow, but usually, it will be



(a) (b)

Figure 4.2: Behavior of number of triplets generated during a search for k-mismatch: (a) average number of triplets, (b) peak number of triplets

performed on only a small proportion of the text. Thus, the performance of the algorithm depends critically on the efficiency of the filtering stage — in terms of computational time and also the number of hypothesis generated.

4.6.1 Locating potential matches

The first phase is based on a known fact about approximate pattern matching. That is, for there to be a k-approximate match in some area in a given text, the text **must** contain at least one block of characters such that the characters in the block are in the same order as they appeared in the pattern. We state this more succinctly in the form of a lemma:

Lemma 4: **[BP92]** Given a text T, a pattern P, ($m = |P|$), and an integer k, for a k-approximate match of P to occur in T, there must exist at least one r-length block of symbols in P that form an **exact match** to some r-length substring in T, where r, the minimum block size is given by : $r = \lfloor \frac{m}{k+1} \rfloor$ $\diamondsuit$

This is trivially the case for exact matching, in which $k = 0$, and hence $r = m$. Using the above lemma, the filtering phase can be performed using the following steps:

1. Compute r, the minimum block size for the q-grams.
2. Generate $\mathcal{Q}_r^T$ and $\mathcal{Q}_r^P$, the permissible r-grams from the text T, and the pattern P, respectively
3. Perform q-gram intersection of $\mathcal{Q}_r^T$ and $\mathcal{Q}_r^P$.

Here, we can define the set $\mathcal{MQ}_k$, such that, $t \in \mathcal{MQ}_k \iff T[t, t+1, \ldots, t+r-1] \in \mathcal{Q}_r^P \cap \mathcal{Q}_r^T$.

Let $\eta_h = |\mathcal{MQ}_k|$. Let $\mathcal{MQ}_k^i$ be the i-th matching q-gram in T, $\mathcal{MQ}_k^i[j]$ the j-th character in $\mathcal{MQ}_k^i$, $j = 1, 2, \ldots r$. Further, let $\mathcal{F}_k[i]$ be the index of the first character of $\mathcal{MQ}_k^i$ in the array of first characters, F. That is, $\mathcal{F}_k[i] = x$, if $F[x] = \mathcal{MQ}_k^i[1]$.

We call $\mathcal{MQ}_k$ the *matching q-grams at k.* Its size is an important parameter for the verification phase (and for the general k-approximate matching). It determines the efficiency of the verification stage, which is often more time consuming than the hypothesis generation stage. We will need the first characters in the matching q-grams and their index in F for the verification stage. These indices can be generated in O(1) time. Similarly, step 2 above can be done in constant time and space. The cost of step 3, will grow slower than $\frac{m^2}{k+1} \log u$. The time required for hypothesis generation is simply the time needed for q-gram intersection, where q is given by **Lemma 4**.

Let $\mathcal{Z}_i^{\mathcal{F}}$ be the number of q-grams in the text starting with the i-th symbol in Σ. Then, $\mathcal{Z}_i^{\mathcal{F}} = C[i+1] - C[i], \forall_{i=1,2,\ldots|\Sigma|-1}$, and $\mathcal{Z}_i^{\mathcal{F}} = u - C[i]$ if $i = |\Sigma|$. Similarly, let $\mathcal{Z}_i^{\mathcal{P}}$ be the number of q-grams in the pattern starting with the i-th symbol in Σ. Let $\mathcal{Z}_{d_P}$ be the number of q-grams in $\mathcal{Q}_q^P$ that started with *distinct* characters — simply, the number of non-empty partitions in $\mathcal{Q}_q^P$. Thus, $\mathcal{Z}_{d_P} \leq m - q + 1$ and $\mathcal{Z}_{d_P} \leq |\Sigma|$. Using these in the analysis for the QGRAM algorithm, we have the following:

Lemma 5: Given $T = t_1 t_2 \ldots t_u$, transformed with the BWT, $P = p_1 p_2 \ldots p_m$, an alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_{|\Sigma|}\}$, and the arrays F, Hr and Hrs. Let k be given. The hypothesis phase can be performed in time proportional to: $\mathcal{Z}_{d_P} \log |\Sigma| + \lfloor \frac{m}{k+1} \rfloor \sum_{i \in \Sigma} \mathcal{Z}_i^{\mathcal{P}} \log \mathcal{Z}_i^{\mathcal{F}}$ ♢

4.6.2 Verifying the matches

Here, we need to verify if the r-blocks that were hypothesized in the first phase (i.e. the matching q-grams in $\mathcal{MQ}_k$) are true matches. Among others, the time required to do this will depend critically on η_h, the number of hypothesis generated. We perform the verification in two steps:

1. *Determining the matching neighborhood.* Use Hr and F to determine the left and right limits of the neighborhood in T where each r-block in $\mathcal{MQ}_k$ could be part of a k-approximate match. The maximum size of the neighborhood will be $m + 2k$.

2. *Verify if there is a match within the neighborhood.* Verify if there is a true k-approximate match in the selected area in the text.

Let $\mathcal{N}_i$ be the neighborhood in T for $\mathcal{MQ}_k^i$, the i-th matching q-gram. Let t be the position in T where $\mathcal{MQ}_k^i$ starts. That is, $t = Hr[F[\mathcal{F}_k[i]]]$. The neighborhood is defined by the left and right limits: t_{left} and t_{right} viz: $t_{left} = t - k$; $t_{right} = t + m + k$. To ensure that the neighborhood does not go beyond the begining or end of the text, we use the following definitions:

$t_{left} = t - k$ **if** $t - k \geq 1$; $t_{left} = 1$ otherwise.

$t_{right} = t + m + k$ **if** $t + m + k \leq u$; $t_{right} = u$ otherwise.

The i-th matching neighborhood in T is therefore simply given by: $\mathcal{N}_i = T[t_{left} \ldots t \ldots t_{right}]$. Thus, $|\mathcal{N}_i| \leq m + 2k, \forall_i, i = 1, 2, \ldots \eta_h$. We then obtain a set of matching neighorhoods $\mathcal{S}_{\mathcal{MQ}} = \{\mathcal{N}_1, \mathcal{N}_2, \ldots \mathcal{N}_{\eta_h}\}$. Verifying a match within any given $\mathcal{N}_i$ can be done with any existing fast algorithm for k-approximate matching, for instance, Ukkonen's $O(ku)$ algorithm [Ukk85].

The cost of the fist step will be in $O(\eta_h)$. We only need pointers to the start and end of the $(m+2k)$-sized neighborhoods in T. The value of the pointers can easily be pre-computed for each of the matching neighborhoods in T. The cost of the second step will thus be in $O(\eta_h k(m+2k)) \leq O(\eta_h k(3m)) \approx O(\eta_h km)$.

Example. Let $T = abraca$ and $P = brace$, with $k = 1$. Then, $r = 2$. The permissible q-grams will be $\mathcal{Q}_2^P = \{ac, br, ce, ra\}$, $\mathcal{Q}_2^T = \{ab, ac, br, ca, ra\}$, yielding $\mathcal{MQ}_1 = \{ac, br, ra\}$, and $\mathcal{N}_1 = [3 \ldots 6]; \mathcal{N}_2 = [1, \ldots, 6]; \mathcal{N}_3 = [2, \ldots, 6]$. The correct matches will be found in $\mathcal{N}_1$ and $\mathcal{N}_2$ at positions 1 and 2 in T, respectively. $\diamondsuit$

We observe that for the given T,P and k used in the example, the neighborhoods for the proposed potential matches included a lot of redundant areas — the original text string plus other smaller substrings. Generally, when k and m are small relative to u, (example, $u > m+2k$), the problem will be reduced. However, the problem becomes more pronounced when u and m are comparable, and gets worse with an increasing allowable error (k) in the match. In fact, this redundancy is a major bottleneck for q-gram-based k-approximate matching, especially for large k (see, for example [SS99]). This problem motivates a modification of the verification process, which removes the redundancy by considering the overlaps in the matching neighborhoods.

4.6.3 Faster verification

Instead of doing the verification sequentially as described above, we can first compute the set of ***all*** matching neighborhoods before doing the verification. That is, for each i-th matching q-gram, determine t, the corresponding starting position in T as described above. Thus, we have a set of positions that contain potential matches in T. Using the t's, we check for possible overlaps in the set of matching neighborhood, and remove the overlaps if any.

Let $\psi_{\mathcal{MQ}}$ be the set of matching neighborhoods without overlaps. Let $\mathcal{N}_i^p$ be the corresponding position indices in T of characters in $\mathcal{N}_i$. We can write:

$$\psi_{\mathcal{MQ}} = \bigcup_{\forall i, i \leq \eta_h} \mathcal{N}_i^p$$

Conceptually, while $\mathcal{S}_{\mathcal{MQ}}$ represents a multset of positions (with possible repetition of position indices), $\psi_{\mathcal{MQ}}$ contains disjoint sets of *ordered* positions, with no repetition. If we let $\psi_{\mathcal{MQ}}^j$ be the j-th set in $\psi_{\mathcal{MQ}}$, while $\psi_{\mathcal{MQ}}[i]$ is the i-th individual member in $\psi_{\mathcal{MQ}}$, (i.e. when we consider $\psi_{\mathcal{MQ}}$ as an ordinary set), then a set boundary is defined whenever $\psi_{\mathcal{MQ}}[i+1] - \psi_{\mathcal{MQ}}[i] > 1, (1 \leq i < u)$. We then only need to verify if there are true matches within the smaller set of positions in each $\psi_{\mathcal{MQ}}^j$.

Essentially, what we have done is to use the left and right limits to merge potential overlaps in the neighborhoods before starting the verification. Using the previous example, we will have $\psi_{\mathcal{MQ}}^1 = \psi_{\mathcal{MQ}} = [1, \ldots 6]$, and $T[\psi_{\mathcal{MQ}}] = abraca = T$, the original string (without the other substrings).

The merging of neighborhoods can be done in $O(|\psi_{\mathcal{MQ}}|)$ time. Generally, $|\psi_{\mathcal{MQ}}| \leq \eta_h(m+2k)$. Hence the time required for verification will be reduced from $O(\eta_h k(m+2k))$ to some $O(k|\psi_{\mathcal{MQ}}|)$. The improvement achievable from the merging procedure depends on the input text, and the relative values of u, m, k, and the filtering efficiency, η_h. (Ironically, with less filtering efficiency (i.e. bigger η_h, matching with merged neighborhoods becomes more attractive). Fig. 4.3 and Fig. 4.4 show the variation of η_h and α with the pattern length, m, for various values of k. Both parameters decreased with increasing m, with η_h peaking at $m = 2k + 1$. In particular, we observe how the merging process has significantly reduced the size of the neighborhoods required for verification.

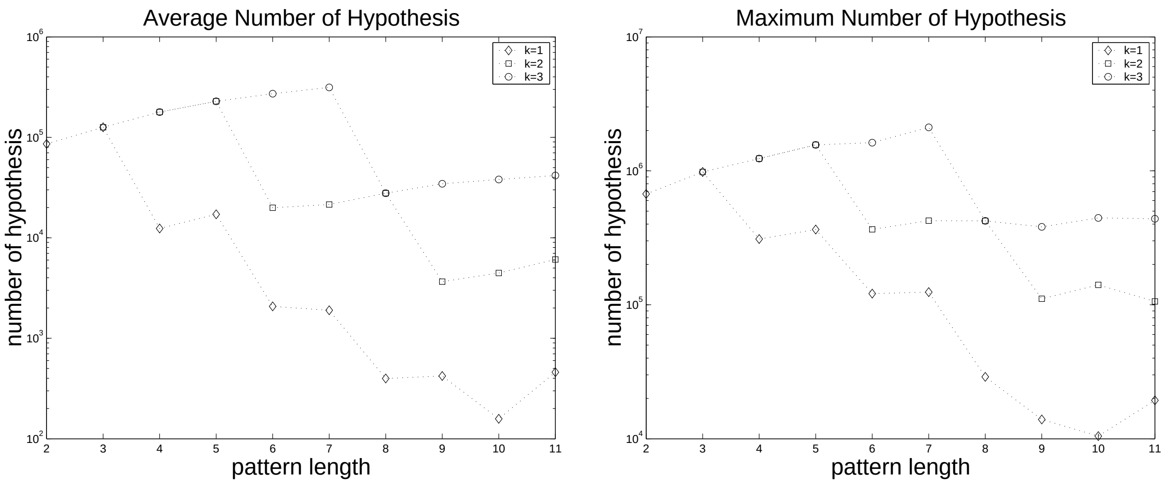


Figure 4.3: Variation of number of hypothesis η_h, with pattern length.

If we copy out the neighborhoods before matching, this could require an extra space in $O(|\psi_{\mathcal{MQ}}|)$, compared with the $O(m + 2k)$ needed with the sequential approach. However, we could use the indices on T (via F and Hr) and thus, will not need any extra space.

With the foregoing description, the combined cost of generating the transformation vectors, hypothesis generation, and hypothesis verification is captured by the following theorem:

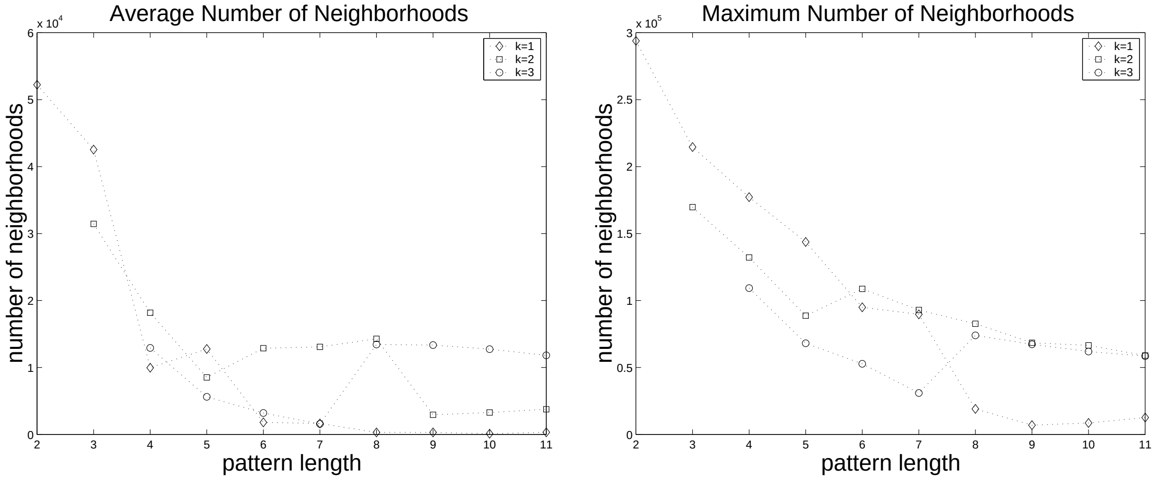


Figure 4.4: Variation of number q-grams in the merged neighborhoods, with pattern length.

Theorem 3: k-approximate matching on BWT-compressed text. *Given a text string $T = t_1 t_2 \dots t_u$, a pattern $P = p_1 p_2 \dots p_m$, and an equi-probable symbol alphabet $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_{|\Sigma|}\}$. Let T be compressed by the BWT to produce a compressed output Z. There is an algorithm to locate the k-approximate matches of P in T, using only the compressed output Z in $O(|\Sigma| \log |\Sigma| + \frac{m^2}{k} \log \frac{u}{|\Sigma|} + \alpha k)$ time on average, ($\alpha = |\psi_{\mathcal{MQ}}| \leq u$), and in $O(|\Sigma| \log |\Sigma| + \frac{m^2}{k} \log \frac{u}{|\Sigma|} + ku)$ worst case, after an $O(u)$ processing on Z.*

Proof. Since all symbols are equi-probable, each symbol should appear as a starting character of some q-gram (assuming $m > |\Sigma|$). Thus $\mathcal{Z}_{d_P} \leq |\Sigma|$. Similarly, $\mathcal{Z}_i^{\mathcal{P}}$, the size of the i-th partition of q-grams from the pattern will be: $\forall i : \mathcal{Z}_i^{\mathcal{P}} = \frac{m-q+1}{|\Sigma|} = \frac{m - \lfloor \frac{m}{k+1} \rfloor + 1}{|\Sigma|}$. Thus, $\mathcal{Z}_i^{\mathcal{P}} \leq \frac{m}{|\Sigma|}$ since $\lfloor \frac{m}{k+1} \rfloor \leq \frac{m}{k+1} + 1$. Also,we have $\mathcal{Z}_i^{\mathcal{F}} = \frac{u}{|\Sigma|}$. Using these in **Lemma 2**, the cost of hypothesis generation will be:

$$Cost \leq |\Sigma| \log |\Sigma| + \lfloor \tfrac{m}{k+1} \rfloor . |\Sigma| . \tfrac{m}{|\Sigma|} . \log \tfrac{u}{|\Sigma|} \leq |\Sigma| \log |\Sigma| + \tfrac{m^2}{k} \log \tfrac{u}{|\Sigma|}.$$

Combine these with the $O(u)$ cost of computing the transformation vectors, and the cost of hypothesis verification as described previously, we obtain the results. For the worst case, we have used $O(ku + u) = O(ku)$. $\diamondsuit$

The complexity above can be further improved by using the improved QGREP algorithm described in Section 4.3 (rather than the QGRAM algorithm used in [AMB02]) during the hypothesis generation phase.

We then have following theorem.

Theorem 4: Faster k-approximate matching on BWT-compressed text. *Given a text string $T = t_1t_2\dots t_u$, a pattern $P = p_1p_2\dots p_m$, and an equiprobable symbol alphabet $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_{|\Sigma|}\}$. Let T be compressed by the BWT to produce a compressed output Z. There is an algorithm to locate the k-approximate matches of P in T, using only the compressed output Z in $O(|\Sigma|\log|\Sigma| + \frac{m^2}{k} + m\log\frac{u}{|\Sigma|} + \alpha k)$ time on average, ($\alpha = |\psi_{\mathcal{MQ}}| \leq u$), and in $O(|\Sigma|\log|\Sigma| + \frac{m^2}{k} + m\log\frac{u}{|\Sigma|} + ku)$ worst case, after an $O(u)$ processing on Z.*

4.6.4 Results

4.6.4.1 Experimental setup

4.6.4.2 Performance comparison

We compare the proposed compressed domain k-mismatch algorithm with a sufffix-tree based algorithm described in [Gus97]. Here, the k-mismatch check at any position i in T is performed using at most k longest common extension computation. Each computation can be performed in constant time, after the longest common ancestor table has been constructed. The suffix tree, however, usually requires a large storage (about $21u$ bytes), although the construction is in $O(u)$ time. The average pattern search time over the whole corpus used by the two algorithms are shown in Fig. 4.5 **(a)** for different k values. The results are for pure search time, and thus do not include the time to construct the auxiliary arrays (0.94s/MB) for our proposed BWT-based method, or the time to construct the suffix tree and then the LCA table (about 14.6s/MB) for suffix-tree based method.

As expected, the search time increased with increasing k. In all cases, the proposed BWT-based methods required less time than the well-known suffix-tree based method. The improvement increases rapidly with increasing values of k. As was predicted by the complexity analysis, we observe that the search time is virtually independent of m, the pattern length.

For k-approximate matches, we used our exact pattern matching algorithms for hypothesis generation, and Ukkonen's algorithm for verification. The construction time for Ukkonen's DFA is shown in Table 4.4. We compared the proposed method with two popular approximate pattern matching algorithms: AGREP [WM92b], and NRGREP [Nav01]. Both algorithms are based on bit-wise operations using the patterns and text. The two algorithms operate on the raw (uncompressed) text. The comparative results for the search time are shown in Fig. 4.5 **(b)**. Results for the proposed BWT-based approach is labeled BWT-dfa. Here, the hypothesis phase is performed by finding the exact matches for the r-grams using the QGRAM algorithm [AMB02], while the verification phase is performed using Ukkonen's DFA [Ukk85]. We have also included two other results: BWT-agrep and BWT-nrgrep. These correspond to when we used the proposed q-gram filtering approach for hypothesis generation, but with the verification phase performed with AGREP and NRGREP respectively. Since AGREP and NRGREP report only one pattern occurrence for each line in the text, the proposed methods were modified to report only one occurrence per line. The k-approximate matching results in Fig. 4.5 **(b)**, represent the average time to search for a single pattern in the whole corpus. At $k = 1$, NRGREP performed better than AGREP. The two produced comparable results at $k = 2$. AGREP's performance seems to improve with increasing k. At $k = 3$, it was clearly better than NRGREP.

The proposed BWT-based methods clearly outperformed AGREP and NRGREP, which operate on the uncompressed text. For our tests (with $m \leq 11$ and $k \leq 3$), the cost for DFA construction is minimal compared to the amortized search time. For each pattern, the DFA

Table 4.4: Construction time for Ukkonen's DFA

m	k=1	k=2	k=3
2	0.0005		
3	0.0008	0.0008	
4	0.0013	0.0021	0.0019
5	0.0018	0.0045	0.0053
6	0.0023	0.0081	0.0133
7	0.003	0.0135	0.0293
8	0.0039	0.0188	0.0568
9	0.0049	0.0237	0.0942
10	0.0063	0.0296	0.1401
11	0.0079	0.035	0.1844
15	0.0118	0.056	0.287
20	0.0167	0.08	0.58
34	0.1	0.52	3.13

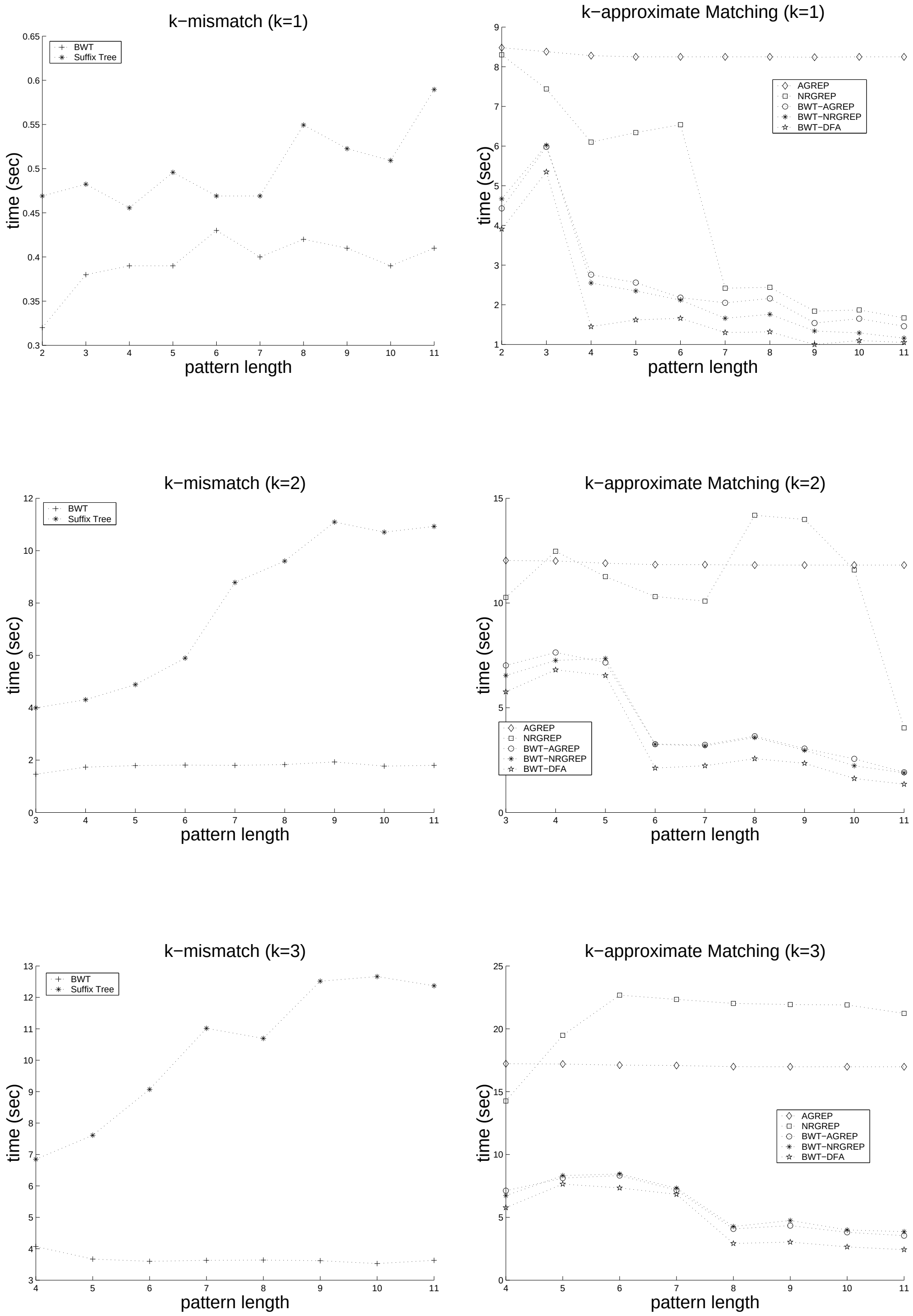
needs to be computed only once, independent of the number of files to search. For verification of a single r-gram, the time is almost constant using AGREP, NRGREP or DFA since they are linear, and the candidate string is only of size $m + 2k$. The fluctuation in search time (for BWT-based methods) comes mainly from the number of r-grams (mostly, $r = 1, 2, 3$ in our case) found at the hypothesis stage, and the number of verifications that failed in a line of text. Expectedly, the search time increased with increasing error parameter, k, since more hypotheses will be generated with increasing k. For a given k, the search time generally decreased with increasing pattern length m. The search time falls very rapidly beyond a certain value of m, typically around $m = 2k + 1$. This rapid decrease can be attributed to the relationship between r and m. For a given value of k, an increase in m leads to a direct increase in r. With a larger r, it becomes more difficult to find exact matches to the r-grams from the pattern. The result is a small number of hypothesis, and hence a shorter verification phase.

We observe that BWT-agrep, BWT-nrgrep, and BWT-dfa each produced shorter search times than the traditional AGREP and NRGREP algorithms. The reported search times do not include the time required for decompression (for AGREP and NRGREP), or the overhead for the auxiliary array construction (for BWT-based methods). When we need to search for multiple patterns across multiple files, both decompression/overhead time and search time become important factors in considering the search results. Fig. 4.6(a) show the time used for searching for multiple patterns including the one time decompression/auxiliary array overhead. It indicates that the amortized cost of our algorithms are lower when the number of search patterns exceeds around 20 for $k = 1$, 10 for $k = 2$, and about 8 for $k = 3$. The results imply that, for applications that involve searching for multiple patterns (for instance, in text retrieval or Internet search engines), the proposed BWT-based methods would produce superior results. Fig. 4.6(b) show the effect of file size on the performance of the k-approximate matching algorithm. This is based on results for 100 randomly selected patterns from the English dictionary. It shows that the search time is almost linear with respect to u, the file size. Most importantly, the proposed BWT-based methods result in a slower growth in search time, as compared with traditional pattern matching algorithms, such as AGREP, NRGREP.

The BWT with its sorted contexts provides an ideal platform for compressed domain pattern matching. We have described algorithms for pattern matching with errors on BWT-transformed text, and showed their significant advantage over popular *decompression-then-search* approaches. The proposed algorithms could be further improved. For instance, the space requirement could be reduced by considering the compression blocks typically used in BWT-based compression schemes, while the time requirement could be further reduced by using faster pattern matching techniques for the q-gram intersection (hypothesis verification). One way to reduce the relatively high overheads will be to consider pattern matching on blocked BWT-compressed files, since these will typically involve smaller text sizes per block.

The overhead of DFA construction could be rather high (compared to the search time), especially with increasing k. But once a DFA is constructed for a pattern, it can be used to find any other pattern with edit distance less than or equal to k without changing the DFA. The proposed approach will thus be very effective in dictionary matching, where one may wish to search for a pattern in multiple text sequences. One possible improvement on the hypothesis verification phase could be to use dynamic construction of the DFA. However, the proposed k-approximate match algorithm does not depend on any particular verification algorithm. This could also be observed from the results with BWT-agrep and BWT-nrgrep. Thus, one could abandon Ukkonen's DFA altogether, and look for alternative faster verification algorithms.

Proposed BWT-based search methods have mainly focused on the output of the BWT transformation stage. A long-standing challenge has been to extend the approach to operate beyond the BWT output, i.e. after the further encoding stages in the BWT compression pipeline. We have described a modified MTF algorithm that can be used to support searching directly on the outputs from the MTF stage. When this is coupled with recent results on pattern matching at the VLC output, we have a potential for pattern matching at any stage of the BWT compression pipeline. An interesting work will be how to tie these different pieces neatly together.



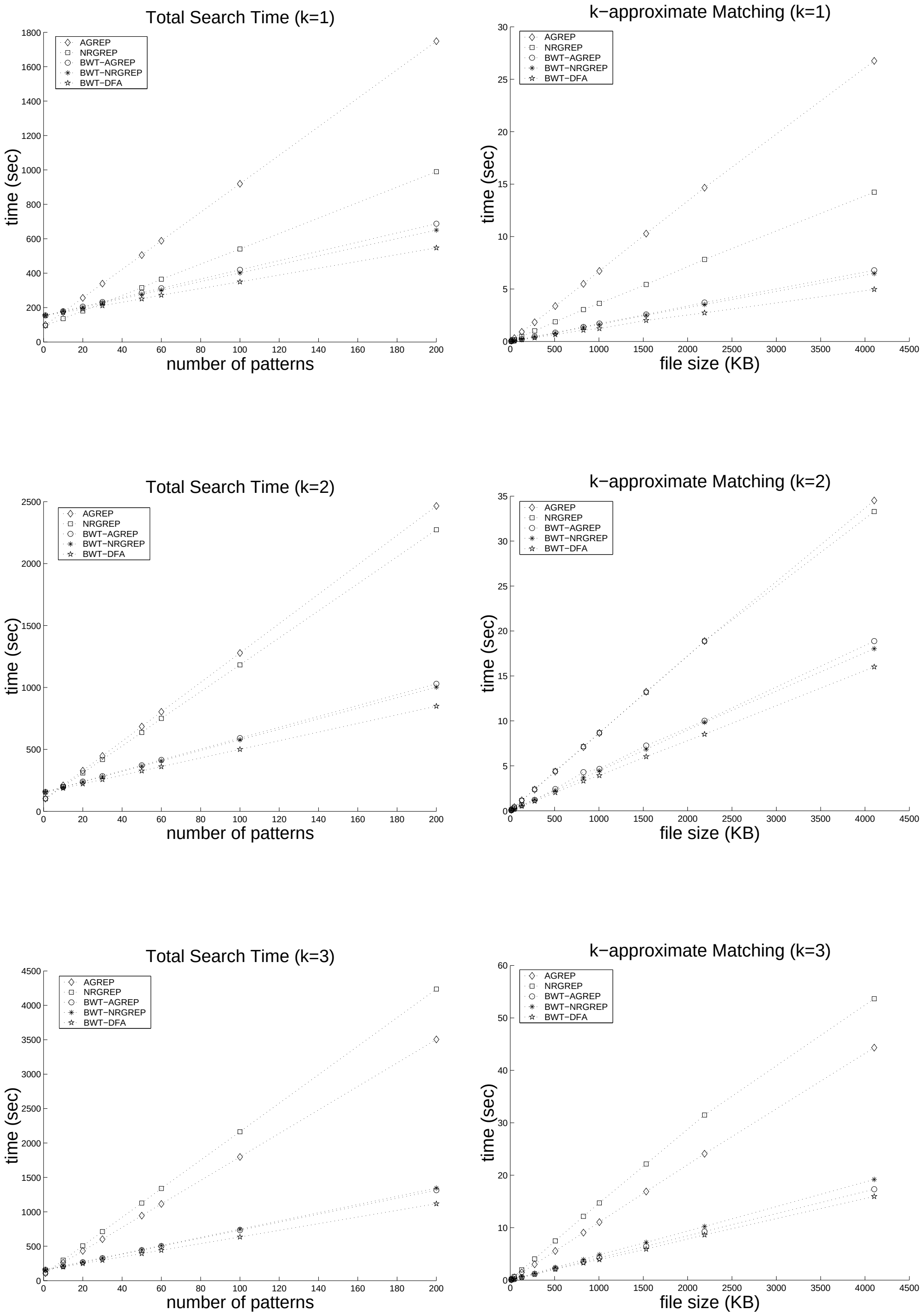
k=1

k=2

k=3

(a) **(b)**

Figure 4.5: Search time for k-mismatches (a) and k-approximate match (b), for various values of k



$k = 1$

$k = 2$

$k = 3$

(a) **(b)**

Figure 4.6: Variation of total search time (including decompression/or array construction overheads) with **(a)** number of patterns and **(b)** file size

CHAPTER 5

TEXT INFORMATION RETRIEVAL ON COMPRESSED TEXT USING MODIFIED LZW

We have introduced the motivation and some background on compressed text searching in Chapter 1 and Chapter 2. In this chapter, we explain how do we select an appropriate compressed scheme and integrate it with the current text retrieval systems.

5.1 Problem Description

Various text compression algorithms have been proposed to reduce the file size in a reasonable amount of time. As we described in 2 They can be roughly categorized as symbol-wise coders and dictionary-based coders. Huffman coding or Arithmetic coding are typically used to determine the actual codes used for both approaches. The LZ family, including LZ77, LZ78, LZW, and their variants [ZL77, ZL78, Wel84] are the most popular dictionary-based compression algorithms because of their speed and good compression ratio. In terms of compression performance, however, the symbol-wise PPM (Prediction by Partial Matching) family of algorithms [CW84]currently produces the best results. However, PPM typically requires much more time. The Burrows-Wheeler Transform (BWT) [BW94a], or block-sorting algorithm has a compression ratio close to PPM (not surprisingly, since it uses similar features of the text to obtain compression) and a speed that is slightly slower than the LZ family. Other well-known symbol-wise text compression methods include Dynamic Markov Coding (DMC) [CH87] and Word-based Huffman [WMB99].

When the problem is to search the compressed data, an important issue is whether the compression algorithm can support direct searching or partial decoding. For example, in Huffman coding, given a query keyword, we can obtain the corresponding codewords from the Huffman table and then search for these codewords directly in the compressed file, using pattern matching algorithms such as the Boyer-Moore or Knuth-Morris-Pratt algorithms. However, this might lead to false match since the code for the query words might be an internal subsequence of a valid Huffman code of an unmatched words. In word-based Huffman codes, a similar method can be used to search for a word, or for partial decoding [ZMN00] For most other compression algorithms, random access is difficult to provide, thus we have to decode from the beginning of the compressed text or from artificially inserted synchronization points.

The amount of storage used and the efficiency of indexing and searching are major design considerations for an information retrieval system. Usually, there is a tradeoff between the compression performance and the retrieval efficiency. Some simple compression schemes, such as run-length encoding provide easy indexing and fast search but a low compression ratio []. Methods have been proposed for compressed domain search and retrieval using the Burrows-Wheeler Transform (BWT), Prediction by Partial Matching (PPM), and other techniques (see Section 2). Although PPM and BWT provide the best compression ratio, performing retrieval is difficult, whether directly or through indexing on the compressed file. Word-based Huffman coding schemes [CW02, ZMN00, MSW93a, WMB99], on the other hand, provide a better balance between compression ability and performance in indexing and searching.

Recently, several researchers have proposed exact and/or approximate pattern matching algorithms that search patterns directly on the compressed file with or without preprocessing (See Chapter 2 and 4. The motivation includes the potential reduction of the delay in response time introduced by the initial decompression of the data, and the possible elimination of the time required for later re-compression. Further, with the compact representation of the data in compressed form, manipulating the smaller amount of data directly will lead to some speedup in certain types of processing on the data such as searching and browsing. The search time can be further improved if the collection of documents can be broken into smaller sets of documents [MSW93b, MZ94]. For compressed domain search, this might affect the overall compression ratio which typically improves with file size - with a bigger the file size, we have a better opportunity to model the type of text in the document.

Given a query using keywords, the most obvious approach to search compressed database is to decompress-then-search, which is not very efficient in terms of search times. Compressed domain pattern matching is an efficient method but it tends to solve the problem on per-file basis. Most practical information retrieval systems build an index or inversion table [BR99a, KT00, MSW93a, WMB99], combined with document frequency using the key words. The documents are ranked using some standard to achieve good precision and recall. Relevant feedback may also help to refine the query to have more accurate results. Such systems using inverted file will need the use of larger size index table with increase of file size. It is possible to adopt an approach to information retrieval which uses compressed domain pattern search, not on the original text, but on the compressed inverted file yielding pointers to documents that are also in compressed form.

Depending on the application, the target for the retrieval operation may vary. Usually only a small portion of the collection that is relevant to the query needs to be retrieved. For example, the user may ask to retrieve a single record, or a paragraph, or a whole document. It is unnecessary to decompress the entire database in order to locate the portion to be retrieved. Using a single level document partitioning system may not be the best answer. Thus we consider the document at different levels of granularity, and propose to incorporate context boundary tags in the document. Different tags indicate different levels of granularity, and decoding will be performed within these context boundaries.

5.1.1 Components of a compressed domain retrieval system

When the objective is information retrieval, the major considerations for a compression scheme, ranked roughly by their importance, are:

- random access and fast (partial) decompression;
- fast and space-efficient indexing;
- good compression ratio.

The compression time is not a major concern since the retrieval system usually performs off-line preprocessing to build the index files for the whole corpus. Besides the searching algorithm, random and fast access to the compressed data is critical to the response time for the user query in a text retrieval system.

We have shown a typical text retrieval system structures in Figure 2.1. The right half of the figure shows the structure of a traditional text retrieval system. In this chapter, we will discuss the structure and algorithms for compressed text retrieval. The rest of the chapter will explain the components on the left half of Figure 2.1, which relate to compressed domain text retrieval. The index construction parts on the right hand side are now done with respect to the compressed text.

5.1.2 Our Contribution

In this thesis, we propose a new text searching algorithm based on the LZW compression algorithm. We incorporate random access and partial decoding in LZ-compressed files using an inverted index. The algorithm is based on an off-line preprocessing of the context trie and is specifically designed for searching library databases using key words and phrases.

The algorithm uses a public trie or a "dictionary" which is trained for efficiency using a pre-selected standard corpus. More specifically, we make the following contributions.

In contrast to the traditional LZW algorithm which is online, our initial approach uses a two-pass off-line method. In the first pass, the LZW dictionary is built and the dictionary is used to compress the text in the second pass. The dictionary is stored as a trie in the implementation. The trie is transmitted along with the text. In the actual implementation, the trie usually has a fixed size, for example, 4k bytes. A second approach is to obtain a common dictionary trained using a large corpus. All other texts are compressed using the common dictionary (also called public trie). This dictionary only needs to be transmitted once and it is stored at both the encoder and decoder. During index file construction and decompression, the node numbers corresponding to index table entries will refer to the public dictionary. In our algorithm, we can decode any part of the text given the index of the dictionary entry and stop decoding when a certain tag is found, or after decoding a given number of symbols. The modified LZW uses a fixed-length code for each node, which we choose to be 16 bits/node. That is, the code is at byte level so that we can still perform fast random access when the indexing granularity increases or decreases.

In our approach, we allow for multiple levels of granularity to be stored in the LZW-coded file by adding tags for different levels of details. The pointers in the inverted index file will be built on the tags. The decoding will be performed by the nature of the query or by user defined level. Although there are systems based on word Huffman with good compression ratio and random access ability [CW02, WMB99], they can only perform random access at a fixed granularity. For example, they partition the files into blocks by paragraph. The system will compress the text block by block. Then display the blocks that contains the keywords during retrieval. We call the level of details to be at paragraph level. User may require showing the result in different levels such as word, line, paragraph, and file, etc. In our approach, we add tags for different levels of details instead of a fixed level of granularity. The pointers in the inverted index file will be built on the tags. The decoding will be performed by the nature of the query or by user defined level. For fixed level partitions [CW02, WMB99], if we need to access more details than the fixed blocks, the new pointers are needed in the index table pointing to the bit level in the compressed text instead of at byte level which may make the pointer size larger. For example, it needs an extra byte per pointer to achieve the level of granularity as our approach does.

Furthermore, our approach offers potential advantages to exploit parallelism and error resilience. Our byte level code brings easy parallel access to any part of the compressed text. The experiments on compression indicate that the best compression ratio is achieved when each trie node is encoded in 16 bits. That means we can decode the sequence from any even number of bytes after the header. Multiple processors can perform the navigation on compressed code in parallel without mutual exclusion.. In our coding scheme, each node is encoded with fixed number of bits. In case there is an error occurring at some position, the error is limited within the range of subsequence represented by the corresponding node. Further decoding will not be affected.

5.1.3 Compressed Domain Pattern Search: Direct vs. Indexed

Different compressed pattern matching methods have been proposed directed towards compression schemes based on the Lempel-Ziv (LZ, for short) family of algorithms. The algorithms can search for a pattern in an LZ77-compressed text string in $O(n \log^2(u/n) + m)$ time, where $m = |P|$, $u = |T|$, and $n = |Z|$ [FT98]. Methods that are more focused on text retrieval, rather than pattern matching have also been proposed. An off-line search index - the q-gram index, was proposed for the LZ78 algorithm in [KU98]. Also, the LZ trie has been used for text indexing and retrieval, with emphasis on LZ78 [Nav02]. A special data structure is built taking bits of space and can report all the occurrences of the pattern. The focus on LZ might be attributed to the wide availability of LZ-based compression schemes on major computing platforms. For example, GZIP and COMPRESS (Unix), PKZIP (MSDOS) and WINZIP (MS Windows) are all based on the LZ algorithm. In general, the LZ-family are relatively fast, although they do not produce the best results in terms of compression ratio. Methods for pattern matching directly on data compressed with non-LZ methods have also been proposed (see Chapter 4).

A vast amount of literature is available on database indexing, query evaluation, relevance feedback etc. and other techniques to improve recall and precision [BR99a, KT00, MSW93a, WMB99], Partial decoding has been recognized as an important aspect of a compressed text retrieval system [MSW93a]. Similar to accessing a shot in a video stream, locating a small portion of the text is critical to text information retrieval, especially when the text is compressed. In most compression algorithms with a high compression ratio, the context information is lost. Hence, it is difficult to access an arbitrary portion of the text using the current information only. There has also been some work on word-based Huffman compression schemes [CW02, ZMN00, MSW93a, WMB99]. In [MSW93a] partial document search is performed by breaking documents into small blocks. The output unit is the block and is fixed by the indexing, so random access is not as flexible as it might be. Users may need to retrieve data at different levels of the detail in the text. For example, the results to a query could be a pattern, a paragraph, a record, or a whole document. We should be able to decode the text as close as possible to the request. Therefore the coding scheme should provide enough flexibility for partial decoding. There are some concerns about the extra space requirement in the index file because of the breakup of the documents. In [MSW93a], the effect of indexing is measured showing that the inverted file is still efficient with more entries since the documents are broken into smaller blocks. In [CW02], the XRAY system also has a training process before compression. Statistics of the phrases are collected and different options for partitioning the string into different phrases are provided. The phrases are then encoded using Huffman codes. In both the word Huffman and phrase based systems [CW02, WMB99], random access is possible for the text at a the current level of granularity or lower. If we need to access smaller units of text, methods that are based on Huffman codes will need to break up the text again. Otherwise, they may need to store pointers to positions inside a block, in which case the size of the pointer may be larger, since the code is at a bit level, compared to the byte level code (or else space must be wasted by putting in extra byte aligned synchronization points).

5.2 Our Approach

To address the problem of random access and partial decoding and to provide flexibility in the indexing and searching, we need to remove the correlation between the current code and the history information (that is, the algorithm cannot use a single-pass adaptive approach). In this section, we describe a modified LZW approach that aims at partial decoding on compressed files, while maintaining a good compression ratio. The dictionary is language independent and is built from the text. A tag system is suggested to output the retrieval results in different scope. Our fixed-length coding scheme also helps to build the index and defines the boundary of the text segments at different levels of granularity. This also facilitates parallel access to the compressed text.

5.2.1 The LZW algorithm

The LZW algorithm [Wel84] is one of the many variations of the Ziv-Lempel methods. The LZW algorithm is an adaptive dictionary-based approach that builds a dictionary based on the document that is being compressed. The LZW encoder begins with an initial dictionary consisting of all the symbols in the alphabet, and builds the dictionary by adding new symbols to the dictionary as it encounters new symbols in the text that is being compressed. The dictionary construction process is completely reversible. The decoder can rebuild the dictionary as it decodes the compressed text. The dictionary is actually represented by a trie. Each edge in the trie is labeled by the symbols, and the path label from the root to any node in the trie gives the substring corresponding to the node. Initially, each node in the trie represents one symbol in the alphabet. Nodes representing patterns are added to the trie as new patterns are encountered. As stated in section 4, Amir proposed an algorithm to find the first occurrence of a pattern in an LZW-compressed file [AC96]. The pattern prefix, suffix, or internal substring is detected and checked to determine the occurrence of the pattern. The pattern is searched at the stage of rebuilding the dictionary trie to avoid total decompression. Obviously, this method cannot satisfy the request to find multiple occurrences in the large collections of the text data in the compressed format. Modifications to the algorithm have been proposed in order to find all the occurrences of a given pattern [TM04].

In the context of indexing in a compressed archival storage system, there are a few disadvantages with the original LZW approach. LZW uses a node number to represent a subsequence, and all the numbers are in sequential order. For example, as shown in Figure 5.1, given the index of the word "pattern", located at the 12th position in the compressed file, we can determine that the node 3 in the dictionary contains the beginning of the word. However, we are not able to decode the node and its neighboring text because the trie, which needs to be constructed by sequential scanning and decoding the text, is not available yet. In order to start decoding the file from some location in the file, the trie up to that location is necessary.

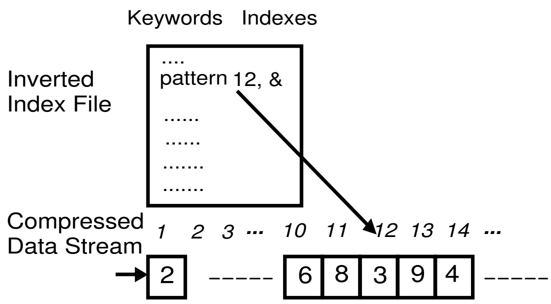


Figure 5.1: Illustration of indexing for an LZW compressed file.

5.2.2 Modification to the LZW algorithm

5.2.2.1 Off-line compression with file-specific trie

We can change an online LZW algorithm into a two-pass off-line algorithm. Figure 5.2 shows an example of a trie. A pattern can be either within a path from the root to the node or be contained in the paths of more than one node. Let us consider the text "aabcaabbaab" with alphabet $\Sigma = a, b, c$. If we compress the text at the same time when the trie is being built as in current LZW, the output code will be: "11234228". The encoder is getting "smarter" during the encoding process. Take the sub-string "aab" as an example; it appears in the text three times. The first time it appears the encoder encodes it as “112”; the second time it is encoded as “42”; the third time it is encoded as “8”. If each codeword is 12 bits, the encoder encodes the same substring as 36, 24 and 12 bits at different places. Thus, we may also consider the encoding process as a ”training process” of the encoder or, specifically, the trie.

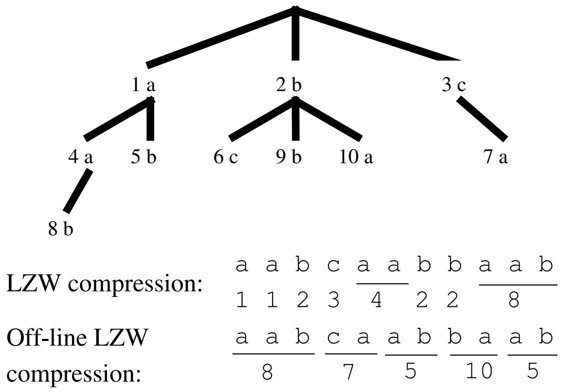


Figure 5.2: Example of online and off-line LZW.

The above example indicates that, if we can ”train” the trie before any compression has started, we may get better compression. This is the basic idea of the two-pass compression scheme proposed in this paper. In this scheme, the first pass is the training process, which builds the entire trie by scanning the text. The second pass is the actual compression process, which compresses the text from the beginning of the text using the pre-constructed trie. For the above example, the text will be encoded as: “875(10)5”. More importantly, since the text is encoded after the trie has been built; it uses the same trie at any point of the encoding process, unlike the original LZW approach, which uses a trie that grows during the encoding.

Thus, decoding from any point in the compressed stream is possible. For example, given the compressed data "(10)5" and the above trie, we immediately decode it as "baab".

In this approach, a separate static dictionary is used for each file. The dictionary trie is attached to the compressed file to insure that random access is possible. The size of the trie can be decided by a parameter indicating the number of bits to be used for the trie. The larger the file to be compressed, the less will be the effect on the compression ratio of the extra trie overhead. A 12 bit trie occupies at most 9064 bytes of space when written to a file. The first 256 nodes in the trie need not be written to the file for obvious reasons. One node number is reserved for special purposes, and hence a total of $(2^{12} - 256 - 1) = 3839$ nodes are written to the file. Each node takes 20 bits of space, 12 bits for the parent id, and 8 bits for the node label. Four extra bytes are necessary for special purposes, and hence the total space required for the trie is $3839 * 2.5 + 4 = 9063.5 \approx 9064$ bytes. Thus for a large file, the size of the compressed file plus the dictionary is very close to results for the LZW. This is in line with Cleary and Witten's results [CW84] that show a close relationship between the size of a statically compressed file plus its model, compared with an adaptively compressed file. We can partially decode a portion of a file given the start and end locations of the node reference in the compressed node string, or start location of a node and the number of nodes to decode, or a start location and a special flag indicating decoding has to stop. The trie for a given file has to be read and loaded into the memory in order to decode parts of that file. The disadvantage is that the trie overhead could become significant when the file size is small.

5.2.2.2 Online compression with public trie

Another approach is to use a public static dictionary trie built off-line based on a training data set consisting of a large number of files. This static dictionary trie is then used to compress all the files in the database. In a network environment, both the encoder and decoder keep the same copy of the public trie. For archival text retrieval, the trie is create once and installed in the system and might undergo periodic maintenance if needed. The public trie needs to be transmitted only once. The text is compressed using the trie known to every encoder/decoder. The compressed files are sent separately without the trie. The decoder will refer to the public trie to decode the compressed file. Since the dictionary captures the statistics of a large training set that reflects the source property, the overall compression ratio is expected to improve.

Of course, some files may have a worse compression ratio than that obtained by using the original LZW algorithm, and (as with any static compression method) there is the risk that some files will have very poor compression for the public trie, although in practice this is unlikely if the database is reasonably homogeneous.

Since the trie size is relatively small compared with the overall text size in the text collection, the amortized cost for the whole system is less than using the original LZW or LZW with individual trie. Another advantage of using a public trie over a file-specific trie is that the words will be indexed based on a single trie. Instead of indexing with a document trie number and the node number inside that trie, we can simply use the node number that is common to all the files in the system.

Figure 5.3 illustrates the difference among the current LZW, the two-pass off-line LZW, and the public trie LZW. The horizontal bars represent the text file from the first symbol to the end-of-file symbol. Figure 5.3a shows the current implementation of LZW. Usually, the trie is not based on the whole text file, but a limited sample from the beginning. The shaded areas indicate the beginning portion of the text that is used for constructing the trie and compressing the text simultaneously; we perform compression only for the rest of the text. To retrieve such a compressed text, we need to reconstruct the trie and then search. Figure 5.3b shows the two pass compression process. The entire text is used to build the trie without any compression. Then actual compression is performed on the whole text from the first symbol after the trie is built. Since the traditional LZW algorithm uses a greedy method to do pattern matching in the trie to find a representing node number, the beginning portion of the text may have a better compression ratio using a fully built trie. Figure 5.3c shows the public trie method that compresses the text file by first constructing the trie using the training text to capture the statistics of the source, then compressing all the (other) text using an existing trie.

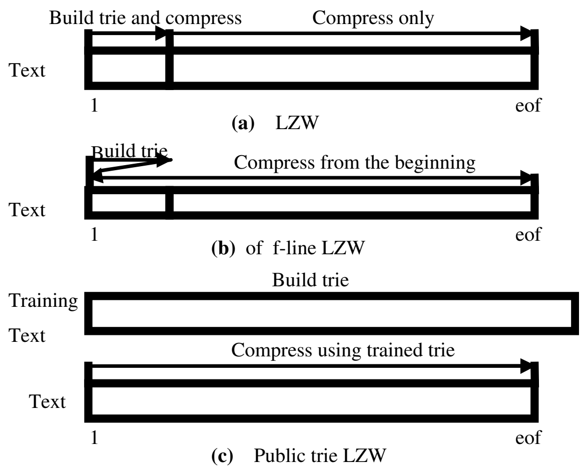


Figure 5.3: Illustration of online, off-line, and public trie LZW approach. The shaded part of the text in (a) and (b) is used for training the trie.

5.2.2.3 The searching performed based on public trie compression

To perform the text indexing and searching on a collection of the text, our method works as follows:

1. Training: Train a trie using a sample text corpus. We run the LZW compression with randomly picked samples with total size of about 10MB symbols from a text corpus. A trie is output as the result instead of actual compressed version of the text set. The optimization of the training process to achieve a good compression ratio will be discussed later.

2. Compression and Indexing: Texts are then compressed using the algorithm stated below which we call the modified LZW (MLZW) algorithm. The index table is also built during the execution of the algorithm.

   - Load the output trie obtained in step 1.
   - Compress the text using the public trie. We find the longest matching path of the input substring to the leaf. Output the node number as part of the compressed file. At the same time if a key word (as defined by the user) appears in this phrase, we also enter into the index table the position number of the tag preceding this phrase which is the beginning of the block. The block is predefined such as a paragraph, a record etc. There are natural block boundaries such as line breaks and space or user defined block boundaries such as the tags.
   - Compress the entire corpus and index all the keyword in the given list. Therefore, we have a set of compressed files encoded using the public trie and a index table containing all the keywords and pointers to the document id and the locations of the encoded boundaries in the compressed text.

3. Retrieval: Given a query word, we look for the corresponding entry in the index table. Then the actual keyword is found in the index table

   For each pointer in the index table for that keyword,

   (a) Find the compressed document.

(b) Directly locate the address given by the pointer which gives the beginning of the block containing the keyword and start decoding using the public trie.

   i. if block boundary is found, start outputting the text after it.

   ii. continue decoding until the next block boundary is found in the text.

If we need to locate the exact location of the query word. We can ether build the index by assigning the block boundary pointer to the word or use a simple pattern matching algorithm such as Boyer-Moore to search in the decompressed portion of the text. Although the second approach has an extra overhead, the search time is minimal since the size of the decoding part are usually very small.

5.2.2.4 Finding a public trie for good compression ratio

Although compaction ability is not a major issue in retrieval (as compared to efficient indexing and searching), it is useful to obtain good compression for a given text corpus. Thus, an ideal compression scheme for retrieval should not degrade the compression ratio. Ideally, it should produce some improvement since extra pre-processing has been used. There could be many ways to build a public trie depending on the context of the text. Examples here could be:

1. Construct a trie with respect to a randomly chosen text files.

2. Find a trie that best compressed the test files and use that as the public trie.

3. Update the trie when the space allocated is full. There could be many criteria to insert/remove a node in the trie. For example, when a new substring occurs, we could prune the node representing the least frequently used substrings. The new substring is added and other updates are performed accordingly. Our aim is to store the most frequently used context in the trie.

4. Build the trie from the frequency of the q-grams (substring that has a length of q). The current PPM mechanism may possibly help to decide which q-grams would be used.

There is no need to construct the trie anymore for searching the pattern if a public trie is used. Therefore,the overhead of constructing the trie is avoided, but the pattern matching complexity, i.e. searching the substring in the existing trie, remains the same. The size of the trie is another factor that may affect the compression performance. If the trie is large, we may find longer matching of the substrings. However, we need a larger code length for each node. There is a tradeoff between the code length and the size of the trie. We are currently using the commonly used trie size in the popular LZW implementation.

A possible solution to find a good public trie based on the LZW dictionary construction method and frequency statistics is as follows. First, we set a relatively large upper bound for the trie size. In addition to the current LZW dictionary construction algorithm that adds the symbol and assigns a new node number, we also add a count to the number of accesses to each node. Note that all the parent nodes have bigger counts than any of their children nodes - in fact, the count of a parent node is the sum of the counts of its child nodes. When the whole trie is built, we start to remove those nodes that have counts less than a threshold value. Obviously, the leaf nodes will be removed first. No parent node will be removed before the child nodes. It is possible that a leaf node has a larger count than a parent node in another branch in the trie, so we will remove the nodes in a branch with a smaller count in a bottom-up order. The pruning will proceed with possibly more than one iteration until a predefined trie size is reached.

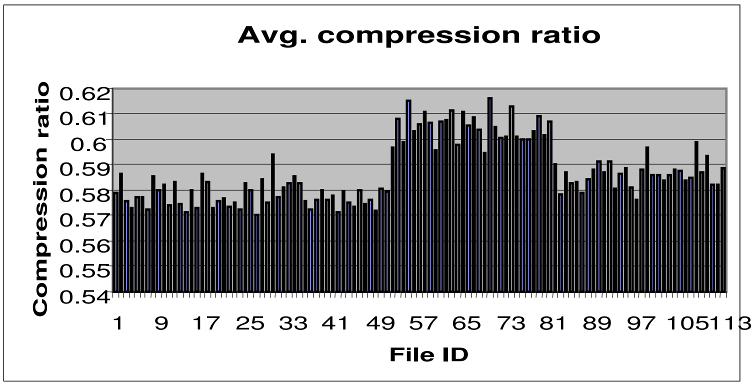


Figure 5.4: The average compression ratio over the corpus using the trie from each file in the corpus.

Figure 5.4 shows the compression ratio using the public trie method. The trie has been generated from a single file in directory AN in the corpus and then applied to all the files in the corpus. The average compression ratio is shown before the node sequence is further compressed using an entropy coder such as Huffman. The compression ratio here is defined as compressed file size / original file size. We pick a trie from an individual file and use the trie to compress all the files in the test corpus and compute the average ratio over all the files represented as a bar for that file ID. A small test corpus is selected from disk 1 of the TREC TIPSTER collection [TRE00]. There are 113 files in total, belonging to three categories AP (51 files) , DOE (30 files), and FR(32 files). The total size of the corpus is about 115 Mbytes. The figure indicates that using different trie has some impact on the compression ratio. It is interesting to observe that the three categories can be roughly distinguished by their compression. It also shows that the difference of compression ratio does not vary too much (within a 5% range), justifying the idea of using of a public trie to compress all the text.

5.2.3 Indexing method and tag system

When an inverted index file is built for the text collection, the keywords are indexed with the document ID and the location where the keywords occurred in the document. In many cases, users are not interested in the exact location of the occurrence, but are interested in the text in the neighborhood of the occurrence. Therefore, we can index the keywords by the block location instead of word location, along with the document ID. For example, if the keyword "computer" is located at the 5th paragraph; we simply have a pointer pointing to the starting address of the 5th paragraph in the inverted index file. When the keyword "computer" is searched using any query search method based on the inverted index file, the 5th paragraph will be returned as the result and the whole paragraph will be displayed. If exact location of the keyword needs to be highlighted, a simple pattern matching algorithm such as Boyer-Moore or Knuth-Morris-Pratt algorithms can be used in the very small portion of the text. The definition of "block" is quite flexible. It could be a phrase, a sentence, a line of text, a paragraph, a whole document, a record in a library or database system, or any unit defined by the user. Thus the size of the block determines the granularity of the index system.

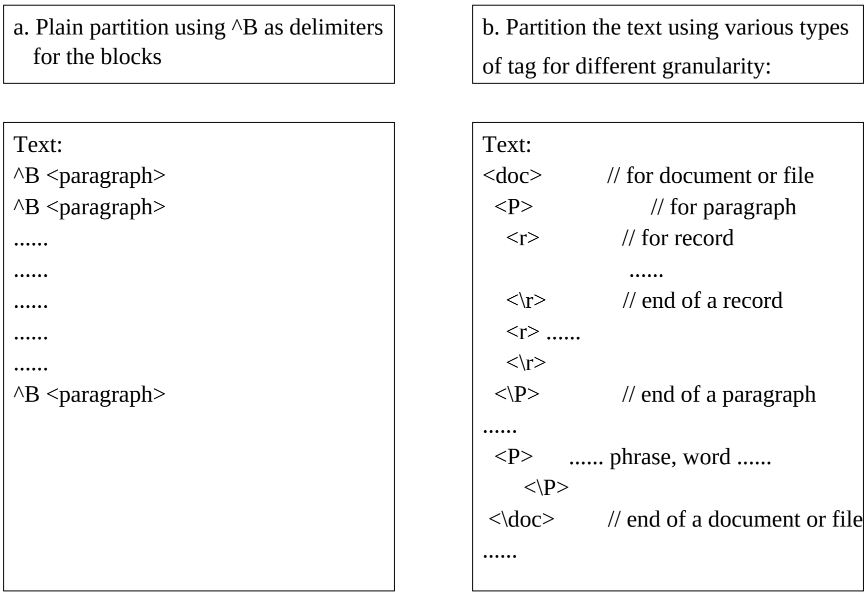


Figure 5.5: The difference between single level partition and multiple level partition of the text.

To provide different levels of granularity in a single index system, we can build a hierarchical system that explores the structure of the text by using the tags inserted at different levels of the file. Although tags have been previously used in retrieval systems (see [WMB99] for example), implementations are generally limited to just one level of hierarchy. Figure 5.5a illustrates the fixed level partition in [WMB99], where compression and index file construction are performed on the block level only using '^B' as a block separator. Taking a cue from XML, a hierarchical tagging system that has been a standard for hypertext, we can add different user-defined tags to indicate the text boundaries for different levels of granularity. For example, we can use `<r>` and `<\r>` as record boundary, `<P>` and `<\P>` as paragraph boundary, `<doc>` and `<\doc>` as document or file boundary, etc. Some boundary indicators occur naturally within the text such as line break and paragraph. Figure 5.5b shows an example of the multiple level tags. Figure 5.6 shows a typical hierarchical structure for the different levels of the granularity. Then index table can be constructed accordingly to access different lengths of context.

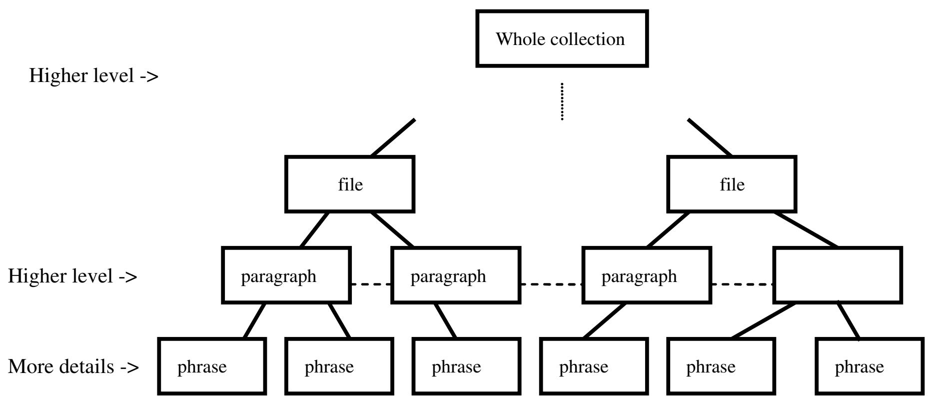


Figure 5.6: The tree indexing structure for different granularities.

We estimate the space cost of the tags as follows. If the file has already been preprocessed by converting to HTML or XML style, tags have already been considered as part of the text. Thus there is no more extra cost. If the tags are inserted for special purpose, the tags will not affect the text size too much. Take the example in [MSW93a]; the average block size is

1,235 bytes, which means if we add a tag </B> for each block, the file size only increase by 0.3%. The paper also shows that the inverted index file size increases by a reasonable size for retrieval. In contrast to [MSW93a], our tag system has a hierarchical format that can satisfy different levels of detail for the text. Although we may have more delimiting tags for different levels of granularity, the number of tags may decrease drastically when the scope of the text represented by the tag is getting larger. Therefore the amount of the extra space for the tags for all the levels will still be small. For example, if the lowest level tag represents 1Kbytes , and each tag in a higher level represent 100 units of the text in the lower level, 1G bytes of text needs a total of (4*1.01001) Mbytes for the tags if each tag takes 4 bytes. The total size increases by just about 0.4%. So, the text can be compressed together without breaking it into small pieces, which may lead to a better compression ratio.

In general, the tag system is very flexible without any conflict with the trie representation in the modified LZW algorithm. In the system provided in [WMB99], if lower resolution is needed, the system can output multiple blocks. As shown in figure 7, we can use a simple tree data structure to represent different levels of the granularity for the text. Several paragraphs can be indexed to a section, several sections to a chapter, etc. until finally the whole collection becomes one entity. If we need to index downward in the fixed system as [WMB99], one option is to reconstruct the whole index file by breaking the system into smaller units. Another option is to generate the new pointers only for the higher levels of granularity. This will cost less reprocessing time. The new pointers would point into the current block. Since each block is encoded using bit level variable length code, the pointer size will be bigger than that pointing to the byte level code. For example, we might use an extra byte for the intra-block pointers. With our compression scheme and tag system, the intra-block pointer will simply point to the byte that represents the node containing the boundary tag. We can start the decoding immediately from that point using the public trie without further checking of the starting bit which is necessary for Huffman code.

5.2.4 Partial decoding with the tag system

We now consider the partial decoding issue with our tag system. Similar to the English word-based model, each word also corresponds to a node in the LZW dictionary and uses the node number as the code. However, we need to distinguish between a linguistic word (L-word) in a language dictionary and a word (T-word) in the dictionary trie obtained by training. A T-word stands for a sequence of symbols. As such, the boundaries of the T-words may not coincide with the boundaries of the L-words. A T-word may contain multiple L-words, or parts of one or more L-words. If we are searching for a particular L-word in the text, we need to have some mechanism to assemble the parts of the word into a valid L-word. A node in the trie may also represent more than a single L-word. Although this is an extra pattern assembling stage compared to English word based compressed pattern searching methods, it is language independent. The cost of the assembly is still linear but is slightly larger by a constant factor compared with other pattern matching algorithms. During decompression we can recognize the tag (a predefined L-word) and then only output the text between the beginning and the end tag indicating boundaries. For example, the index of a keyword may point to the location 200, where node number 6 is located. Node 6 may represent a subsequence "ion `<B>`thi". We then have to output the text after "`<B>`" and continue decoding the succeeding nodes until the next "`</B>`" is encountered. To avoid the nonsense characters before and after the tags, we can assign some frequently used tags in the trie during initialization stage of the training. Since the low level tags may be very frequently used, we can set the tag to be a node in the trie manually. So the text boundary can be easily recognized by the unique node number.

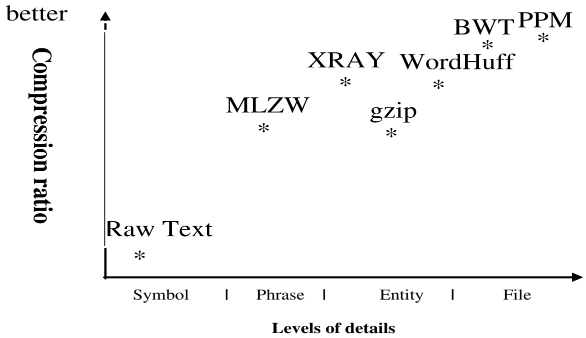


Figure 5.7: Compression ratio vs. Random access. The block here usually refers to a text size equal to a paragraph or other predefined size. MLZW refers to our modified LZW algorithm with a pruned trie. The code size is 16 bits.

5.2.5 Compression Ratio vs. Random Access

Besides the equal length byte level code that provides the possibility of dynamic indexing for the context with various resolutions, it also provide the flexibility for the random access. Figure 5.7 shows qualitatively the tradeoff between compression ratio and flexibility of random access for the various compression schemes. The PPM and BWT algorithm give the best compression ratio but can not perform partial decoding at any point. Usually, we have to decode the whole document for further processing. BWT implementation has an option for different block size. The coding for each block are relatively independent. Therefore BWT has a slightly finer resolution than PPM. The WordHuff and XRAY provide a better compression ratio than the popular GZIP as well as a better random access at the level of predefined blocks or document. Our modified LZW using public trie provides easier random access than the other compression algorithms and a compression ratio similar to GZIP. In our approach, we can access to the level of the phrases defined in the public dictionary. Raw text is considered the extreme case of no compression and full random access at the symbol level.

5.3 Results

5.3.1 Experimental Setup

To evaluate the performance of the modified LZW approach for partial decoding, we performed experiments on a text database. The database is made up of a total of 840 text files selected from the TREC TIPSTER1993 corpus including Wall Street Journal (1987-1992) and part of AP, DOE and FR files from Disk 1 of the TIPSTER corpus. The file size is 650MB. The tests were carried out on a Pentium-II PC (300 MHz, 512MB RAM) running RedHat Linux 7.0 operating system. In the current LZW implementation, a reference number is used to indicate the code size and the maximum trie size for the dictionary. For example, if we take 12 bits as reference, each node is coded as a 12 bit sequence. The storage of the trie will actually be a prime number greater than . The prime number is used for in a hashing function to expedite the access to the trie nodes. With a larger number, we have a larger size for the trie.

Experiments were performed on different trie sizes and file sizes. In the original LZW, the beginning part of the text is used to build and update the dictionary trie until the trie is full. Then the rest of the text uses the trie to compress without updating the trie. To test our off-line method using file-specific trie, we use the beginning part of the text to build the trie and use the trie to compress the text from the starting point of the text again. In the experiments on the public trie method, we randomly pick the training set from the corpus.

Then the trie is used to compress all the files. A pruning algorithm is also implemented and preliminary results are shown.

5.3.2 Performance comparison

We compared the original LZW implementation tzip with our off-line version trzip and the public trie version trdzip. The trie size is 9604bytes for a 12-bit trie, 20836 bytes for a 13-bit trie, and 44356 bytes for a 14-bit trie. trzip uses the beginning part of the text to build the dictionary trie and uses the trie to compress the whole file from the first symbol. trdzip loads the dictionary trie from the file that is stored after selection or training. The compression ratio is high when the trie size is selected to be relatively small. We justify our idea with small size trie and then pick the size with best performance in further experiment using larger trie size.

We test the overall compression ratio for the LZW, off-line LZW, and public trie LZW with different trie sizes of 12, 13, and 14 bits respectively. The overall compression is the sum of the compressed file sizes over the sum of the raw text file sizes. The off-line LZW has a slightly smaller compression ratio (0.75%) than the LZW since the beginning part of the text is compressed using the trie trained from itself. The trie size is not included in the individual file because it is fixed no matter how big the file is in the current implementation. The LZW with public trie has a slightly worse compression ratio than LZW (about 7%). However, when we combine the whole corpus together so that we have a universal index over the whole collection instead of breaking the collection into small files, the public trie method has the best compression ratio.

The overall and the average compression ratio are better than both LZW and off-line LZW algorithms. The overall compression ratio is given using the trie from the sample file in the corpus that gives the best overall compression. The ratio is computed using the sum of the compressed file size divided by the total corpus size. The average compression ratio is given using the trie from the sample file in the corpus that gives the best average compression. The ratio is computed as the average of the compression ratio of the individual files.

Figure 5.8 illustrates that the encoding time is linearly increasing with the text file size. Our algorithm simply makes another pass that linearly scans through the file after we use partial text to build the trie. If using a public trie, we do not need to build a trie from the beginning. Our algorithm will still be an online one with a predefined dictionary. Although the pattern matching time is not tested here using a large public trie, the time complexity is not expected to significantly improve since the hashing is used to refer to the node. Figure 5.9 indicates that the compression ratio is uniformly getting better when the trie size is chosen to be larger. Since the trie is trained using only a small portion of the text and no updating occurred during compression, the compression ratio is roughly constant with the file size. We implement and test the results when the trie is trained using a large collection of the text.

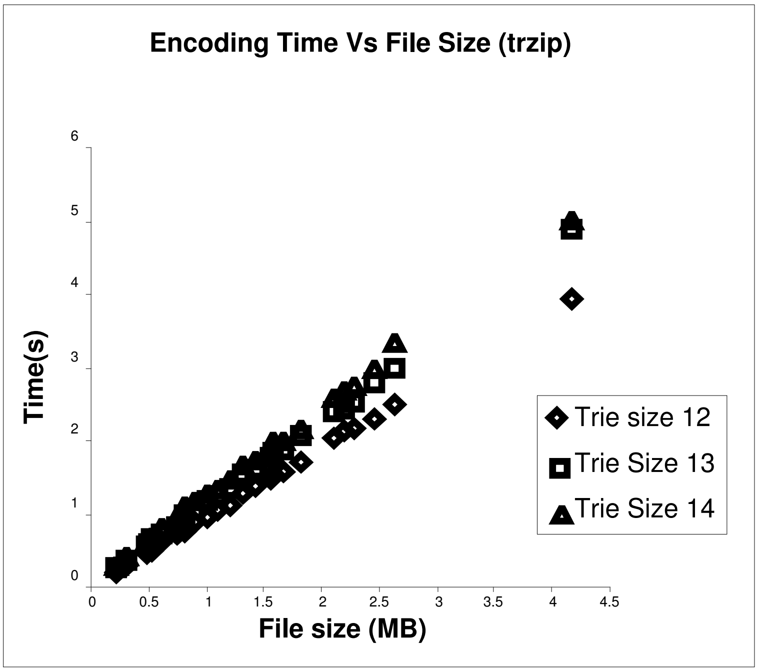


Figure 5.8: Encoding time for the modified algorithm vs. file size.

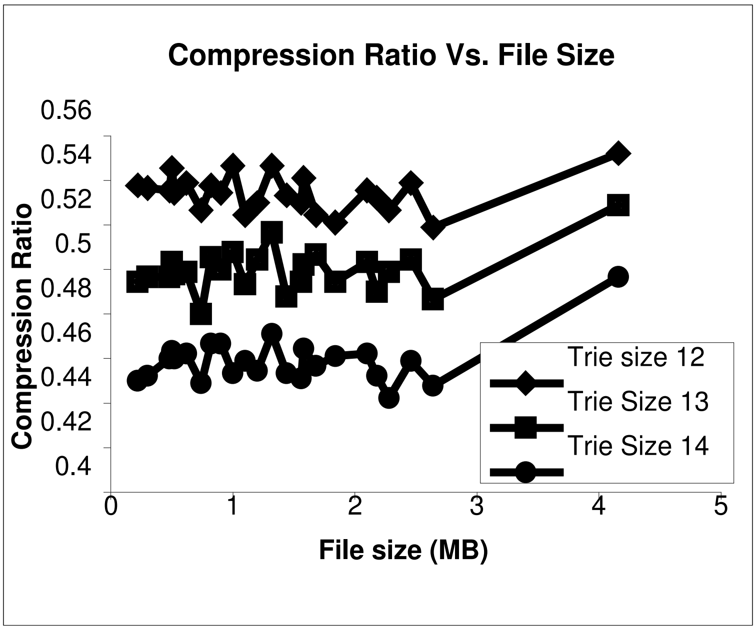


Figure 5.9: Compression ratio with different file sizes.

Figure 5.10 shows more choices of different code size (trie size). The results indicate that the 16 bit code has the best compression ratio in our corpus. Such a byte level coding also brings the advantage of easy random access and parallel processing during retrieval and decoding. The size of the trie is 64k bytes. This is much smaller than an English dictionary. We randomly pick the files from the corpus for the training purpose. The training file size is usually around 4Mb. The average compression ratio for our corpus is 0.346 compared with 0.297 from the Word Huffman based compression described in Managing Gigabytes (MG system)[WMB99]. We also tested on the preprocessing of text using LIPT. Using the same training set that is transformed, the compression ratio is improved to 0.32. The compression ratio by gzip is 0.335 for our corpus. It is worth to mention that in MG system, the compression is based on the statistics of the whole corpus while ours is based upon a small portion of the text.

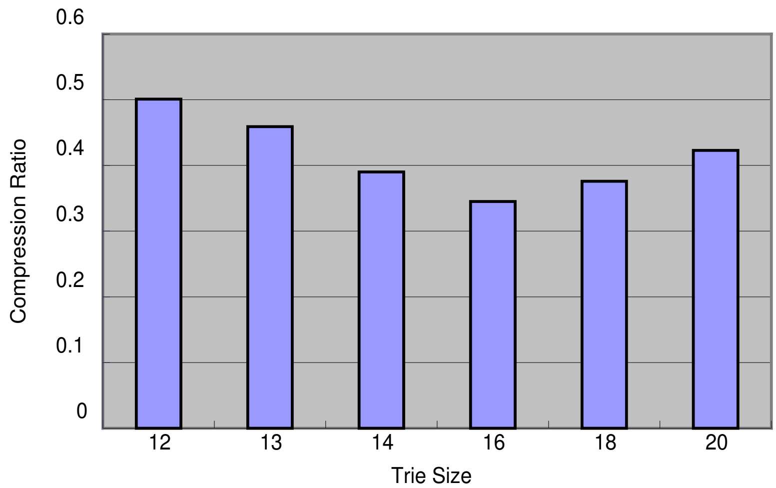


Figure 5.10: Compression ratio vs. trie size (code length).

An algorithm has been designed to find a public trie for an overall optimal compression ratio although it is not the primary concern of a compressed text retrieval system. We build the trie for a file that gives the best average compression ratio for the corpus with a 14 bit trie size. Then, we gradually prune the leaf nodes with least frequency until the trie reaches 12 bit size. Figure 14 shows that the average compression ratio is better than selecting the best trie with 12 bit trie size without pruning. It also shows that if each file is compressed using its own pruned trie. The compression is even better. This justifies that our idea of keeping the most frequent words in the trie is effective.

Considering the method of using a public trie, we have shown that the compression ratio is relatively stable even if we use a trie from the other files without further training process as shown in Figure 5. We pick a random subset of the corpus as training set with the size of 4Mb. The pruning algorithm also shows the improvement on the compression ratio. After trying different combination of the initial and ultimate trie size, we conclude that the initial trie size chosen to be 18 or 20 bits and pruned to 16 give the best compression ratio. The 20 to 16 bit is slightly better with minimal margin.

It should be pointed out that our public trie method makes little change to the current indexing method using inverted indexing file. Moreover, comparing to the MG system that has a fixed lowest context resolution, for example, paragraphs, we do not need to recompress if the resolution is required to be higher. We just need to build the new pointers for the boundaries with higher granularity at byte level code. If we do not recompressed the text using MG system, the new pointers will point to the bit level positions in the compressed text. Hence the size of the pointer will be larger that requires more space for the index file than our solution.

Figure 5.11 shows the partial decoding times for different file sizes given a location index and the number of nodes to decode. We can observe that for a given file size, the search time is roughly linear in the number of nodes to decode. Since each node represents a substring of the text, the length of the substring varies. Hence the decoding time is not exactly linear. In the current implementation, locating the starting nodes takes longer for larger files. This is because the current implementation uses a linear search to locate the starting node of the block. In our implementation, we search the file as it is being loaded. It is possible to use binary search to reduce the effect of the file size on the decoding time if the file is already in the main memory. In practical situation, the file is usually loaded at the time of search. In fact, once the file is loaded into the memory, direct access can be used to locate the start of the block in constant time. The emphasis here is to show that the decoding time varies linearly with block size.

During decoding stage, our method requires only the storage of the public trie and the space for the text in the stream that requires minimal space. We need 64 Kb space for the 16 bit trie. The XRAY system claims to have a 30 MB space requirement. We implement the retrieval system based on the MG system. The dictionary is replaced by the public trie and the pointers. We use the same document ranking function and query evaluation with MG system. The decoding speed is around 5 Mb per second for a single CPU that is similar

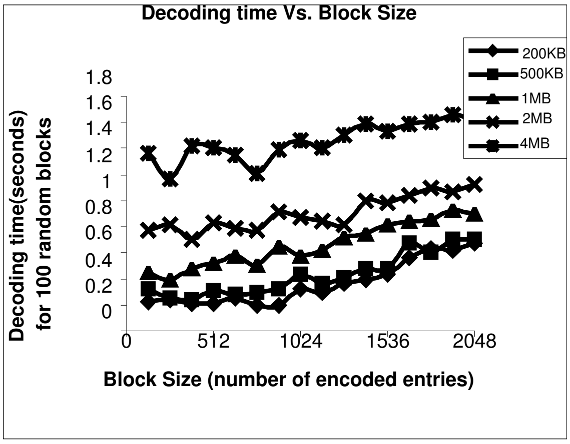


Figure 5.11: The performance of the partial decoding time for different file sizes given a location index and the number of nodes to decode.

to the MG system. That is, browsing a paragraph or a small file takes insignificant amount of time. We expect to have better performance for the multiple processor system.

5.4 Conclusion

We have shown that random access and partial decoding can be done efficiently on compressed text with a few modifications to the LZW algorithm. We obtain the ability to randomly access any part of the text given the location index. Compared to a decompress-then-search method with a complexity of O(z+u), where z is the compressed file size and u is the decoded file size, our method performs in a sublinear complexity since we only decode a small portion of the file. Moreover, our modified method improved the compression ratio compared to original LZW.

We have justified the feasibility of using a public (static) trie by selecting a trie from a part of the text collection. A better trie that leads to an overall improved compression ratio could be obtained by some more fine- tuned training algorithms, and we suggest use a large data set to train the trie. As we can see in the experiments, the trie is filled quickly and updating stops because the trie is set to have a fixed size. There are several ways to solve this problem. For example, we can update the trie by some rules that may add new nodes and remove current nodes - a pruning algorithm can be used to decide which node should be removed. A popular pruning standard is to remove the least frequently used node, so that the rest of the nodes have a higher access frequency. There are other factors that can be considered, such as the order of the removal, the length of the subsequence represented by the node. Since all of the training is performing off-line. We can apply algorithms that reach the optimal compression ratio while the time complexity is not a major concern. Another method comes from the observation of Figure 5. The trie may reflect the statistical property for different categories of the text. Text from different author, publisher, or language may have different probability distributions. We can train several tries for compression. In the indexing stage, there is no change needed. When partial decompression is needed, we only need to read a flag in the compression file that is stored in the header during the compression stage. Then pick the corresponding trie to decode. Since each trie size is just a few kilobytes in the current implementation, the space for several tries is still minimal compared to the text collection.

Comparing with the Word-Huffman approach, the trained trie can capture the internal context property for the given training set independent of the language. Although a substring of a node may not represent a "word" exactly, we can use a very simple algorithm to detect the actual word for a language since the new algorithm provides the best locality, namely, we only need to check a very small neighborhood.

Even though we use the selected trie without fine-tuned training, the compression performance is comparable to gzip, Word Huffman and XRAY. Most importantly, we obtained the flexibility of indexing on various details of the texts and the ability of random access to any part of the text and decode the small portion comparing with the whole document. Parallel processing can be performed on our byte level code without inherited difficulty. The XRAY and word Huffman schemes are based on the Huffman code for the final compression output. "The sequential algorithm for Huffman coding is quite simple, but unfortunately it appears to be inherently sequential. Its parallel counterpart is much more complicated" [CR94]. In our algorithm, each node is encoded in 16 bits (2 bytes). We can start decoding from any even number byte after the header in the sequence to obtain the correct original text. Therefore it is more convenient for our compression scheme to provide a parallel access to the compressed code. We have simulated the parallel decoding of the compressed text by using multiple threading in the Unix system. Each thread can read any part of the compressed text without confliction with the other threads

Our method also makes the file more error resilient. Once there is an error occurred during transmission or by other reasons. We can discard or ignore the current double bytes sector and continue decoding without rippling the error. Since all the nodes are referred to the public trie, the decoding is not dependant on the history in the same file.

CHAPTER 6

CONCLUSION AND FUTURE WORKS

In this dissertation, we have presented the novel algorithms for lossless text compression using transformation with static dictionary. The compression ratio improved significantly. meanwhile, by employing ternary tree data structure, the extra step only brings insignificant delay. The overall performance is improved in the scenario of massive data transmission such as internet. Less bandwidth are required and the encoding/decoding workload are distributed over the computers in different locations.

The limitation of the transforms is due to the static dictionary mapping. Currently only English dictionary is used and tested on the limited amount of the text data. There would be text in different language that cannot be transformed with the current dictionary. New dictionary should be used to adopt the change. However, it will bring extra cost of the dictionary storage and code length as well. Moreover, words in a specific domain would have higher frequency than usual in the domain-specific texts. For example, biological works includes for biological taxonomies than the articles in ordinary newspaper. Therefore, we should take special care of such situation.

We also presented the compressed domain pattern matching based on BWT text. The methods is better than full decompression with the cost of the auxiliary arrays. If the text is frequently searched, the proposed methods will greatly improve the searching. Searching BWT text with wild card symbols in the given pattern and regular expression are still need to be taken care of. Long sequence matching might be a special problem since more and more applications on biological information retrieval are using long patterns as DNA sequences.

Our novel approach to incorporate the current compression algorithm and the index based text retrieval provide a solution to handle the compression and partial decompression in the low level without touching the high level query evaluation. The current system can be adopted to the MLZW based compression efficiently by changing the content of the pointers. Further works would be finding a better public trie training algorithm to obtain better compression. Domain specific trie could also be developed to handle text from different backgrounds.

LIST OF REFERENCES

[AB92] A. Amir and G. Benson. "Efficient two-dimensional compressed matching." In *Proceedings of IEEE Data Compression Conference*, pp. 279–288, 1992.

[ABF96a] A. Amir, G. Benson, and M. Farach. "Let sleeping files lie: Pattern matching in Z-compressed files." *Journal of Computer and System Sciences*, **52**:299–307, 1996.

[ABF96b] A. Amir, G. Benson, and M. Farach. "Let sleeping files lie: Pattern matching in Z-compressed files." *Journal of Computer and System Sciences*, **52**:299–307, 1996.

[ABF97] A. Amir, G. Benson, and M. Farach. "Optimal two-dimensional compressed matching." *Journal of Algorithms*, **24**:354–379, 1997.

[AC75] A. V. Aho and M. J. Corasick. "Efficient string matching: An aid to bibliographic search." *Comm. ACM*, **18**(6):333–340, 1975.

[AC96] A. Amir and G. Calinescu. "Alphabet independent and dictionary scaled matching." *Combinatorial Pattern Matching, LNCS 1075*, pp. 320–334, 1996.

[AGS99] M. Atallah, Y. Génin, and W. Szpankowski. "Pattern matching image compression: algorithmic and experimental results." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **21**:618–627, 1999.

[Aku94] T. Akutsu. "Approximate string matching with don't care charaters." *Proceedings, Combinatorial Pattern Matching, LNCS 807*, pp. 240–249, 1994.

[AL96] G. Ahanger and T. D. C. Little. "A survey of technologies for parsing and indexing digital video." *Journal of Visual Communication and Image Representation*, **7**(1):28–43, 1996.

[AL97] D. A. Adjeroh and M. C. Lee. "Robust and efficient transform domain video sequence analysis: An approach from the generalized color ratio model." *Journal of Visual Communication and Image Representation*, **8**(2):182–207, 1997.

[ALK99] D. A. Adjeroh, M. C. Lee, and I. King. "A distance measure for video sequence similarity matching." *Computer Vision and Image Understanding*, **75**(1):25–45, 1999.

[ALV92] A. Amir, G.M. Landau, and U. Vishkin. "Efficient pattern matching with scaling." *Journal of Algorithms*, **13**:2–32, 1992.

[AMB02] D. A. Adjeroh, A. Mukherjee, T.C. Bell, M. Powell, and N. Zhang. "Pattern matching in BWT-transformed text." *Proceedings, IEEE Data Compression Conference*, p. 445, 2002.

[Arn00] Ziya Arnavut. "Move-to-Front and Inversion Coding." In *Proceedings of the Conference on Data Compression*, p. 193. IEEE Computer Society, 2000.

[ASG00] M. Alzina, W. Szpankowski, and A. Grama. "2D-Pattern matching image and video compression: theory,algorithms, and experiments." *IEEE Transactions on Image Processing*, **9**(8), 2000.

[Bak78] T.P. Baker. "A technique for extending rapid exact-match string matching to arrays of more than one dimension." *SIAM Journal on Computing*, **7**(4):533–541, November 1978.

[BCA98] P. Barcaccia, A. Cresti, and S. De Agostino. "Pattern Matching in text compressed with the ID Heuristic." In *Proceedings of IEEE Data Compression Conference*, pp. 113–118, 1998.

[BG92] R. Baeza-Yates and G. H. Gonnet. "A new approach to text searching." *Communications of the ACM*, **35**(10):74–82, 1992.

[Bir77] R. S. Bird. "Two dimensional pattern matching." *Information Processing Letters*, **6**(5):168–170, October 1977.

[BK93] T.C. Bell and D. Kulp. "Longest-match string searching for Ziv-Lempel compression." *Software—Practice and Experience*, **23**(7):757–772, July 1993.

[BK00] B. Balkenhol and S. Kurtz. "Universal data compression based on the Burrows and Wheeler-transformation: Theory and practice." *IEEE Transactions on Computers*, 2000.

[BKL96a] Piotr Berman, Marek Karpinski, Lawrence Larmore, Wojciech Plandowski, and Wojciech Rytter. "The Complexity of Two-Dimensional Compressed Pattern Matching." Technical Report TR-96-051, International Computer Science Institute, Berkeley, CA, December 1996.

[BKL96b] Piotr Berman, Marek Karpinski, Lawrence Larmore, Wojciech Plandowski, and Wojciech Rytter. "The complexity of Two-Dimensional Compressed Pattern-Matching." Technical Report 85156-CS, University of Bonn, Department of Computer Science, August 1996.

[BKL97] P. Berman, M. Karpinski, L.L. Larmore, W. Plandowski, and W. Rytter. "On the complexity of pattern-matching for highly compressed two-dimensional texts." In *Combinatorial Pattern-Matching, 8th Annual Symposium*, 1997. LNCS 1264.

[BKS99] B. Balkenhol, S. Kurtz, and Y.M. Shtarkov. "Modofications of the Burrows and Wheeler data compression algorithm." *Proceedings, IEEE Data Compression Comference*, 1999.

[BM77] R.S. Boyer and J.S. Moore. "A fast string searching algorithm." *Communications of the ACM*, **20**(10):762–772, October 1977.

[BM89] T.C. Bell and A. Moffat. "A note on the DMC data compression scheme." *Computer Journal*, **32**(1):16–20, February 1989.

[BP92] R. Baeza-Yates and C.H. Perleberg. "Fast and practical approximate string matching." *Proceedings, Combinatorial Pattern Matching, LNCS 644*, pp. 185–192, may 1992.

[BPM02] T.C. Bell, M. Powell, A. Mukherjee, and D. A. Adjeroh. "Searching BWT compressed text with the Boyer-Moore algorithm and binary search." *Proceedings, IEEE Data Compression Conference, 2002*, pp. 112–121, 2002.

[BR99a] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley, Harlow, England, 1999.

[BR99b] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[BS90] H. Bunke and A. Sanfeliu, editors. *Syntatic and Structural Pattern Recognition: Theory and Applications.* World Scientific, Singapore, 1990.

[BS97] J. L. Bentley and R. Sedgewick. "Fast algorithms for sorting and searching strings." In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pp. 360–369, 1997.

[BST86a] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. "A locally adaptive data compression scheme." *Communications of the ACM*, **29**:320–330, April 1986.

[BST86b] J L Bentley, D D Sleator, R E Tarjan, and V Wei. "A locally adaptive data compression scheme." *cacm*, **29**(4):320–330, 1986.

[BW94a] M. Burrows and D.J. Wheeler. "A block-sorting lossless data compression algorithm." Technical report, Digital Equipment Corporation, Palo Alto, California, 1994.

[BW94b] M. Burrows and D.J. Wheeler. "A block-sorting lossless data compression algorithm." Technical report, Digital Equipment Corporation, Palo Alto, California, 1994.

[Ca94] M. Crochemore and et al. "Speeding up two string-matching algorithms." *Algorithmica*, **12**:247–267, 1994.

[CH87] G.V. Cormack and R.N. Horspool. "Data compression using dynamic Markov modeling." *Computer Journal*, **30**(6):541–550, December 1987.

[CH93] Linda A Curl and Brent J Hussing. "Introductory computing: a new approach." In *Proc. SIGCSE 93*, pp. 131–135, March March 1993.

[CL92] W. I. Chang and J. Lampe. "Theoretical and empirical analysis of approximate string matching algorithms." *Proceedings, Combinatorial Pattern Matching, LNCS 644*, pp. 175–184, 1992.

[CL94] W. I. Chang and E. L. Lawler. "Sublinear approximate string matching and biological applications." *Algorithmica*, **12**:327–344, 1994.

[CL96] M. Crochemore and T. LeCroq. "Pattern-matching and text compression." *ACM Computing Surveys*, **28**(1):39–41, March 1996.

[CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[Cor] Gutenberg Corpus. "The Gutenberg Corpus, http://www.promo.net/pg/.".

[Cor00a] Calgary Corpus. "The Calgary Corpus, ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus." 2000.

[Cor00b] Canterbury Corpus. "The Canterbury Corpus, http://corpus.canterbury.ac.nz." 2000.

[CR94] M. Crochemore and W. Rytter. *Text algorithms*. Oxford Press, New York, 1994.

[CS94] C. Constantinescu and J. Storer. "Improved techniques for single-pass adaptive vector quantization." *Proceeding of the IEEE*, **82**:933–939, 1994.

[CT97] J.G. Cleary and W.J. Teahan. "Unbounded length contexts for PPM." *The Computer Journal*, **36**(5):1–9, 1997.

[CW84] J.G. Cleary and I.H. Witten. "Data compression using adaptive coding and partial string matching." *IEEE Transactions on Communications*, **COM-32**:396–402, April 1984.

[CW02] A. Cannane and H. E. Williams. "A general-purpose comrpession scheme for large collections." *ACM Trans. on Information Systems*, **20**:329–355, 2002.

[Eff00] M. Effros. "PPM performance with BWT Complexity: A fast and effective data compression algorithm." *Proceedings of the IEEE*, **88**(11):1703–1712, 2000.

[EV88] T. Eilam-Tzoreff and U. Vishkin. "Matching patterns in strings subject to multi-linear transformations." *Theoretical Computer Science*, **60**:231–254, 1988.

[EV02] M. Effros and K. Vusweswariah. "Universal lossless source coding with the Burrows-Wheeler Transform." *IEEE Transactions on Information Theory*, **48**(5):1061, 2002.

[FB92a] W.B. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[FB92b] William B. Franks and Ricardo Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice Hall PTR, 1992.

[Fen96a] P. Fenwick. "Block Sorting Text Compression." In *Proceedings of the 19th Australasian Computer Science Conference*, pp. 193–202, 1996.

[Fen96b] P. Fenwick. "The Burrows-Wheeler Transform for block sorting text compression." *The Computer Journal*, **39**(9):731–740, September 1996.

[Fen96c] P. Fenwick. "The Burrows-Wheeler Transform for block sorting text compression." *The Computer Journal*, **39**(9):731–740, September 1996.

[FM96] R. Franceschini and A. Mukherjee. "Data compression using encrypted text." In *Proceedings of the Third Forum on Research and Technology, Advances in Digital Libraries*, pp. 130–138, 1996.

[FM00] P. Ferragina and G. Manzini. "Opportunistic Data Structures with applications." *Proceedings, 41st IEEE Symposium on Foundations of Computer Science, FOCS'2000*, 2000.

[FM01] P. Ferragina and G. Manzini. "An experimental study of an opportunistic index." *Proceedings, 12th ACM-SIAM Symposium on Discrete Algorithms, SODA'2001*, 2001.

[FT95a] Martin Farach and Mikkel Thorup. "String matching in Lempel-Ziv compressed strings." In *Proceedings of the twenty-seventh annual ACM symposium on the Theory of Computing*, pp. 703–712, New York, May 1995. ACM.

[FT95b] Martin Farach and Mikkel Thorup. "String matching in Lempel-Zive compressed strings." In *Proceedings of the twenty-seventh annual ACM symposium on the Theory of Computing*, pp. 703–712, New York, May 1995. ACM.

[FT98] M. Farach and M. Thorup. "String matching in Lempel-Ziv compressed strings." *Algorithmica*, **20**:388–404, 1998.

[GG97] R. Giancarlo and R. Gross. "Multi-dimensional pattern matching with dimensional wildcards: Data structures and optimal on-line search algorithm." *Journal of Algorithms*, **24**:223–265, 1997.

[GKP96a] L. Gąsieniec, M. Karpinski, W. Plandowski, and W. Rytter. "Randomized efficient algorithms for compressed strings: the finger-print approach." *Proceedings, Combinatorial Pattern Matching, LNCS 1075*, pp. 39–49, 1996.

[GKP96b] L. Gąsieniec, M. Karpinski, W. Plandowski, and W. Rytter. "Randomized efficient algorithms for compressed strings: the finger-print approach." *Proceedings, Combinatorial Pattern Matching, LNCS 1075*, pp. 39–49, 1996.

[GP90] Z. Galil and K. Park. "An improved algorithm for approximate string matching." *SIAM Journal of Computing*, **19**(6):689–999, 1990.

[Gus97] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[GV00] R. Grossi and J. Vitter. "Compressed suffix arrays and suffix trees with applications to text indexing and string matching." *Proceedings, 32nd ACM Symposium on Theory of Computing*, 2000.

[HS91] A. Hume and D. Sunday. "Fast string searching." *Software—Practice and Experience*, **21**(11):1221–1248, November 1991.

[Huf52] D.A. Huffman. "A method for the construction of minimum redundancy codes." *Proc. IRE*, **40**(9):1098–1101, September 1952.

[KB00] S. Kurtz and B. Balkenhol. "Space-efficient linear time computation of the Burrows and Wheeler-transfrmation." Technical report, Technishe Fakultat, Universitat Bielefeld, 2000.

[KF93] H. U. Khan and H. A. Fatmi. "A novel approach to data compression as a pattern recognition problem." *Proceedings of IEEE Data Compression Conference*, 1993.

[KM98] Holger Kruse and Amar Mukherjee. "Preprocessing Text to Improve Compression Ratios." In *"Proc. IEEE Data Compression Conference"*, 1998.

[KM99] J. R. Knight and E. W. Myers. "Super-pattern matching." Technical Report TR-92-29, Department of Computer Science, University of Arizona, 1999.

[KMP77] D.E. Knuth, J.H. Morris, and V.R. Pratt. "Fast pattern matching in strings." *SIAM Journal of Computing*, **6**(2):323–350, June 1977.

[KNU00a] J. Kärkkäinen, G. Navarro, and E. Ukkonen. "Approximate string matching Ziv-Lempel compressed text." *Proceedings, Combinatorial Pattern Matching, LNCS 1848*, pp. 195–209, 2000.

[KNU00b] J. Kärkkäinen, G. Navarro, and E. Ukkonen. "Approximate string matching Ziv-Lempel compressed text." *Proceedings, Combinatorial Pattern Matching, LNCS 1848*, pp. 195–209, 2000.

[Kos95] S.R. Kosaraju. "Pattern matching in compressed texts." In P.S. Thiagarajan, editor, *Proceedings, Foundations of Software Technology and Theoretical Computer Science, 15th Conference*, pp. 349–362. Springer-Verlag, 1995.

[KPR95] M. Karpinski, W. Plandowski, and W. Rytter. "The Fully Compressed String Matching for Lempel-Ziv Encoding." Technical Report 85132-CS, Department of Computer Science,University of Bonn, apr 1995.

[KR81] R. M. Karp and M. O. Rabin. "Efficient randomized pattern matching algorithms." Technical Report TR-31-8, Aiken Computation Lab, Harvard University, 1981.

[KT00] M. Kobayashi and K. Takeda. "Information retrieval on the web." *ACM Computing Surveys*, **32**(2):144–173, 2000.

[KTS99] T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. "Shift-And approach to pattern matching in LZW compressed text." *Proceedings, Combinatorial Pattern Matching, LNCS 1645*, pp. 1–13, 1999.

[KU98] J. Kärkkäinen and E. Ukkonen. "Lempel-Ziv index for q-grams." *Algorithmica*, **21**:137–154, 1998.

[Lar99] N.J. Larsson. "The context trees of block sorting compression." *Proceedings, IEEE Data Compression Conference*, pp. 189–198, mar 1999.

[LKP97] J. S. Lee, D. K. Kim, K. Park, and Y. Cho. "Efficient algorithms for approximate string matching with swaps." *Proceedings, Combinatorial Pattern Matching, LNCS 1264*, pp. 28–39, 1997.

[LS97] T. Luczak and W. Szpankowski. "A suboptimal lossy data compression based on approximate pattern matching." *IEEE Transactions on Information Theory*, **43**:1439–1451, 1997.

[LV88] G. M. Landau and U. Vishkin. "Fast string matching with k differences." *Journal of Computer and System Sciences*, **37**:63–78, 1988.

[LV94] G. M. Landau and U. Vishkin. "Pattern matching in a digitized image." *Algorithmica*, **12**(4/5):375–408, 1994.

[LVS03] P. Lyman, H. R. Varian, Kirsten Swearingen, Peter Charles, Nathan Good, Laheem Lamar Jordan, and Joyojeet Pal. "How Much Information? 2003, http://www.sims.berkeley.edu/research/projects/how-much-info-2003/." Technical report, School of Information Management and Systems at the University of California at Berkeley, 2003.

[MA94] A. Mukherjee and T. Acharya. "Compressed pattern-matching." In *Proceedings of IEEE Data Compression Conference*, p. 468, 1994.

[MA95] A. Mukherjee and T. Acharya. "VLSI Algorithms for compressed pattern search using tree based codes." In *Proceedings, International Conference on Application Specific Array Processors*, pp. 133–136, 1995.

[MA02] A. Mukherjee and F. Awan. "Text Compression." In Khalid Sayood, editor, *Lossless Compression Handbook*, chapter 10. Academic Press, 2002.

[Maa93] C.-Y. Maa. "Identifying the existence of bar codes in compressed images." In *"Proceeding of IEEE Data Compression Conference"*, p. 457, 1993.

[Man97] U. Manber. "A text compression scheme that allows fast searching directly in the compressed file." *ACM Transactions on Information Systems*, **15**(2):124–136, April 1997.

[Man99] G. Manzini. "An analysis of the Burrows-Wheeler transform." Technical Report B4-99-13, Universita Del Piemonte Orientale, Instituto di Mathematica Computazionale, 1999.

[McC76] E. M. McCreight. "A space-economical suffix tree construction algorithm." *Journal of the ACM*, **23**(2):262–272, 1976.

[MIP99] M. K. Mandal, F. Idris, and S. Panchanathan. "A Critical Evaluation of Image and Video Indexing Techniques in the Compressed Domain." *Journal of Image and Vision Computing*, **17**(7):513–529, 1999.

[MM93] U. Manber and G. Myers. "Suffix Arrays: A new method for on-line string searches." *SIAM Journal of Computing*, **22**(5):935–948, 1993.

[MNB00] E. S. Moura, G. Navarro, and R. Baeza-Yates. "Fast and flexible word searching on compressed text." *ACM Transactions on Information Systems*, **18**(2):113–139, 2000.

[MNU01] V. Mäkinen, G. Navarro, and E. Ukkonen. "Approximate matching of run-length compressed strings." *Proceedings, Combinatorial Pattern Matching*, pp. 31–49, 2001.

[MNZ00] E. S. 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.

[Mof90a] A. Moffat. “Implementing the PPM data compression scheme.” *IEEE Transactions on Communications*, **38**(11):1917–1921, November 1990.

[Mof90b] A. Moffat. “Linear time adaptive arithmetic coding.” *IEEE Transactions on Information Theory*, **36**(2):401–406, March 1990.

[MS90] M. Mongeau and D. Sankoff. “Comparison of musical sequencs.” *Computers and the Humanities*, **24**:161–175, 1990.

[MSW93a] A. Moffat, R. Sacks-Davis, R. Wilkinson, and J. Zobel. “Retrieval of partial documents.” In *Proc. TREC Text Retrieval Conference*, pp. 181–190, 1993.

[MSW93b] A. Moffat, N.B. Sharman, I.H. Witten, and T.C. Bell. “An empirical evaluation of coding methods for multi-symbol alphabets.” In J.A. Storer and M. Cohn, editors, *Proc. IEEE Data Compression Conference*, pp. 108–117, Snowbird, Utah, March 1993. IEEE Computer Society Press, Los Alamitos, California.

[MW85] V. Miller and M. Wegman. “Variations on a theme by Ziv and Lempel.” In A Apostolico and Zvi Galil, editors, *Combinatorial Algorithms on Words, Volume 12*, NATO ASI Series F, Berlin, 1985. Springer-Verlag.

[MW92] Udi Manber and Sun Wu. “Some Assembly Required. Approximate Pattern Matching: Agrep’s algorithms let you perform text searches using an approximate pattern.” *Byte Magazine*, **17**(12):281, November 1992.

[MW94] Udi Manber and Sun Wu. “GLIMPSE: A Tool to Search Through Entire File Systems.” In USENIX Association, editor, *Proceedings of the Winter 1994 USENIX Conference: January 17–21, 1994, San Francisco, California, USA*, pp. 23–32, Berkeley, CA, USA, Winter 1994. USENIX.

[Mye94] E. W. Myers. “A sublinear algorithm for approximate keyword searching.” *Algorithmica*, **12**:345–374, 1994.

[MZ94] A. Moffat and J. Zobel. “Self-indexing inverted files.” In *Proc. Australasian Database Conference*, Christchurch, New Zealand, January 1994. World Scientific. To appear.

[Nav01] G. Navarro. “NR-grep: A fast and flexible pattern matching tool.” *Software – Practice and Experience*, (31):1265–1312, 2001.

[Nav02] G. Navarro. “Indexing Text using the Ziv-Lempel Trie.” In *Proceedings of 9th String Processing and Information Retrieval (SPIRE’02), LNCS 2476, 2002. Extended version to appear in J. of Discrete Algorithms.*, pp. 325–336, 2002.

[NO00] H. Ney and S. Ortmanns. "Progress in dynamic programming search for LVCSR." *Proceedings of the IEEE*, **88**(8):1224–1240, 2000.

[NR99a] G. Navarro and M. Raffinot. "A general practical approach to pattern matching over Ziv-Lempel compressed text." *Proceedings, Combinatorial Pattern Matching, LNCS 1645*, pp. 14–36, 1999.

[NR99b] G. Navarro and M. Raffinot. "A general practical approach to pattern matching over Ziv-Lempel compressed text." *Proceedings, Combinatorial Pattern Matching, LNCS 1645*, pp. 14–36, 1999.

[NR00] G. Navarro and M. Raffinot. "Fast and flexible string matching by combining bit-parallelism and suffix automata." *ACM Journal of Experimental Algorithms*, **5**(4), 2000.

[NT00] G. Navarro and J. Tarhio. "Boyer-Moore string matching over Ziv-Lempel compressed text." *Proceedings, Combinatorial Pattern Matching, LNCS 1848*, pp. 166–180, 2000.

[OM88] O. Owolabi and D. R. McGregor. "Fast approximate string matching." *Software – Practice and Experience*, **18**:387–393, 1988.

[PW93] P. A. Pevzner and M. S. Waterman. "A fast filtration algorithm for the substring matching problem." *LNCS 684, Combinatorial Pattern Matching*, pp. 197–214, 1993.

[RC94] I. Rigoutsos and A. Califano. "Searching in parallel for similar strings." *IEEE Computational Science and Engineering*, pp. 60–67, summer issue 1994.

[Ris79] J.J. Rissanen. "Arithmetic codings as number representations." *Acta. Polytech. Scandinavica*, **Math 31**:44–51, 1979.

[RL79] J. Rissanen and G. G. Langdon. "Arithmetic coding." *IBM Journal of Research and Development*, **23**(2):149–162, 1979.

[Sad00] K. Sadakane. "Compressed text databases with efficient query algorithms based on the compressed suffix array." *Proceedings, ISAAC'2000*, 2000.

[Sal00] David Salomon. *Data Compression: The Complete Reference*. Springer-Verlag, 2nd edition, 2000.

[Say00] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 2nd edition, 2000.

[SC78] H. Sakoe and S. Chiba. "Dynamic programming algorithm optimization for spoken word recognition." *IEEE Transactions on Acoustic, Speech and Signial Processing*, **26**(2):43–49, 1978.

[Sel80] P. Sellers. “The theory of computation of evolutionary distances: Pattern recognition.” *Journal of Algorithms*, **1**:359–373, 1980.

[Sew01] J. Seward. “Space-time tradeoffs in the Inverse B-W Transform.” *Proceedings, IEEE Data Compression Conference*, pp. 439–448, 2001.

[SG93] Y. Steinberg and M. Gutman. “An algorithm for source coding subject to a fidelity criterion, based on string pattern matching.” *IEEE Transactions on Information Theory*, **39**:877–886, 1993.

[Sha48] C.E. Shannon. “A mathematical theory of communication.” *Bell Systems Technical Journal*, **27**:379–423, 623–656, 1948.

[Sha51] C.E. Shannon. “Prediction and entropy of printed English.” *Bell Systems Technical Journal*, **30**:55, 1951.

[SI99] K. Sadakane and H. Imai. “A cooperative distributed text database management method unifying search and compression based on the Burrows-Wheeler Transform.” *Proceedings, Advances in Database Technology*, (LNCS 1552):434–445, 1999.

[Sik97] T. Sikora. “The MPEG-4 Video Standard Verification Model.” *IEEE Transactions on Circuit and Systems for Video Technology*, **7**(1), 1997.

[SMT00] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa. “A Boyer-More Type algorithm for compressed pattern matching.” *Proceedings, Combinatorial Pattern Matching, LNCS 1848*, pp. 181–194–13, 2000.

[SS99] E. Sutinen and W. Szpankowski. “On the collapse of the q- gram filteration.” *Technical Report*, 1999.

[ST85] D.D. Sleator and R.E. Tarjan. “Amortized efficiency of list update and paging rules.” *Communications of the ACM*, **28**:202–208, 1985.

[ST96] E. Sutinen and J. Tarhio. “Filtration with q-samples in approximate string matching.” *Proceedings, Combinatorial Pattern Matching, LNCS 1075*, pp. 50–63, 1996.

[STS99] Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. “Pattern matching in text compressed by using antidictionaries.” *Proceedings, Combinatorial Pattern Matching, LNCS 1645*, pp. 37–49, 1999.

[SW98] C. E. Shannon and W. Weaver. *The Mathematical Theory of Communication.* University of Illinois Press, Champaign, 1998.

[Szp93] W. Szpankowski. “Asympotic properties of data compression and suffix trees.” *IEEE Transactions on Information Theory*, **39**:1647–1659, 1993.

[Tak94] T. Takaoka. "Approximate pattern matching with samples." In *Lecture Notes in Computer Science, Springer-Verlag*, pp. 234–242, 1994.

[Tak96] T. Takaoka. "Approximate pattern matching with grey scale values." In *Proceedings, CATS 96 (Computing: the Australasian Theory Symposium)*, pp. 196–203, 1996.

[TC96] W. J. Teahan and John G. Cleary. "The Entropy of English Using PPM-based Models." In *Data Compression Conference*, pp. 53–62, 1996.

[TM04] T. Tao and A. Mukherjee. "LZW based compressed pattern matching." In *"Proc. IEEE Data Compression Conference"*, 2004.

[TRE00] TREC. "Official webpage for TREC – Text REtrieval Conference series. http://trec.nist.gov." 2000.

[TY85] W. S. Tsai and S. S. Yu. "Atributed string matching with merging for shape recognition." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **7**(4):453–463, 1985.

[Ukk85] E. Ukkonen. "Finding approximate patterns in strings." *Journal of Algorithms*, **6**:132–137, 1985.

[Wat89] M. S. Waterman. *Mathematical Methods for DNA Sequences.* CRC Press, Boca Raton, Florida, 1989.

[Wel84] T.A. Welch. "A technique for high performance data compression." *IEEE Computer*, **17**:8–20, June 1984.

[WF74] A. Wagner and M. J. Fischer. "The string-to-string correction problem." *Journal of the ACM*, **21**:168–173, 1974.

[WM92a] S. Wu and U. Manber. "Fast text searching allowing errors." *Communications of the ACM*, **35**(10):83–91, October 1992.

[WM92b] Sun Wu and Udi Manber. "agrep — A Fast Approximate Pattern-Matching Tool." In USENIX Association, editor, *Proceedings of the Winter 1992 USENIX Conference: January 20 — January 24, 1992, San Francisco, California*, pp. 153–162, Berkeley, CA, USA, Winter 1992. USENIX.

[WMB99] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Morgan Kaufman, second edition edition, 1999.

[Yam02] Yamaguchi. "Test data compression using the Burrows Wheeler Transform ." *IEEE Transactions on Computers*, **51**(5), 2002.

[YK95] E. H. Yang and J. Kieffer. "On the performance of data compression algorithms based on string pattern matching." *IEEE Transactions on Information Theory*, **41**, 1995.

[YK96] E. H. Yang and J. Keiffer. "Simple universal lossy data compression schemes derived from Lempel-Ziv algorithm." *IEEE Transactions on Information Theory*, **42**:239–245, 1996.

[YL96] B. L. Yeo and B. Liu. "Rapid scene analysis in compressed video." *IEEE Transactions on Circuits and Systems for Video Technology*, **5**(6):533–, 1996.

[Yok97] H. Yokoo. "Data compression using a sort-based context similarity measure." *Computer Journal*, **40**(2/3):94–102, 1997.

[ZL77] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression." *IEEE Transactions on Information Theory*, **IT-23**:337–343, 1977.

[ZL78] J. Ziv and A. Lempel. "Compression of individual sequences via variable rate coding." *IEEE Transactions on Information Theory*, **IT-24**:530–536, 1978.

[ZMN00] N. Ziviani, E. S. Moura, G. Navarro, and R. Baeza-Yates. "Compression: A key for next generation text retrieval systems." *IEEE Computer*, **33**(11):37–44, 2000.

[ZT89] R.F. Zhu and T. Takaoka. "A technique for two-dimensional pattern matching." *Communications of the ACM*, **32**(9):1110–1120, September 1989.